2014

An Overview of Homotopy Type Theory and the Univalent Foundations of Mathematics

Lawrence Dunn III

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

AN OVERVIEW OF HOMOTOPY TYPE THEORY AND THE UNIVALENT FOUNDATIONS OF MATHEMATICS

By

LAWRENCE DUNN

A Thesis submitted to the
Department of Mathematics
in partial fulfillment of the
requirements to graduate with
Honors in the Major

Degree Awarded:
Spring Semester, 2014

Lawrence Dunn defended this thesis on April 24, 2014.
The members of the supervisory committee were:

Dr. Ettore Aldrovandi
Thesis Director

Dr. Mark van Hoeij
Committee Member

Dr. Piers Rawling
Outside Committee Member

TABLE OF CONTENTS

LIST OF TABLES

ABSTRACT

Homotopy type theory, the basis of "univalent foundations" of mathematics, is a formal system with intrinsic connections to computer science, homotopy theory, and higher category theory. Rooted in type theory, the theoretical basis of most modern proof assistants, the system admits an interpretation as a logical calculus for homotopy theory and suggests a foundational system for which abstract "spaces" – not unstructured sets — are the most primitive objects. This perspective offers both a computational foundational for mathematics and a direct method for reasoning about homotopy theory. We present here a broad contextual overview of homotopy type theory, including a sufficiently thorough examination of the classical foundations which it replaces as to make clear the extent of its innovation. We will explain that homotopy type theory is, loosely speaking and among other things, a programming language for mathematics, especially one with native support for homotopy theory.

Remark. *Admittedly, homotopy type theory is a very young field and this survey is rooted in a self-guided study of the literature. As such, it is not unthinkable that some technical inaccuracies are included or that some subtleties are glossed over. The author accepts full responsibility for any mistakes or omissions. Certainly there is understandably less nuance to be found below than in more specialized treatments, and we hope that wherever there are shortcomings, they do not significantly detract from the goal of introducing the ideas covered here to non-specialists.*

We owe our gratitude to the general homotopy type theory community for shedding light on this subject. This paper would not have been possible without the commentary of leading researchers, whose combined insight we have attempted to consolidate into this paper.

CHAPTER 1

INTRODUCTION

Mathematicians in the 20^{th} century witnessed the gradual refinement and formalization of set theory as the underlying foundation for mathematics. The notion of a *set*, apparently, is sufficiently flexible that mathematics itself can be logically derived from the axioms of set theory, at least in principle. But sets apparently have little to offer computer scientists, who are concerned less with the properties of mathematical objects and more with computable operations applied to various data. Now, data are not all of the same sort, and generally it makes sense to delineate between the different types of data according to what kinds of operations we intend to perform with them — integers can be added and multiplied, for instance, while characters are concatenated. For this reason, computer scientists are more accustomed to *type theory* which, while every bit as rigorous as set theory, is more in tune with such concerns, although not often used in pure mathematics. What we present here is a system of interest to scholars of both sorts: *homotopy* type theory. Homotopy type theory results from an interpretation of type theory into abstract homotopy theory – a category-theoretic examination of homotopy theory, itself a branch of topology. The resulting formal system is comparable to set theory in its capacity as a sort of universal language for mathematics, but it has a distinctly computational and geometric character.

This new system, a "work in progress"[1] formulated in just the last decade[2], represents a fundamentally different way of practicing mathematics than is captured by the usual set-theoretic treatment familiar to modern mathematicians. Our goal with this paper is to expand upon these differences, making explicit what might

[1] [Uni13] p. 1

[2] Initial investigations into homotopy type theory began around 2006 due to independent work by Awodey and Warren, and Voevodsky. See [Uni13] p. 4

be taken for granted in a more concise treatment and placing strong emphasis on potential *mis*interpretations of the ideas which define homotopy type theory. To accomplish this, we first make clear what is meant by "set-theoretic treatment," and we make clear the relationship between the so-called naïve set theory of everyday mathematical practice and its axiomatic counterpart, ZFC. Following this treatment of set theory, we proceed with something like a "tutorial" on type theory, which is a branch of both logic and theoretical computer science. We are concerned with one particular variant of type theory, one which has been developed to formalize intuitionistic mathematics, which we shall also address. Having established the principles of this type theory, we present the recent developments which have led to homotopy type theory, which is being studied by the recently-formed "univalent foundations" program as a promising alternative to set theory.

Although homotopy type theory is interesting as a foundation for mathematics, the theory is also interesting to researchers who are not immediately concerned with mathematical foundations, such as homotopy theorists, category theorists, and computer scientists. In particular, it is stressed that studying and exploiting the practical benefits of homotopy type theory does not necessarily commit one to a rejection of set theory as the "standard foundation" for mathematics — homotopy type theory does *absolish* everyday practice but in many ways *enriches* it with additional clarity and structure. This is true even though the intuitionistic mathematics that type theory formalizes is associated with a rejection of classical logic itself. However, there is at least one very distinct advantage associated with homotopy type theory as a foundational system: The natural computational basis of the theory allows for efficient computer formalization and verification of proofs. Indeed, researchers are currently formalizing homotopy theory and general mathematics into homotopy type theory rather than more classical foundations. Moreover, this work is being carried out in a computer environment, leading to the construction of libraries of formal definitions and verified proofs. Potential benefits of such research are expounded upon in the concluding remarks.

Currently, the most relevant literature seems to be partitioned into three categories: cursory articles, which accessibly introduce the basic ideas but necessarily lack much technical content[3]; technical papers which may be inaccessible to non-specialists; and a textbook ([Uni13]) recently made available which aims to teach the more practical aspects of homotopy type theory to mathematicians. What we offer here is, we hope, at least somewhat unique in its aim to cover as much broader context as possible, carefully emphasizing the innovation of the theory over a more classical approach. It is our hope that this overview may be of some small assistance to a broader audience venturing to understand homotopy type theory and its relationship to other mathematics, perhaps useful as a segue into more advanced and extensive material. Much of our account can be considered "informal formal mathematics," as we proceed much more casually than is seen in material written for specialists. Italicized words are used for stress, and quotations are often used to indicate where the author's phrasing is deliberately informal. While visited topics range from ZFC to abstract homotopy theory, our treatment is intended to be suitable for an advanced undergraduate audience in pure mathematics, and a reader lacking experience in the many topics addressed should be able to pick up at least the general ideas.

This paper is divided into three primary sections which ought to be read sequentially, followed by concluding remarks and three appendices. First, we present a brief overview of conventional axiomatic set theory – the formalization of everyday set theory using the axioms of ZFC as formalized in first-order predicate calculus. Then, we provide an introduction to the type theory of Martin-Löf, which brings us into contact with intuitionistic mathematics. We follow with an explanation of the homotopical interpretation of this type theory by means of Quillen model categories, the crucial development which has led to homotopy type theory. From here we describe the univalent foundations program, briefly motivating the so-called univalence axiom of Vladimir Voevodsky. We conclude with a brief assessment of

[3]For instance, [SA13]

applications and future prospects. The three appendices document, respectively, the axioms of ZFC, the formal definition of the some common objects in type theory, and sample code for the proof assistant COQ which demonstrates a simple computer-verified proof. Dividing the thesis into several sections assures that we keep our analysis broad, so as to be interesting and generally accessible to many different groups of readers. The motivations for each section are given below, with more specific descriptions given at the beginning of each section:

Section 1: Set theory and ZFC This section introduces some essential notions from axiomatic set theory and formal logic. What we gain with this brief exposition is not so much any special insight into the laws of classical mathematics, but exposure to how these laws are formalized. Historically, the formal codification of mathematics has developed somewhat retro-actively, so the formal systems examined here simply reflect the usual principles of everyday mathematics. But homotopy type theory is a departure from this line of thought, beginning from different "first principles," so it pays to examine the bedrock of classical mathematics with greater-than-usual scrutiny. Material set theory and its canonical axiomatization as ZFC are discussed, forming a backdrop against which we later contrast type theory. The two most important ideas covered in this section are this: the manner in which logic and set theory are kept separate by ZFC, and the technique of using pure sets to "encode" properties of mathematical objects (e.g. numbers and Cartesian products).

Section 2: Intuitionistic type theory This section introduces the major principles of type theory, which out of the other ideas covered in this thesis we assume to be the most new and exotic for most readers. Type theory is a formal system (actually, several related ones) primarily used in theoretical computer science as a device for reasoning about computer programming, strongly distinguished from ZFC in its admittance of difference "types" of objects which are each subject to (and defined by) their own list of rules. To the extent that mathematicians do not consciously treat all mathematical objects as fundamentally of the same sort, it can

be argued that mathematicians actually practice something like type theory. As a corollary, familiarizing oneself with type theory can add clarity to mathematical practice. The most innovative aspect (over classical foundations) of the particular theory we use here is that proofs themselves are treated as ordinary mathematical objects, allowing for a more subtle treatment of logic than is usually available. The reader should come away from this section with a general understanding of the mindset of the type theorist, in particular with regard to these key ideas: the manner in which logic is subsumed by type theory, and the manner in which definitions of types capture the essential properties of mathematical objects. Visited topics include natural deduction, intuitionistic logic, the Curry-Howard correspondence, dependent types, and identity types.

Section 3: Homotopy Connections This section presumes a general familiarity with category theory and homotopy theory, and begins by introducing the informal homotopy-theoretic interpretation of type theory. With an informal analogy in place, we define Quillen model categories and explain their relation to classical set-theoretic homotopy theory. From here we introduce the interpretation of type theory into Quillen model categories, through which we make rigorous the informal interpretation. The section proceeds with an overview of applications to homotopy theory, and briefly visits Voevodsky's "univalence axiom." To avoid redundancy with the literature, our treatment avoids developing much of homotopy theory through type theory. However, enough groundwork is developed in our paper that a reader who understands the ideas this far would be prepared to consult the literature for this topic.

CHAPTER 2

SET THEORY

Introduction: The purpose of this section is to establish sufficient understanding of classical formal mathematics as to permit a compare-contrast exposition in the section explaining intuitionistic type theory. In total, we demonstrate how one can formalize mathematics through axiomatic set theory. To formalize the axioms of set theory themselves, we encounter first-order logic, the most common approach to codifying the logic and syntax of mathematics. To facilitate a rigorous discussion of logic, we show how a formal mathematical language can be defined, if only to demonstrate how such a formalism *could* work — we immediately discard the burdensome formality and resort to the usual notation. We explain the role of *inference rules* in predicate calculus, and demonstrate how the basic principles of mathematics can be derived formally. The section concludes by explaining how common mathematical notions can be represented in ZFC.

Readers interested in a more historical and philosophical analysis of set theory may find an excellent resource in [Gra08]. Let us stress now that homotopy type theory need not be studied to the exclusion of everyday set theory.[1]

Set theory is the branch of mathematics within which one is concerned with aggregrates of objects, the former known as *sets* and the latter as a set's *elements*. The objects of a set can be any sufficiently well-defined *things*, but mathematicians generally think of sets as being collections of numbers, points in space, or other mathematical entities. Set theory can be — and very commonly is — taken as the

[1]For instance, [Uni13] states on p. 8 that "... any sort of mathematics which can be represented in an ETCS-like theory (which, experience suggests, is essentially all of mathematics) can equally well be represented in univalent foundations."

foundation for mathematics, which is to say that (almost) every idea in mathematics can be couched in the language of sets.

Even in its capacity as a foundational system, set theory is typically practiced informally. That is, sets are mostly assumed to be an intuitive idea, in fact one of the most primitive and fundamental notions in mathematics. Rarely does the practicing mathematician make formal appeal to the exact specification of what rules sets follow, since rather than being studied for their own sake, sets are used primarily as an aid in communication; mathematical arguments are usually expressed in natural language augmented by set-theoretic symbolism, appealing to an implicit universal understanding of basic logical principles. A combination of experience and training, rather than a cumbersome list of formal rules, is what guides mathematicians in formulating definitions and proofs in a set-theoretic language.

When desirable, though, set theory can also be studied formally, as a specific *formal system* which entirely codifies all of the permitted constructions and logical deductions within the universe of sets. History has shown that studying mathematics which is directly based on such formalities would be an unwarranted burden, at least for most mathematicians. For our purposes, though, we will need slightly more formality to make precise the vague idea that "mathematics is based on set theory." Actually, let us suggest a slightly broader idea which will become more developed throughout this paper: Most ideas in mathematics *can* be formulated in terms of the abstract hierarchies considered by set theory, but mathematics and set theory need not be taken as synonymous and alternate approaches may shed additional insight into mathematics in general.

Remark. *Throughout this paper we use the phrase "set theory" in a broad manner, referring to the topic of collections and their elements but not necessarily to a particular formalization of it. When the distinction is particularly significant, we could use the phrase "naïve set theory" to mean everyday informal set theory, and a something like "formal set theory" to mean some specific formalization, usually ZFC, wherein one explicitly writes down the properties assumed of sets.*

2.1 Material sets and ZFC

When speaking of axiomatic set theory, it makes sense to speak of *this* or *that* set theory, and different theories may vary significantly in their approach. Examples of set theories include ZFC, NBG, SEAR, and ETCS. Note that we are not talking about different *models* of some particular set theory — i.e. different mathematical structures which, under an appropriate interpretation, satisfy the axioms of a particular system. We are instead talking about entirely different definitions of "set theory," in other words different axiomatic systems that have at various times been offered as ways to formalize the study of collections of objects.

We can roughly partition set theories into two sorts — material theories versus structural ones. The former class includes ZFC and NBG and is far more commonly studied than structural set theories, the latter including versions of set theory which are formulated in terms of category theory (such as ETCS). The classical (material) approach is to consider a single universe of sets whose elements are themselves other sets, and then define all other mathematical objects by picking out a set whose elements somehow "contain" information about how the object behaves. The structural approach typically considers a universe of somewhat primitive objects called "sets," where a particular set is determined by its external relations to other sets (rather than by its elements). It is the classical material approach which we scrutinize in this section, so we shall think of the phrase "set theory" as having this connotation. We shall not be concerned with structural set theories.

As explained, the "setting" of a material set theory, in other words the collection of *things* being considered by the theory, is some ambient universe[2] V of all sets (which, by Russell's paradox, will not itself be a set but a proper class). In the present context, the word "set" is not referring to all conceivable collections of objects, but specifically the *pure sets*. These are the sets whose elements are

[2]The word "universe" is being taken as a primitive notion here. It is a collection of things, making it what logicians call a "class." In particular, it is the class of all sets, where a "set" by definition is an element of V. But V is not an element of itself, as mandated by the axiom of regularity. Suffice it to say that a concerned reader need not worry about any apparent circularity.

themselves sets, in fact themselves pure sets. In the context of formal (material) set theory, it is understood that "set" is being used in this sense. We think of V as the collection of all (pure) sets and no other objects whatsoever but sets. This universe of sets comes equipped with a binary relation $\in$ which, *a priori*, may or may not hold between any two sets in V. This relation is, of course, the usual set-theoretic membership relation. Now, the ideas of V and $\in$ themselves aren't quite well-defined by the given description, meaning we will need some way of characterizing exactly what sorts of properties they have. First let us examine what sorts of properties we want them to have when we define them formally.

The relation $\in$ is global, meaning we can ask whether any two sets in V stand in such relation to one another. The particular focus on the globally-applicable membership relation is the distinguishing property of a material set theory. In such a theory, any set S is defined entirely by the other sets T for which

$$T \in S$$

holds. Besides what we can say describe in terms of predicate logic (examined below) and this relation, sets do not have any properties, although it turns out we can express quite a lot with just these notions. To be clear: any set is determined just by its elements, these elements being sets in their own right. Here we run up against an important facet of formal set theory, namely that the only objects we consider are sets. The reader should thus realize as a corollary that, in formal set theory, any mathematical objects *whatsoever* are subject to set-theoretic operations: intersection, union, taking the powerset, etc. How to derive the rest of mathematics in light of this seemingly limiting restriction to pure sets will become more clear when we look at particular examples of pure sets in the next section.

Before we can look at pure sets, we have to make our system more precise. The given description of V and $\in$ is a good enough starting point for our intuition, but the exact nature of V and $\in$, such as which pure sets are assumed to exist inside V and the law which states that sets are determined only by their elements, needs to

be codified by *axioms*. Here is usually where different material set theories diverge. For instance, we might want to impose an axiom asserting the existence of a set with no elements, like so[3]

$$\exists x\,(\forall y\,\neg[y \in x]).$$

It turns out, however, that we are not yet in a position even to write down the axioms of any particular theory, since even the notion of an axiom does not live in a vacuum. A material set theory like ZFC is *not free-standing*, but rests on top of another system called first-order logic (FOL). Usually in mathematics, we could withhold a description of first-order logic, since it is merely the formalization which permits us to impose axioms describing whatever system we are interested in studying. However, homotopy type theory is exotic and does not use first-order logic, so to appreciate this difference it makes sense to say a few words about what first-order logic is.

To say that ZFC rests on first-order logic is to say that the latter is logically antecedent, the former being defined as a particular application of it. Indeed, FOL is a system in its own right, one designed to codify the principles of logical reasoning in classical mathematics. In the realm of formalized (classical) mathematics, first-order logic is our most anterior idea, a sort of "initial element" in the poset of formalized mathematics arranged by logical precedence. By itself the system would not usually be of interest except perhaps to logicians, since it plays a supporting role to other ideas: individual mathematical theories like set theory, group theory, and number theory, can be formally defined and studied using the machinery provided by it. In fact, the very syntax we use to write down axioms, and the notion of a valid conclusion derived from axioms, is captured by this system.

[3]In ZFC, this fact is actually implied by other axioms, so we don't need to make this an explicit axiom.

2.2 First-order logic

First-order logic formally secures two key notions which form the basis of logical reasoning: propositions and proofs. Classical mathematics revolves around these concepts, so it makes sense that any formal foundation should give precise characterizations of both. Here, the word "proposition" indicates a statement which is either true or false, and is not synonymous to "theorem" as it is elsewhere used (theorems being provable propositions). The goal of first-order logic is to provide an appropriate rigorous framework for representing propositions symbolically and defining how they stand in relation to one another. With this system in place, we can describe mathematical theories by their axioms and derive logical consequences of the axioms in an unambiguous manner.

Generally, phrases like "first-order logic," "first-order predicate calculus," "predicate logic," etc., all refer to the same kind of system, namely the one we describe here. We will use these phrases somewhat interchangeably, for linguistic variety. Different authors' presentations may vary in their exact formalisms, but the basic ideas of predicate logic are standard and constitute an extension of propositional logic ("propositional calculus," "zeroth-order logic," etc.). Specifically, in addition to the usual laws for propositions, predicate logic contains machinery to define the notions of predicates and quantification (leading to the alternative phrase "quantification theory"). A predicate is a statement not dissimilar to a proposition but involving variables, an example being the expression $3 + x = 7$ where x is a variable ranging over the integers. Such a statement has no truth value, unlike a proposition — it gives rise to a proposition when we supply an specific object to be substituted for each of its variables. Therefore predicates may be thought of as functions mapping objects to propositions, permitting the terminology of predicates as "propositional functions." Predicates may also be thought of as families of related propositions indexed over some class of objects.

A binary relation like $\in$ (the standard membership relation in set theory) is a predicate which accepts two arguments, since the expression $x \in y$ is either true or false as the variables x and y range over the universe V of all sets. A predicate can also give rise to a proposition by *quantifying* each of its variables v with a phrase like "for all v" or "there exists a v." For instance, $\forall x\ 3 + x = 7$ is a proposition, as is $\exists x\ \forall y\ y \in x$, even if both are false under their usual interpretation.[4] The variable used in the quantifying phrase is considered to be *bound* by the quantifier — intuitively, whenever the variable appears after the quantifying phrase, it is thought of as a fixed general object of the appropriate sort and not a variable.

For obvious reasons, we cannot use a formal system like ZFC to define first-order logic as we might do with other mathematical ideas. The formal definitions of axiomatic set theories come as a particular application of predicate logic, so the only "system" we have access to in defining the latter is our natural language. Actually, we can use words like "set" to describe logic insofar as a set is an everyday idea, although for clarity we will use the word "collection." Inevitably, we must take certain ideas to be understood before we begin to develop first-order logic, including the notions of collections and natural numbers. Naturally, though, we restrict our use of such ideas to instances where they are practically unambiguous.

Remark. *There are at least two ways to interpret our exposition of logic and axiomatic set theory. One view is that we assume only a working understanding of the English language and then lay down a series of definitions which accumulate to form the "True Foundation of Mathematics," so to speak. Under this interpretation, the everyday mathematician is one who disregards the troublesome formalities of a rigorous foundation and develops informal but rigorous arguments which, it seems plausible, could be formulated entirely into the foundation.*

[4] A word on notation: While mathematicians might write something like "$\exists x$ such that $P(x)$," where $P(x)$ is some predicate involving the variable x, logicians tend to omit the joining phrase "such that" and simply write $\exists x\ P(x)$

Another view is that our definitions and formalisms are actually studied in the same ambient set theory used to study any other branch of mathematics. Under this interpretation, we are not laying down the "true" mathematical bedrock but simply developing a general framework for investigating topics in logic. That is, we are not working to secure a more philosophically palatable foundation but simply applying our usual techniques to logic, as opposed to some other field. Under this interpretation, our studies exist at the same "level" as abstract algebra or number theory, for example.

Each view perhaps shines more prominently than the other in certain places throughout our exposition. If so, this does not reflect an entirely conscious decision. The two positions need not be considered wholly exclusive, depending on the purpose of the investigation.

The development of first-order logic proceeds naturally in a series of steps. First we develop a precise syntax for representing mathematics symbolically, which we call a formal language. For this task we first introduce the alphabet of the language, this being the collection of symbols at our disposal. Then we provide syntactical rules which specify which string of symbols we consider to be well-formed formulas, these being the ones which can be interpreted as making some kind of statement, although we keep in mind that we cannot use the intended interpretation of the symbols in defining such a notion. After this stage we are able to define the rules of the system, and we use this idea to define the notion of a (formal) proof, which concludes our development of first-order logic. From here, we are able to define a theory such as ZFC simply by listing its axioms in the language developed here.

We turn our attention now to formulating an alphabet for first-order logic, which for clarity we develop in stages. Our goal is not actually to use the formal language developed, but to establish that such a device can be defined. This will provide some rigor to the subsequent discussion regarding a general syntax for mathematics, which itself is a necessary step in explaining the inference rules of classical logic. All of

these notions are developed so as to provide a sufficient understanding of ZFC to contrast type theory against that system.

The logical symbols

The first symbols to include in our alphabet are naturally the common logical symbols like

$$\textbf{Logical Symbols}\ \forall, \exists, \neg, \implies, \iff, \wedge, \vee, =$$

These symbols, as the reader may well know, are commonly used to represent the ideas

For all, there exists, not, implies, if and only if, and, or, equals,

respectively, although it is worth emphasizing that we are not formally giving these symbols any interpretation whatsoever.

Variables

Commonly, mathematicians use symbols like

$$x, y, z, \ldots n, m, j, k, \ldots a, b, c \ldots$$

to stand in for general objects without a specific "value." For example, we use variables in such phrases as "Let n be a natural number," or "For all group elements x." Our formal language reserves the following symbols for this purpose:

$$\textbf{Variables} \qquad x_0, x_1, x_2 \ldots$$

The subscripts are simply notation to differentiate between different symbols. When we later build theories on top of predicate logic, these symbols will be used to stand in for general sets, group elements, numbers, etc., depending on the theory. These are not the symbols we attach to specific objects like the empty set $\emptyset$ or the definite number 5.

Constants and functions

Mathematicians tend to reserve certain symbols which, in a given context, represent some "function" (of however many arguments) or some constant object. Recurring examples include

$$f, m, g, h, \mathcal{P}, \cup, \cap, +, -, \times, /, \emptyset, 0, 1, 2, \ldots$$

These are the symbols which might represent taking a powerset (or union) of sets, or an algebra's operations, or just everyday functions on sets. By "function," we really mean any piece of notation which, when supplied with the right number of objects, represents some other object. In particular, functions mapping sets to sets are just one kind of logical "function." For example, writing $\mathcal{P}(X)$ to indicate the powerset of a set X is also being called a function despite the fact that it takes arguments from the entire set-theory universe V, which is not a set. For the purpose of representing functions, our formal language employs the symbols

Functions $\quad f_0^0, f_1^0, f_2^0, \ldots f_0^1, f_1^1, f_2^1, \ldots f_0^2, f_1^2, f_2^2, \ldots$

The subscripts again are simply markers for distinguishing between symbols. The superscript, which can also be considered notation, keeps track of the *valence* of a function, the (natural) number of arguments that it takes syntactically. The valence will be formally reflected in the rules for well-formed formulas, where we specify that a string like $f_0^2(x, y, z)$ is not well-formed because f_0^2 should take precisely two arguments (for instance). The reader should keep in mind, still, that we are only working with symbols at this stage, and resist the inclination to read a raw string like $f_0^2(x, y)$ as more than an arrangement of letters and punctuation (and not, for instance, as the specific value returned by the function). Notice that constants, such as 0 in an abelian group or $\emptyset$ in set theory, are effectively functions which take no arguments (that is, functions of valence 0).

Propositional variables and predicates

We also include symbols to represent relations like $\in$ in set theory. One could call these "predicate" symbols because of their role in forming predicates (e.g. $x \in y$). Another suggestive term is to call them "relation" symbols, although this brings up some additional nuance — here, a (logical) relation is a function-like notion which accepts a certain number of arguments and "returns" a value of either true or false. Therefore it is different from the notion of a (set) relation in set theory, a subset of a Cartesian product. The connection is that a logical relation accepting n variables from a class $\mathcal{C}$ of objects can (informally) be identified with the subclass of $\mathcal{C}^n$ consisting of those n-tuples for which the logical relation holds true. For logical relations, we use the symbols

Predicates (Relations) $\quad R_0^0, R_1^0, R_2^0, \ldots R_0^1, R_1^1, R_2^1, \ldots R_0^2, R_1^2, R_2^2, \ldots$

As with function symbols, we use superscripts to denote valence, i.e. the number of arguments the relation takes. For instance, $\in$ is a relation of valence two on the class V, accepting two sets x and y and returning the proposition $x \in y$. Logical relations of valence 0 are simply propositional variables — they are either true or false and contain no other information.

Remark. *Note that we often write xRy in place of $R(x, y)$ for relation R of valence two — formally speaking, we could say that an expression like $x \in y$ is notation for $\in (x, y)$.*

Punctuation

Finally, we include whatever punctuation we need

( ) ','

to make things readable.

The formation rules

Thus far we have not formally specified anything about the symbols, since we have only listed them and described their intended usage. At this stage, all of the symbols are more-or-less on equal ground. We have to specify rules about how we can string these symbols together, through which we begin to recover their usual meaning.

First, we define *terms*. Effectively, these are the "objects" being considered by the logic. The rules are simple and work inductively:

- Individual variables are terms
- $f(t_1, t_2, \ldots, t_n)$ is a term if f is a function symbol with valence n and each t_i is a term
- No other expressions are terms

Notice that terms need not have a specific "value," since we have said that variables are terms. Constant values are also terms, since we have identified constants with functions of valence zero.

We wish to designate certain strings of symbols as (well-formed) formulas. Formulas are the strings which, loosely speaking, could be interpreted as making *some* kind of statement (possibly involving variables) — not necessarily one usually interpreted as true. As with terms, the rules are simple and work inductively.

- $P(t_1, t_2, \ldots, t_n)$ is a formula if P is a predicate symbol with valence n and each t_i is a term (Note that the terms may be variables or constants)
- If ϕ and ψ are formulas, so are these strings: $\phi \implies \psi$, $\phi \vee \psi$, $\phi \,\&\, \psi$
- If ϕ is a formula, so is $\neg\phi$
- If ϕ is a formula and x is a variable symbol, $\forall x\, \phi$ is a formula, as is $\exists x\, \phi$
- No other expressions are formulas.

A string like

$$\forall\exists\neg x \implies \iff \forall\forall,$$

as expected, is not a well-formed formula. Moreover the syntactical nature of the logic means we can say that this string is not well-formed simply by examining the arrangement of the symbols, without having to resort to some vague explanation like "this string obviously cannot be interpreted as a statement." A string like

$$P(x) \implies \neg\exists y\,(y \in x)$$

(for a predicate P of valence 1 and variables x,y) is an example of a formula.

Now, the well-formed formulas are not all of the same sort. Under some interpretation of the symbols, certain formulas can represent propositions — statements with a fixed truth value. Others involve free[5] variables and can only represent predicates. The syntactical rules make this notion precise as well, such that we can examine any arrangement of symbols and mechanically determine whether a formula involves free variables (such as $P(x, y)$ and $x = y$ for some predicate P and variables x and y) or only bound variables (like $\forall x\,(x = x)$ and $\exists y\,\exists z\,f(y, z) = 0$). The ones which contain no free variables are called *sentences*, and these are the ones which can be interpreted as having a specific truth value. Sentences, then, represent propositions, at least in some context where we are giving the symbols a semantic interpretation. We shall often blur the distinction between a proposition and the string of symbols which represents the proposition, which is customary and avoids rather unwieldy phrasing.

As this stage we can now say unambiguously:

- Which strings of symbols can represent (constant or variable) objects (like x, $\emptyset$, $f(x)$, $x + y$) and which cannot (like $\forall$, $y = 3$, $\in$). These objects are the *terms*.
- Which strings of symbols can represent *some* kind of statement ($x \in \emptyset$) versus those too ill-formed to be interpreted in any way ($x \in \neg$)
- Of the sensible strings, which could be interpreted as having a fixed truth value and which cannot (because they have free variables)

[5] "Free" as opposed to "bound" by a quantifier $\forall$ or $\exists$

Thus we have a working formal language for speaking about mathematics. Still, we have not specified any rules that would allow us to interpret individual symbols in a normal manner (e.g. $\emptyset$ is only a constant symbol yet, not "the empty set"). Moreover, even though we know which formulas can be interpreted as having a truth value, we could not say what that truth value is. What we are really missing is a notion of *proof*, which will later permit a notion of *axioms*.

Inference

We still do not know what a proof is, so we turn our attention to this subject now. Still at this stage we do not appeal to any interpretation of the symbols, just as all previous rules have only referred to the arrangement of the symbols. The motto here is "logic is independent of content," so it makes sense to define the rules of logic without knowing precisely how the symbols are going to be interpreted in a particular context. For instance, it is reasonable to say that whenever we know "ϕ implies ψ" is true, and that ϕ is true, we may infer that ψ is true, without knowing what ϕ or ψ are representing in context. Such are the kinds of rules we are now specifying — the *inference* rules.

Mathematical practice consists of passing *judgments*. A judgment is an belief *about* some theory, rather than a statement expressed *within* a theory (which would be a proposition). It is thus an "external" belief living in the realm of meta-theory, rather than a formulation within the theory being examined. For instance, the conclusion that a certain string of symbols is a sentence (a formula with no free variables), is a kind of judgment. In contrast, the proposition being expressed by the sentence (maybe an assertion about two topological spaces) is not a judgment, even if we think the proposition is true, since it is a statement formulated in the language of the system. We could, however, form the judgment that the proposition is true, specifically that it has a *proof*. This belief would be a judgment because it is a belief based on the rules of the theory, and not a proposition formulated symbolically in accordance with those rules. Our task right now is to explain those rules which permit such a judgment.

To codify the notion of proof, we can give a collection of *inference rules* which specify which sorts of propositions can be used to establish the truth of which others, in other words which inferences are valid. There is no canonical way to present the inference rules of first order logic, and we avoid developing them here. The important part is that these rules should have the expected behavior with respect to the logical symbols given above. For instance, so far we can say that $\phi \implies \psi$ is a formula, but nothing specified so far would justify reading the symbol $\implies$ as "implies." We could justify this reading through an inference rule stating that the formula $\phi \,\&\, (\phi \implies \psi)$ allows for the inference of the formula ψ. Similarly, we would want the inference rules to specify that the formula $\phi \,\&\, \psi$ is sufficient to infer both ϕ and ψ. Provided with a collection of inference rules, a formal proof would be any list of consecutive formulas, each of whose truth is inferable from the truth of the preceding ones in accordance with the inference rules. We judge that a proposition is provable when can find some formal proof which culminates in that proposition.

For comparison:

- In set theory, the notion that the empty set has no elements is not a judgment, but a proposition. It is a statement expressed in the language of first-order logic, perhaps written $\neg\exists x\,(x \in \emptyset)$.
- That we can *prove* the empty set has no elements is a judgment. We can appeal to the inference rules of ZFC to produce a formal proof culminating in the formula $\neg\exists x\,(x \in \emptyset)$.

The inference rules must be defined syntactically, i.e. just in terms of symbolic formulas and not in terms of their underlying interpretations. Still, the rules serve just to codify the sort of logical reasoning that mathematicians are used to, and therefore permit a customary interpretation of the language. Although we do not give an exact presentation of the inference rules, suffice it to say that at this stage, we have a notion of "proof" which behaves in precisely the way that mathematicians would want it to. First-order logic is now completely defined.

Remark. *At this stage, we can resort to the usual notation of mathematics if we keep in mind how to carry out a translation into the formal language. For instance, we can use the symbol* $\in$*, keeping in mind that such a symbol can be represented with a relation symbol like* R_0^2*. We can also introduce a constant* $\emptyset$*, perhaps with the formal symbol* f_0^0*.*

However, we cannot justify a customary interpretation of any new symbols we introduce. For instance, we cannot disprove the (hopefully false) proposition $\forall x\,(x \in \emptyset)$*, since the relation* $\in$ *has no special properties except that it is a binary predicate, while* $\emptyset$ *is just a constant symbol. Rectifying this situation is our next topic.*

Axioms

We now have a working system of (first-order) logic in place, which boils down to knowing what propositions and proofs are. At this stage, we can formally express laws of general predicate logic and then prove them as theorems. For purposes of mathematics though, predicate logic is not enough. While the system is sufficient for examining truths which are fundamental to logic, there are certain truths that we wish to take for granted — we are not interested in deriving general logical theorems as much as deriving *consequences* of propositions already taken to be true.

What we are missing is *axioms* — judgments we take for granted, rather than deriving from the rules of pure logic. To get some theory (like set theory) "off the ground," we supply a list of propositions which we take to be true at the outset, so that we subject *these* propositions to the rules of inference. Moreover, the axioms are what allows for an interpretation of the symbols, since we will use the axioms to specify additional properties of symbols like $\in$ besides what can be said about binary relations in general. First-order logic, by itself, has no axioms — the system is only concerned with general logic, not with particular "things" like sets. With predicate logic in place, we can impose the axioms which characterize some particular theory *on top of* the logic. For demonstration, let us examine how we could formalize group theory directly in terms of first-order logic.

First, we specify that the structure called "group" comes with two functions — group multiplication m and the inverse operation i — and a constant 1. These specifications are made using the notion of a *signature*, but that detail does not concern us. Multiplication m, of course, must be a function of two arguments, whereas i takes just one. Then we impose the following recognizable axioms using the language of first-order logic, our intended interpretation being that all objects refer to elements of the group (so variables represent arbitrary group elements):

Remark. *For readability, we write $i(x)$ as simply x^{-1}. We also string together multiple equalities with one formula, although a completely formal presentation might separate each equality as a different axiom.*

$$\forall x\, m(x, 1) = m(1, x) = x$$

$$\forall x\, m(x, x^{-1}) = m(x^{-1}, x) = 1$$

$$\forall x\, \forall y\, \forall z\, m(x, m(y, z)) = m(m(x, y), z)$$

The resulting system of first-order logic and these additional axioms[6] is known as "first-order group theory," which describes group theory by itself, *outside* of set theory. The behavior of the group operation m is specified by the axioms, which state that it has certain behavior with respect to the other function (taking the inverse) and with the special constant 1, and moreover it behaves associatively. It is because of the axioms that we can think of the expressions 1, m, and x^{-1}, in the usual manner. Any structure which satisfies the axioms can be called a "group," where a "structure" is any package containing the following things: some collection of underlying objects, a designated special object, a binary operation, and a unary operation.

The rules of first-order logic now allow us to derive further propositions from the axioms. If we agree with the rules embodied by first-order logic, then whatever

[6] An axiom specifying closure is not necessary here.

can be proved from the axioms is a property common to all structures satisfying the axioms. We can interpret these propositions as true facts common to all such structures *if we find such things*. Nothing about first-order group theory asserts that there are any such structures to be found — it merely allows us to formally derive properties consequent to the group axioms. It may be useful to think of the axioms as a hypothetical context. *If* there is some collection of things that is associated to two functions of the correct valence, where one object is marked as a "special" one, and *if* such a system satisfies the axioms, *then* it also satisfies whatever can be proved from the axioms according to the inference rules. Of course, not *all* of such a system's properties have to be consequences of the axioms, but all consequences are properties of all such systems. If we are given a particular, actually existing group, such as the integers $\mathbb{Z}$ under addition, then the group is said to be a *model* of group theory — some kind of structure that satisfies the axioms. The obvious question then is, "Whence do we obtain the object $\mathbb{Z}$?" The answer lies in set theory.

Remark. *Here is one area where we begin to encounter some issues which are not pragmatically challenging but may be awkward foundationally. If we want to assume only a working understanding of natural language, then a "model" for group theory is taken as an intuitive idea: a collection of sufficiently well-defined things, with one thing designated as special, and two sufficiently well-defined methods (of the correct valence) for associating things to things.*

The more practical viewpoint for many purposes is to assume the same background set theory which permeates most of mathematics, so that a model of group theory is a set equipped with a special element and two functions. Then our following treatment of ZFC is not seeking to define set theory at the most foundational level, but to investigate some of its properties and limitations (under a particular axiomatization) the same way we study group theory in everyday practice.

The foundational significance of a formal set theory like ZFC is that the objects it examines — the so-called pure sets — can be used to represent just about anything in

mathematics. For instance, in everyday set theory we often take "the set of integers" as a somewhat primitive idea whose elements we simply understand. Through ZFC, we can appeal to the axioms to formally pick out a particular set $\mathbb{Z}$, whose elements are exactly specified as sets themselves, after agreeing that $\mathbb{Z}$ has the right properties to consider it a fair representation of the integers. We might even use first-order logic to axiomatize the integers themselves, and then check that the set $\mathbb{Z}$ is a model for integers.

Let us turn now to the formalization of set theory, specifically through ZFC, which is recognized as the canonical formulation of set theory. ZFC, like first-order group theory, is the combined system of first-order logic and some additional axioms, namely the axioms of **Z**ermelo-**F**raenkel and the axiom of **C**hoice. These axioms are intended to characterize the nature of sets, especially the behavior of the $\in$ predicate. A complete list is included in the first appendix, but we offer two here, for demonstration. The intended interpretation is now that all objects (terms) are sets, so that variables represent arbitrary sets.

$$\textbf{Extensionality} \quad \forall x \, \forall y \, x = y \iff \forall z \, (z \in x \iff z \in y)$$

$$\textbf{Pairing} \quad \forall x \, \forall y \, \exists z \, [x \in z \wedge y \in z]$$

The axiom of extensionality specifies that two sets are equal precisely if they have all of their elements in common. The pairing axiom says that for any two objects (sets) that exist, there is also a set that contains both of them as elements. Again, formally speaking, we are not initially attaching any meaning to the symbol $\in$ (nor any others, technically), but instead *giving* this symbol meaning by our choice of axioms, or at least justifying its usual interpretation.

One subtlety here is that the axioms of ZFC still are not declarations of faith any more than the axioms of group theory. The axioms of group theory only define what it means for something to be a group, and the axioms of ZFC only define what it means to be a model of ZFC. Set theory is somewhat special, though, since

we often imagine one "true" universe V of sets whose properties are described by the axioms. Actually, we can consider many non-identical structures which, when the axioms are suitably interpreted, are models of ZFC. As with group theory, the models are themselves sets, although necessarily these are not sets which can be formally defined by appeal to the axioms of ZFC. [7] Examining these different structures is the method by which we prove that certain questions we could ask about sets, such as the continuum hypothesis, are not "decided" by the axioms of ZFC. This is not different to the situation wherein the axioms of group theory do not "decide" whether a group must be abelian, and we can prove this independence by providing some groups that are abelian and some that are not. Presently this topic does not concern us, but it should be appreciated.

With first-order logic defined and the axioms of set theory imposed, we now have a working set theory which behaves in the expected manner, notwithstanding some interesting behavior which is not usually relevant to everyday practice. For instance, there is an axiom which justifies our use of set-builder notation, and another one which states that powersets exist. Another one says that we can take unions, while still another one states that there is at least one infinite set. Therefore we can now reason with our usual set-theoretic intuition. For convenience we can discard the formalities of the formal language and instead use our usual notation, which we will do. There is just one major caveat: the only objects we formally have access to are pure sets. The section below will examine how various mathematical notions can be represented as pure sets, allowing us to recover everyday mathematics from just set theory.

It may be useful to think of axiomatic set theory as something like an algebraic theory. The axioms are analogous to generators, representing a sort of initial state

[7] Essentially, Gödel's Incompleteness Theorems imply that if ZFC can prove the existence of a model of ZFC, then ZFC is inconsistent. Models of ZFC are generally studied under the additional assumption that ZFC is consistent, which by another theorem of Gödel implies there is a set in the universe which is itself a model of set theory. A model would be a set equipped with a binary relation satisfying the axioms assumed of $\in$.

to our collection of true propositions. The inference rules, almost like algebraic operations, are applied to the generators to produce additional truths. The resulting totality of provable propositions is true of any structure satisfying the axioms.

Before we proceed with examples of regular mathematics being performed in ZFC, the reader should reflect on this fact, which is fundamental to later appreciating type theory: propositions and proofs are not objects within set theory or first-order logic. The former are (represented by) certain strings of symbols and the latter are consecutive strings of propositions. First-order logic is the system for writing these ideas down symbolically, but the objects of first order logic are used to represent mathematical entities like numbers, sets, and spaces — not propositions and proofs themselves. Nor are these ideas part of set theory, since neither a proposition nor a proof is a set inside V. Therefore proofs and propositions lie in the domain of mathematical practice, but are not objects described within mathematics itself. We will see later that type theory actually subsumes logic, bringing propositions and their proofs into mathematics as first-class citizens subject to the same kinds of operations as other objects.

2.3 Recovering mathematics

In material set theories like ZFC, all common objects have to be modeled in terms of sets, rather than existing as pre-defined objects which are subject to being collected into sets (as would be the case with naïve set theory). For instance, let us examine the set-theoretic construction of the natural numbers.

The standard formalization of the natural numbers is according to this scheme, following von Neumann:

$$
\begin{aligned}
0 &= \emptyset = \{\} \\
1 &= \{0\} = \{\{\}\} \\
2 &= \{1, 0\} = \{\{\{\}\}, \{\}\} \\
3 &= \{2, 1, 0\} = \dots \\
&\dots
\end{aligned}
$$

In general, we encode the successor (i.e. $n + 1$) of a natural number n as

$$\text{succ}(n) = n \cup \{n\} = \{0, 1, 2, \dots, n\}.$$

From here, we define $\mathbb{N}$ as the collection of such objects — the smallest set containing the element 0, subject to the rule that $n \in \mathbb{N} \implies \text{succ}(n) \in \mathbb{N}$.

The given representation, for better or for worse, is not unique. We can very well define the natural numbers in this manner:

$$
\begin{aligned}
0' &= \emptyset = \{\} \\
1' &= \{0'\} = \{\{\}\} \\
2' &= \{1'\} = \{\{\{\}\}\} \\
3' &= \{2'\} \dots \\
&\dots
\end{aligned}
$$

where we define the successor $\text{succ}(n)$ to the number n as simply the set $\{n\}$. Then we define the set $\mathbb{N}'$ as the collection of all such numbers. Either of these definitions, $\mathbb{N}$ or $\mathbb{N}'$, can be taken as the formal definition of the natural numbers, and although they are set-theoretically not equal, the two sets are not functionally different to the

mathematician. For instance, both definitions are convenient for describing proofs by induction.

Let us turn to ordered pairs. The ordered pair (a, b) of two objects (themselves necessarily sets) a and b, is clearly not the same thing as $\{a, b\}$, since sets are not ordered.[8] We seek a way to "encode" the pair into some unordered set, perhaps defining $(a, b)_{\text{simple}} = \{a, \{a, b\}\}$ (where the subscript keeps track of which definition of (a, b) we are using). Another common definition is $(a, b)_{\text{less simple}} = \{\{a\}, \{\{a, b\}\}\}$. Both of these definitions are sufficient, insofar as we can prove the defining property of ordered pairs, namely that

$$(a, b) = (a', b') \iff a = a' \text{ and } b = b'.$$

With these definitions in place, we call the reader's attention to a peculiar fact. The axiom of extensionality ("sets are defined by their elements") can be seen to imply that $(0, 0)_{\text{simple}}$ is equal to the number 2 (in $\mathbb{N}$) under our definitions (which are in fact both common definitions). Of course, a mathematician would not normally ask if an ordered pair is equal to a natural number, because in practice these are different sorts of objects which we do not consider as being subject to equality, except maybe to say they are un-equal because they are different sorts of objects. Thus we have begun to encounter the strangeness of using pure sets to represent mathematics.[9]

Having formally defined ordered pairs and natural numbers, one is in a position to define the integers, often as equivalence classes of ordered pairs of natural numbers (perhaps with the equivalence class of (a, b) defined so as to represent the integer $b - a$, for example). From here, rational numbers are conventionally defined as equivalence classes of ordered pairs of integers, so as to reflect the usual laws about

[8] Notice that the ability to form pairs like $\{a, b\}$ is a consequence of the pairing axiom of ZFC. Actually, this axiom only guarantees a set containing $\{a, b\}$ as subset, but the axiom of (restricted) comprehension then permits forming the exact two-element set.

[9] Formalizing the notion of "these are not the same kind of object" is a motivation for using type theory in computer science and mathematics.

rational numbers. Then real numbers can be encoded in a varient of ways, often as equivalence classes of Cauchy sequences of rational numbers.

At this point we have seen (if only in passing) how to define $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ using pure sets, as well as ordered pairs. We now look at functions, and hereafter the reader may find it plausible that most other mathematical notions are definable with pure sets as well. Given two sets X and Y, we can define a function $f : X \to Y$ to be any set $f \subset X \times Y$, where $X \times Y = \{(x, y) \,|\, x \in X,\, y \in Y\}$, satisfying

$$\forall\, x \in X\ \exists\, y \in Y \text{ such that } (x, y) \in f$$

with the property that moreover this y is uniquely determined by a choice of x. We use the notation $f(x)$ to represent this y.

Interestingly, a material set theory such as ZFC allows for set-theoretic comparison of any two objects, even though such a comparison is not always meaningful or interesting. For instance, provided we have some formally established set-theoretic definition of the rational number 7.2, the space L^2 of square-integrable functions on $\mathbb{R}$, and the function $f : \mathbb{N} \to \mathbb{R}$ sending the natural number n to the real number $n\pi$, we may ask whether $7.2 \in (L^2 \cap \mathcal{P}(f))$, where $\mathcal{P}$ represents the taking of the powerset. If this seems incorrect, recall that all objects are subject to set-theoretic operations because all objects are sets. Of course, this question is unusual to ask, and is neither interesting to most mathematicians, nor unique with respect to the usual, informal definitions of the objects involved with respect to their functional properties.

Also notice that in order to define complicated objects, such as $\mathbb{R}$, we first define less complicated objects (like $\mathbb{Q}$, itself built out of $\mathbb{N}$), and these definitions accumulate hierarchically to form describe the final product.[10] This construction process leaves behind much residual structure which is not relevant mathematically but only set-theoretically. Here we see the motivation for calling ZFC a "material" set theory,

[10] If this is unclear, examine the definition of $\mathbb{R}$ an think about what its elements are, and what kind of elements *those* sets have.

since its objects are "made out of something." Indeed, sets have an elaborate but irrelevant internal structure resulting from the accumulation of previously defined sets.

The nuances of ZFC tend to conflict with mathematical intuition, especially the tendency to informally "identify" structures in mathematics which are set-theoretically distinct. For instance, groups which are isomorphic are identifiable insofar as group-theoretic properties of one can be translated into properties of the other. The powerset of a set X is commonly identified with the set 2^X of functions from X into 2, even though the sets have entirely different kinds of elements. Most common objects like the ring $\mathbb{Z}$ are often defined without even describing their elements in full detail, since it is not nature of the elements but the ring operations which are important. In general, everyday set theory is rather distant from its formalization.[11] Apparently, this removal is not too terrible for mathematics, since mathematicians are capable of systematically avoiding unnecessary formalities (which is emphatically *not* to imply that everyday mathematics is not rigorous.).

However, there is at least *some* disadvantage with the disconnect between everyday mathematics and its formalization. One could imagine that tools like proof verifiers, computer programs which can automatically check the correctness of a purported mathematical proof, would be valuable for mathematicians. Computer formalization of ZFC is possible, but not generally in a way that accommodates how mathematicians tend to think about sets — a verifier based on ZFC would require mathematicians to input formal set-theoretic proofs, or else require a very complicated engine which converts regular proofs into formal ones. Formalizing all proofs into ZFC would perhaps be as excruciating as programming computers with machine code, which would certainly be an unacceptable state of affairs in computer science. In part, the problem is that material set theory provides a very low level

[11] Avoiding these nuances of material set theory is the purpose of a structural set theory like ETCS "Structural," in this context, refers to the fact that objects are defined by their relevant structure, rather than by their material structure which is not functionally relevant

of abstraction for working with mathematical objects[12] – abstract objects such as $\mathbb{R}$ have to be exactly specified in concrete detail, and in such a situation, keeping track of the map between some informal idea and its formal representation is left to the mathematician. For instance, if we think of mathematics purely in terms of ZFC, the mathematician is responsible for keeping track of the following:

- In what ways $\mathbb{N}$ and $\mathbb{N}'$ represent the "same idea."
- Whether the set $\{0, \{0\}\}$ is functioning as $(0, 0)$ or 2 in context.
- Whether two entirely distinct sets may be equivalent in some interesting way
- How to define a real number as a set of sets (of sets...)
- That the question of whether $7.2 \in (L^2 \cap \mathcal{P}(f))$, however well-defined, is probably uninteresting as a mathematical proposition.

Proof verifiers do in fact exist and are increasingly used in mathematics and computer science. These systems do not use ZFC set theory, but rather type theory. We shall now see that type theory is also useful not only as a highly abstract system for expressing general mathematical notions, but can be used informally while still being readily formalized into a computer environment. Furthermore, type theory, similarly to category theory,[13] emphasizes the relevant structure of mathematical objects more so than set theory. The central thesis to be explained in this paper is that, remarkably, type theory also has strong connections to homotopy theory. Let us then turn our attention to the theory of types.

[12] In this case, abstraction away from the underlying and non-unique representation of some mathematical idea.

[13] This is no coincidence, since the two systems are closely related

CHAPTER 3

INTUITIONISM AND INTUITIONISTIC TYPE THEORY

Introduction: In this section we introduce the reader to a variant of type theory. Since this system was developed to formalize a variant of constructive mathematics, we visit that perspective as well in order to motivate the type-theoretic approach. We carefully highlight the initial subtleties in type theory which can make the system quite different from ZFC and everyday mathematics. After providing a tutorial on type theory in general, we describe how to formalize logic as an application of type theory, contrasting to the approach above where we build ZFC on top of formalized logic. The section concludes with an explanation of propositional equality, which leads us to the homotopy interpretation in the next section.

Arguably, most mathematicians informally practice a sort of type theory, albeit with the background assumption that such work could be formalized into a set-theoretic formal system like ZFC. Below, we explore type theory somewhat more casually than set theory, constrained by the scope of this paper, although type theory itself can be completely codified into a rigorous formal system. Because type theory is commonly employed by computer scientists studying such topics as *program correctness*, where informalities are unacceptable, type theory is often presented as completely formal system; it is a stated goal of the univalent foundations program to develop a more informal account of type theory which is sufficient for mathematical practice. However, one key advantage of type theory is that type-theoretic reasoning can be readily formalized (say, into a proof assistant), while work carried out in set theory can be challenging to formalize.

The phrase "type theory" could refer to any formal system from a class of related ones, as with "set theory." The idea was originally pioneered by Bertrand Russell (1872-1970) to circumnavigate the paradoxes of too naïve an approach to set theory. A "type" in its original sense is some class of objects which can be quantified over in a predicate, or in Martin-Löf's words, "The range of significance of a propositional function."[1] ZFC can be regarded as a degenerate type theory, since all objects are sets and, at the formal level, quantification ranges over all sets.[2] Today, type theories are primarily used in (theoretical) computer science as a device for reasoning about programming, where the word "type" is used in a slightly more concrete sense explained below. Type theories can be used to verify the logical correctness of computer programs or even to derive a correct program from a given specification that the program should meet. Papers which make use of type theory, more so than with set theory, tend to vary with respect to which formalization is meant. In this paper we are interested in a system successively developed by Swedish logician Per Martin-Löf ([ML84], [ML98]), and we shall use the phrase "type theory" to refer to this particular system. However, many aspects described below are common to type theory as a general paradigm.

Within type theory, each object is of a certain kind which we call its "type." Any given object is of only one kind, so an object's type is unique. As with first-order logic, our objects are called *terms*, and a new term is only ever introduced alongside an assertion of its type. Types, in turn, behave like the collection of all of their terms, which makes them somewhat like sets. Most generally, types are computational specifications — a new type is defined by specifying the general properties of the terms it contains. Ideally, we would like to have an algorithm which can validate or refute the assertion that some term is of a certain type, which is one respect in which some type theories are preferred to others. The nature of types and their terms will become more clear throughout the following.

[1] Quotation taken from [ML98]

[2] A statement such as "For all real numbers x, Φ" translates formally as "For all sets x, $x \in \mathbb{R} \implies \Phi$"

Martin-Löf's type theory was designed as a foundational system for intuitionistic mathematics — mathematics as seen through the philosophical framework associated with the 20^{th} century mathematician L.E.J. Brower. We give a cursory account of the intuitionistic perspective, familiarity with which is helpful to understand the intuitionistic type theory of Martin-Löf.

3.1 Brouwer's Intuitionism

L.E.J. Brouwer (1881-1966) is well-known for his work in topology and philosophy of mathematics, in the latter field as the founder of the intuitionistic school of thought. Intuitionism, a particular kind of mathematical constructivism, stems from the belief that mathematics is a process of mentally *computing* abstract objects, rather than a process whereby one discovers absolute truths about "real" objects living in some Platonic realm. Constructive mathematics in general tends to limit itself to operations which can have some kind of computational significance. For instance, the axiom of choice in ZFC, which can be found in the first appendix, is not embraced by constructivists because it postulates the existence of a certain set (a choice function) but does not *compute* such an object. In contrast, we shall later see that a function is defined in type theory by (effectively) providing a program to compute it. But unlike some systems used in constructive mathematics, our type theory is not simply a modification of set theory, perhaps replacing the axioms with some more modest ones. Instead, type theory is a scheme for classifying different kinds of computations.[3] The most significant consequence of this emphasis on computation is that we do not begin building up our mathematics on top of logic as we do with set theory. Instead, we commit to "Brouwer's Dictum," to borrow a

[3]On an historical note, the reader should appreciate that constructive mathematics predates the proliferation of modern computing machines, such machines in fact having their theoretical basis in mathematical thought. In this sense it is anachronistic to think of constructive mathematics as a post hoc restriction of mathematics to that which can be represented on a computer.

phrase used by Robert Harper.[4] That is, we will be defining all of mathematics, most notably *logic itself*, with respect to computation.

The intuitionist sees propositions and proofs on par with any other mathematical objects. Logic, by which we mean the topic of proofs and propositions, is not the underlying substratum upon which mathematics is built — computation is. Whereas classical mathematics begins with logic, we describe logic as a particular example of computation, giving rise to "proof-relevant" logic. We now attempt to pin down exactly what it means to base mathematics on top of computation, starting with a recapitulation of first-order logic for contrast.

First-order logic provides for classical mathematics the machinery necessary to formalize logic. Here our most fundamental notion is a that of a proposition, a statement of fact which is subject to proof or disproof, although at all times being either true or false. We formalize such a notion on the bedrock of first-order logic, which describes all of the syntactical rules for representing propositions as strings of symbols. We furthermore syntactically describe the rules of inference, making precise our thoughts about which kinds of propositions can be used to prove which others. A formal proof, then, is a series of justified inferences, and we judge a proposition to be (formally) provable whenever we can describe an appropriate proof. Provided with a list of axioms which abstractly characterize some class of objects, we can without ambiguity derive further propositions which any such objects (models) would also have to satisfy. One especially prominent device is set theory, itself characterized by a list of axioms, which we use to describe models for other mathematical theories. One of the most interesting observations that mathematicians have made is that the axioms of set theory themselves do not fully characterize any one particular model of set theory; regardless, axiomatic set theory is sufficiently powerful and well-behaved that it is often taken as the universe within which mathematics resides. [5]

[4] Quotation taken from Rober Harper. "Extensionality, Intensionality, and Brouwers Dictum." `http://existentialtype.wordpress.com/2012/08/11/extensionality-intensionality-and-brouwers-dictum/`

[5] Another fascinating and directly related observation comes from the incompleteness theorems of Gödel. These effectively imply that *any* reasonable foundation for mathematics will not be able

Intuitionistic type theory provides the machinery to formalize mathematics with respect to computation, rather than logic. Computation itself is a tricky notion to define, but certainly the word indicates constructive activities which can be described in an exact fashion. Intuitively, computation is whatever a computer can do. Perhaps this seems circular — the word "computer" conjures up thoughts about Boolean logic and mathematical formalisms — computer science is arguably a *branch* of mathematics, not its foundation. Indeed, many different mathematical formalisms have been offered as ways to formally describe what "computation" means — perhaps it is whatever can be represented with *Turing machines*, or perhaps with some variant of *lambda calculus*. Actually, all such notions have ended up being equivalent, and Gödel himself is quoted as have declared:

"... the concept 'computable' is in a certain definite sense 'absolute,' while practically all other familiar metamathematical concepts (e.g. provable, definable, etc.) depend quite essentially on the system to which they are defined." [6]

Therefore we are justified in taking "computation" as a primitive idea that we treat intuitively — to compute is to follow an exact procedure for calculating some kind of output. We take the words *construction*, *computation*, *calculation*, etc., as essentially synonymous. We also take *function*, *method*, *program*, *procedures* etc., as synonymous.

Propositions in an intuitionistic setting are not represented as strings of symbols for which we might find a formal proof, but as specifications of what kinds of proofs would need to be *constructed* to demonstrate their truth. Thus any proposition characterizes its proofs in its very definition, which is a tremendous departure from first-order logic. We consider a proposition to be true only when we have on hand a satisfactory proof-object, which can be called a *witness* to the proposition's truth.

to prove certain propositions expressed in the system. This observation is hardly limited to set theory and applies even to type theory.

[6] Translation of Martin Davis in *The Undecidable*, p. 83

In fact, we tend to avoid statements like

Proposition P is true.

The appropriate wording is something more like

Object x is a witness to the truth of proposition P

where x is some object defined by a constructive process and meeting the specification which defines P. We take the assertion "P is true" to mean "We know of (i.e. have some sort of access to) at least one object witnessing the truth of proposition P."

One can perhaps begin to see that a proposition here behaves almost like a set, since it can be considered the collection of all of its satisfactory proofs. A true proposition is one whose set of proofs we know to be non-empty. Here is where logic merges with type theory — this set-like notion is exactly that of a type whose terms are its proofs. The key insight embodied by intuitionistic type theory is that propositions are just a special kind of type, since other kinds of objects (like $\mathbb{N}$, as we shall see) can also be represented as types. It is for this reason that we say type theory subsumes logic, since our propositions and proofs are just special instances of the same kinds of objects we use to describe any other mathematical idea. This difference in perspective, as anyone would expect, has many striking consequences.

We return to the topic of logic in the section on propositions-as-types. First, we concentrate on developing our intuition for types in general, which are both analogous to sets and also drastically different from them.

3.2 Types as Constructive Sets

In the author's experience, the most significant obstacle to learning type theory is to rely too heavily on analogies with set theory. In fact, the reader is encouraged to appeal to intuition from computer programming, perhaps more so than from set

theory, since type theory is often a blend of both. The theory can be recognized as something like a programming language, and the rules are sufficiently precise that they can be implemented on a computer in the form of proof verifiers and related tools. The rough goal is to provide a method for constructing terms (mathematical objects) and to separate the different kinds of terms according to which sorts of operations can be applied to them. The operations themselves are used to build up new terms from existing ones, or else to keep track of which strings of symbols represent the same term. As with first-order logic, our objects are represented by strings of symbols, but here they are attached to extra data — we know which type each term is of, and we only apply operations to a term as permitted by the term's type. For instance, a term f might have the type of a function taking numbers as input and returning ordered pairs as output — terms of this type are constructed by providing a definite method for calculating their output. Another term n might have the type of a (natural) number, constructed by repeatedly applying the successor function to the basic term 0. With this information, we can invoke the operation of *function application* to construct a new term called $f(n)$ which, according to the type of f, has the type of an ordered pair. Then we invoke the *computation rule* for f to keep track of the fact that the term represented by "$f(n)$" is entirely equivalent to the ordered pair calculated by inputting the number n into the program used to define f in the first place. We can be confident that the underlying method defining f would return an ordered pair, since knowing f's type implies we are sure about the kind of output its method produces. Throughout this process of constructing terms, assuming we are in a computer setting, if ever we attempt to apply an operation that doesn't make sense — perhaps applying the function f to the wrong kind of data — then our computer environment will raise an error.

While we highlight the set-like nature of types, let us emphasize that type theory should be taken simply at face value, since whatever analogies we could offer tend to break down quickly. Even if types are like sets, the reader may not apply set-theoretic reasoning to reason about the properties of types. We are not working

within ZFC, and our objects are not literally sets in the sense represented by that system. Here, we mean nothing more than what is said, and we attempt to keep nothing but unnecessary formalities out of sight — the reader may not infer *any* properties of *any* objects but that which is grounded in appeal to the rules. Syntactically, one should avoid taking expressions like "(a, b)" or "f" as anything more than symbolic expressions whose behavior is being specified — our ordered pairs are not secretly a kind of set, and our functions are not secretly collections of ordered pairs. As with first-order logic, the symbols we use only have meaning insofar as we give them meaning with rules.

Despite our caution about treating types *too* much like sets, it is true that the type-theoretic notion most analogous to sets is that of types. In the literature, types are sometimes even called sets, terms are called elements, and the symbol : (introduced below) is sometimes replaced with $\in$. All types are at least somewhat like sets, and types are certainly act like collections of objects (terms), so this analogy is a fair starting point. It should be discarded whenever the two systems conflict, and not all ideas from one theory translate into the other. In sum, our caution is reflected in the following quote:[7]

> The easiest way to learn intuitionistic type theory is to disregard any preconceptions about logic and set theory and start afresh with the definitions and axioms of intuitionistic type theory.

Actually, this quote might be slightly misleading out of context, because type theory is not built from axioms in the usual sense. Type theory is not defined using first-order logic, and we are not using the common logical symbols given earlier, like

$$\forall, \exists, \neg, \implies, \iff, \wedge, \vee, =$$

[7] [Gra08], page xi.

which are more often associated with traditional logic. This abandonment of the classical logical basis for mathematics is what is meant in the following quote from Shulman: [8]

> Thus, type theory is not an alternative to set theory built on the same "sub-foundations"; instead it has re-excavated those sub-foundations.

Without even the sub-foundations of first-order logic, where do we begin? In the above treatment of set theory, we had to proceed in two quite distinct steps: first we described a system of logic, and then we defined a system of sets on top of it, so that our final foundational system consisted of two distinct layers (one layer containing the rules for sets, the other the rules for proofs and propositions in general). Type theory, in contrast to set theory, is self-contained — we will not need any supporting system like predicate calculus to define the free-standing framework of type theory. Logic — the topic of propositions and their proofs — will come *later* as a particular application of type theory. In a certain light, one could say that type theory defines logic on top of set theory, as Shulman has pointed out: Whereas classically, we start with logic and use it to define sets, in type theory we start with "sets" (types) and define logic in terms of sets (of proofs). Again, we iterate that our approach is *not* circular, since type theory is explained in terms of computation and we take computation as a primitive idea.

To the extent that types are like sets, then we might call them *constructive* sets, reflecting the computational emphasis in our approach. Here, "constructive" does not mean that for any type P we could list all of its terms. It *does* mean that we know how an arbitrary (possibly hypothetical) term of the type behaves computationally — what operations can be applied to it and how to perform them. One important difference between material sets and types is that the latter are not usually permitted to overlap — any particular term belongs exclusively to one type,

[8] Quote taken from Mike Shulman. "From Set theory to Type Theory." Online at `http://golem.ph.utexas.edu/category/2013/01/from_set_theory_to_type_theory.html`

since types are used to classify different sorts of terms. In set theory, of course, one element can be an element of many different sets, so we can see that taking the analogy too literally fails immediately.[9] In situations such as a disjoint union $A + B$ of types A and B, the type theorist prefers to make "copies" of each of the terms in A and B and considers the union to be a collection of such copies, rather than of the original terms.[10]

For readers with some familiarity with typing systems in computer science, we can say that our types are similar to the data-types used in some programming languages. Within a given programming language (with a strict typing system), any given object (variable or constant) is perhaps an integer or a character or else some other kind of thing — always *some* kind of thing, but not more than one kind of thing. Indeed, one of the key advantages of using types instead of pure sets is that types allow for keeping different objects "separated." Whereas in set theory, we can ask whether $(0, 0) = 2$ (and indeed, we may get an affirmative answer!), type theory will force us to ask only those questions which make sense. We will see below that in type theory, there is no such notion whatsoever as "$(0, 0) = 2$," and a software environment would object to the very idea.

Our account of type theory proceeds slightly less formally than the above treatment of set theory. This choice is sensible, since the formalities serve not to make the informal ideas more clear but more rigorous. However, we do not lose much in avoiding completely formalized type theory, since informal type theory does not stray very far from its formalism. This is especially true for our presentation in which we make little attempt to avoid using unfamiliar notation.

Let us begin to build up type theory. As with first-order logic, the rules of type theory are rules for forming judgments. First order logic has rules for forming

[9] Actually, the sets of structural theories tend to be disjoint as well. Type theory is quite like a constructive structural set theory.

[10] In terms of category theory, this requirement is effectively that a given morphism has just one codomain. For instance, a morphism $a : 1 \to A$ in **SET** is distinct from one $a' : 1 \to A + B$, the connection between the two morphisms being that $i_A \circ a = a'$ where i_A is the injection morphism of A into $A + B$.

judgments about, for instance, whether an expression P is a well-formed formula, whether it is a sentence (i.e. has no free variables), and whether P has a proof. Type theory contains its own rules for forming judgments. They are in fact at the core of type theory, since even the *definitions of particular types* are just rules for which kinds of judgments are permitted by the system. Judgments thus play a more prominent and diverse role in type theory than in set theory. Since the goal of any formal system is to codify which judgments are valid, it makes sense to start by describing the kinds of judgments that we intend to form using type theory. We are concerned with four different kinds:

- That some object P is a type, written $P : \mathcal{U}$
- That some object p is a term of a type P, written $p : P$
- That two types P and Q are judgmentally equal types, written $P \equiv Q : \mathcal{U}$
- That two terms p and q are judgmentally equal terms of some type P, written $p = q : P$

The colon symbol : is analogous to the set-theoretic $\in$, since $p : P$ expresses that p is a term (element) of P. The second kind of judgment, $p : P$, is called a *typing judgment*, a phrase we shall make use of. If $p : P$ is a correct typing judgment, then p is "well-typed." What about the apparently strange notation $P : \mathcal{U}$, which by analogy seems to suggest that P is a term of a type $\mathcal{U}$? In fact this is the case, but for now we may read $P : \mathcal{U}$ as "P is a type." We return to the matter below when we discuss dependent types.

Remark. *We will frequently speak of "a term $p : P$," but technically this is not quite right — what we really mean is a term p of type P, i.e. a term p for which $p : P$ is a valid judgment. This notational convenience is analogous to speaking of "an element $x \in X$," to indicate that x is some object and $x \in X$ is a property assumed of x. There is a slight but important difference between the two expressions though — $p : P$ is a judgment, a conclusion that we come to. We can prove or*

disprove $x \in X$ because it is a proposition (not a judgment), but we cannot disprove $p : P$ since the expression indicates an actual judgment that we form. The nuanced distinct will become more clear when we later look at propositions in type theory.

We do not mean with the above list of judgments that we are confining our terms and types to just individual symbols like 'P' or 'p.' Rather, we use are using a symbol like 'P' as a *meta-variable*, analogous to speaking of "a proposition P." 'P' could be a string of several symbols, like the strings "$A \times B$" or "$A \to B$." A term 'p' could also be a string of symbols, like (a, b) or $\lambda x.y$. Whenever symbolic expressions are used in type theory, we should read them as literal strings of symbols or meta-variables standing in for literal strings, as we later do with such phrases as, "where Φ is some expression" (see function types, below). Type theory, to a large extent, is a system of rules for rewriting strings of symbols. As with first-order logic, some of our symbols represent *variables*, some represent *constants*, while still others are simply punctuation. How a symbol is being used in a particular instance should be more clear with experience and context, although when necessary we can produce an exact codification of the proper syntax for type theory.[11]

Equality by judgment, or *judgmental equality*, is the strongest type-theoretic conception of equivalence between objects, although not the only one we will encounter. It represents the meta-theoretical notion of equality between p and q (or P and Q) *as symbolic expressions*. It may also be called *computational* equality (a term favored by the author), since two expressions are judgmentally equal if, in a certain sense, they evaluate to the exact same object. A very common phrase is *definitional* equality, since judgmentally equal expressions are so interchangeable that one is justified in saying they are equal by definition. We have found this phrase to slightly conflict with intuition, since it may happen that p is judgmentally equal to q without that equality having been explicitly stated as a definition (perhaps the equality comes as a consequence of some other rules). We will use

[11] As would be necessary, for instance, to create a proof-verifier or similar tool.

the phrase "definitional equality" to mean judgmental equality introduced with an explicit declaration that some expression is being defined as equal to another one.

Remark. *We usually write judgmental equality as just $p \equiv q$, leaving off the type P, when it is clear from context what P is. We also adopt the rules that each term and type is judged as equal to itself and that this equality behaves symmetrically and transitively.*

Equality is a very subtle topic in type theory, so we offer a few extra words about it at the risk of redundancy. Asking whether two expressions a and b are judgmentally equal exists at the same "level" as asking whether P is a well-formed formula in first order logic — both are valid questions to ask, but they are answered by examining the judgment rules of a formal system rather than the mathematical properties of the objects within the system. They are external in the sense that we would not represent such questions using the language *of* the formal system, but in our natural language while speaking *about* the system. For example, such a question is fundamentally different than asking whether two topological spaces are homeomorphic, which is a question we might ask "inside" type theory or set theory. Type theorists usually want to be sure that the question of whether two objects are judgmentally equal can always be answered algorithmically, although this cannot be guaranteed with some variants of type theory.

Judgmental equality is distinct from the notion of equality as a proposition. Propositional equality, which has yet to be introduced, is internal to type theory — it is the sort of relation between objects which we would prove, disprove, or hypothesize. Judgmental equality cannot be proved[12] and it cannot be hypothesized, since it merely relates raw different strings of symbols that, by design, represent the same object. Hypothesizing judgmental equality would be analogous to hypothesizing that a string of symbols is a well-formed formula, or that a purported proof is a

[12]Depending on what one means by "proof." The topic will be examined later.

valid one. As with many other aspects of type theory, the initial subtleties of judgmental equality become more clear with experience, especially after we introduce propositional equality in section 3.6.

The type of natural numbers

For basic practice with the four forms of judgment, let us first describe how to represent natural numbers with type theory. This process will also provide exposure to the general pattern of defining a new type, since most types are defined in a standardized way. The collection of natural numbers, which as usual we denote $\mathbb{N}$, will be represented as a type whose terms are the natural numbers.

It is customary in the literature to define the rules of type theory using *natural deduction*. Each rule expressed with this method describes a judgment which can be formed under the appropriate hypotheses. We write the rules in this form:

$$\frac{\text{Premises}}{\text{Permitted Conclusion}}$$

Above the line is a collection of judgments necessary before the conclusion is permitted, separated by large spaces as necessary. Below, we write the judgment which we declare can be inferred from them. The first rule for natural numbers is

$$\frac{}{\mathbb{N} : \mathcal{U}}$$

This rule says that, without any other conditions, $\mathbb{N}$ is a type. It is called the *formation rule* for $\mathbb{N}$, since it tells us when we may form the type. In this case, we may form the natural numbers type without knowing anything else.

We have two more rules

$$\frac{}{0 : \mathbb{N}}$$

$$\frac{n : \mathbb{N}}{\text{succ}(n) : \mathbb{N}}.$$

These rules are called the (*term*) *introduction rules* for the natural numbers, since they introduce terms of the type. These rules intuitively state that 0 is a natural

number and that, for any natural number n, its successor is also a natural number. The symbols $\mathrm{succ}(n)$, we state for emphasis, are simply symbols, where n is acting as a variable. Thus, 0, $\mathrm{succ}(0)$, $\mathrm{succ}(\mathrm{succ}(0))$, etc., are all terms of the type $\mathbb{N}$. If we want to use more common notation, we can introduce new symbols $1, 2, 3, \ldots$ and lay down the appropriate definitional equalities $1 \equiv \mathrm{succ}(0)$, $2 \equiv \mathrm{succ}(1)$, etc. Notice that by the substitution rule of judgmental equality, we have $\mathrm{succ}(1) \equiv \mathrm{succ}(\mathrm{succ}(0))$, so transitivity gives us $2 \equiv \mathrm{succ}(\mathrm{succ}(0))$.

We specify no other introduction rules for $\mathbb{N}$, so we have no way to judge that anything not given from the above rules is a term of $\mathbb{N}$. Can we then prove with type theory that each term of $\mathbb{N}$ either 0 or a successor? We cannot, yet, even though meta-theoretical inspection tells us that such is the case. Within type theory, we only know what is expressed by the judgments: $\mathbb{N}$ is a type containing 0 as a term and that the type is closed under the taking of successors. At this stage, these judgments are all we may form regarding the type $\mathbb{N}$, so we cannot take any other properties of the natural numbers for granted. We are in fact quite far from being able to do any kind of proofs — we have not even explained what a proof is. Later, when we have encountered function types, we will provide two other collections of rules for $\mathbb{N}$ besides the formation and introduction rules above. We shall later generalize those rules using dependent types. Then, after we have incorporated logic into type theory, we shall be able to prove (within type theory) that all terms of $\mathbb{N}$ are either 0 or a successor. For now, we return to a broader examination of type theory.

The universe of type theory starts off with a handful of basic types like $\mathbb{N}$. For instance, we can define a type $\mathbf{0}$ behaving like the empty set (which obviously would have no introduction rules), and a type $\mathbf{1}$ which acts like a generic singleton set. For now, let us take some basic types for granted.

$$A, B, C, \ldots, \mathbf{0}, \mathbf{1}, \mathbb{N}, \mathrm{Unit\_type}, \mathrm{basic\_type\_number2}$$

The latter two names have a distinctly computer science ring to them, but we include them here to emphasize that technically they are valid names of types. For mathematical work, we will use generic names like A and B for general types, or special names like $\mathbb{N}$ and $\mathbf{0}$ for particular types.

Initially we may start off with only a few types, but type theory provides a handful of *type forming operations*, which are methods of inductively constructing new types from the ones that already exist. For instance, we will define an operation $\to$ to form function types

$$A \to B,\ \mathbb{N} \to C, \ldots$$

There is also a method to form product types, analogous to Cartesian products, which in conjunction with function types allows us to form types like:

$$A \to (C \times B),\ (C \times A) \times (\mathbb{N} \to \mathbb{N}), \ldots$$

There is also a disjoint union (sum) operation $+$, so we can form types like

$$A \to (C \times (\mathbb{N} + A)),\ (C + A) + (B \times \mathbb{N}), \ldots$$

Before visiting the product and sum types in detail, we first describe the function types. Since functions are a somewhat special topic, the function type will expose us to quite a bit of additional notation that will take some getting used to. But we need to know about function types to more fully characterize $\mathbb{N}$ and the other type forming operations.

Function Types

If we have judged that A and B are types, we may form the type $A \to B$ which acts as the collection of functions from A to B. That is, the formation rule for function types is

$$\frac{A : \mathcal{U} \qquad B : \mathcal{U}}{A \to B : \mathcal{U}}$$

This formation rule has non-trivial premises — we must know that A and B are types before we may speak of the type $A \to B$. Notice that we can iterate the rule

— after judging that $A \to B$ is a type, we are permitted by the same rule to infer that $A \to (A \to B)$ is a type. In general, this process of inductive constructions is part of what makes type theory so powerful. Still, knowing that $A \to B$ is a type, by itself, is not very interesting. Lacking the term introduction rules, for example, we cannot yet say that the type has any terms, so we turn our attention to precisely that.

Functions are a more primitive concept here than in set theory, where functions are encoded as certain binary relations. Here, we define a particular function in type theory by giving explicit rules for applying it (following a standard pattern for such specifications). That is, a term f of the function type $A \to B$ is given by explicitly describing a method for accepting any term a of A as input and returning a term $f(a)$ of B. In effect, functions are computer programs accepting input of one kind and returning output of another kind.

We often define functions using *lambda abstraction*, for which we introduce some new notation. A typical introduction of a function being defined by lambda abstraction looks like this

$$\lambda(x : A).\Phi : A \to B.$$

The symbol Φ stands in for any expression (as foreshadowed earlier), usually one involving the symbol x as a variable. The λ notation intuitively indicates a function which, given some input of type A, returns the object Φ', where the latter expression is the result of substituting the input in place of each occurrence of x in the expression Φ. We are still in the process of describing the rules for functions, so none of these properties are formally represented yet, but that is the behavior we want the rules to capture. If, assuming x is some arbitrary input of type A, the expression Φ is a term of type B, then the above typing judgment would be justified by the following introduction rule for function types

$$\frac{x : A \vdash \Phi : B}{\lambda(x : A).\Phi : A \to B}$$

The premise uses new notation, the turnstile symbol $\vdash$. When a judgment is written with a turnstile, everything to the left of the turnstile is thought of as something like a local hypothesis for the judgment. It is called the *context* of the judgment, and a context may be called a list of *assumptions* about the variables appearing to the right of $\vdash$. The judgment itself may be called a hypothetical judgment, although the whole expression is a bona-fide judgment and not necessarily being "hypothesized" in any sense.

Remark. *There is a fundamental distinction between the premises of a rule formulated through natural deduction and the hypothesis of a judgment. So important is this distinction that we could not omit such a large remark as this one. The rules do not live "inside" type theory but instead define it. The judgments live inside type theory and are governed by the rules. Therefore, premises of a rule are meta-theoretical — each rule permits a certain judgment under the "hypothesis" that we the mathematicians have formed certain other judgments. The hypotheses of a judgment are an intrinsic component of the judgment being made by we the mathematicians. It is sensible for a hypothetical judgment to appear as part of the premises or conclusion of a rule, but nonsensical to use a rule in this manner.*

In many ways, a hypothetical judgment is similar to a proposition like $A \implies B$ in first-order logic, which is intended to represent the idea that B is true under the hypothesis A. The familiar behavior of such an expression (which lives inside predicate logic) comes from the inference rules of the logic, which themselves live in the meta-theory (the English language, maybe). For instance, the inference rule of predicate logic imply that that if $A \,\&\, (A \implies B)$ is judged true, then B is true. That rule is why $A \implies B$ can be read conventionally, though it is ultimately just a string of symbols.

Similarly, the expected properties of a "hypothetical" judgment comes from the rules of type theory. These rules are set up such that $x : A \vdash \Phi : B$ is inferrable if we can infer $\Phi : B$ by invoking rules which possibly require the judgment $x : A$

as a premise, just as predicate logic is set up such that $A \implies B$ is provable if we can prove B from inference rules depending on A as a premise. See page 67 for an example of a formal derivation involving hypothetical judgments.

The context of a hypothetical judgment contains information about how the variables in the right-hand expression are to be interpreted. Whereas the expression $\Phi : B$ is read "Φ is a term of type B," the judgment $x : A \vdash \Phi : B$ may be read "In the context where x is a general (fixed) term of the type A, the expression Φ is of type B." In such a context, each occurrence of the variable x in Φ is treated as an arbitrary term of A, rather than as a variable. More concisely, the judgment is read "Φ is a term of type B if we assume that x is a (general) term of A." The introduction rule for function types states that if this judgment is valid, then the function mapping input of type A to the (substituted) expression Φ' is a term of the appropriate function type.

For instance, we can introduce a new function

$$\lambda(x : \mathbb{N}).\mathrm{succ}(x) : \mathbb{N} \to \mathbb{N}.$$

The term (in this case, the function) is well-typed because, by one of the introduction rules for $\mathbb{N}$ listed earlier, whenever x is a term of $\mathbb{N}$, then so is $\mathrm{succ}(x)$. Thus we can infer the hypothetical judgment $x : \mathbb{N} \vdash \mathrm{succ}(x) : \mathbb{N}$ per the relevant introduction rule for the natural numbers type. Therefore, since we have established the premise of the introduction rule for function types, we may form above judgment introducing the "successor function" on $\mathbb{N}$. The entire judgment may be read "The function mapping a natural number x to its successor is a function from the natural numbers to the natural numbers." For the formally-inclined, and to provide a demonstration that $x : \mathbb{N} \vdash \mathrm{succ}(x) : \mathbb{N}$ is a consequence of the introduction rules for $\mathbb{N}$, a formal derivation of this judgment is given on page 67.

If we wish to use more common notation for functions, we may introduce a symbol like f and lay down a definitional equality like

$$f \equiv \lambda(x : \mathbb{N}).\mathrm{succ}(x)$$

so that the left- and right-hand expressions are interchangeable. However, we have not seen how to *use* functions yet — we cannot yet say (given $n : \mathbb{N}$) that $f(n) : \mathbb{N}$, nor that $f(n) \equiv \text{succ}(n)$, although that is the ultimate goal. For this purpose, we continue listing the rules for function types.

We have so far given two kinds of rules, formation and introduction, for both the type $\mathbb{N}$ and the type forming operation $\to$. There are two other sorts of rules, and now we can present both sorts for both types. After the introduction rule, one usually gives the *elimination rule* for a type. The elimination rule is used to specify how, given a general term of the type, to form a term of some other type. For function types, elimination is this rule:

$$\frac{f : A \to B \qquad a : A}{f(a) : B}$$

The rule says that if f is a function from A to B (in other words, a term of the function type) and if a is a term of type A, then we may infer that $f(a)$ is a term of B. Therefore, for the successor function described earlier, we can say $f(n) : \mathbb{N}$, as intended, as long as we know $n : \mathbb{N}$. But the string "$f(n)$" is taken literally — we cannot simplify it to some other form like $\text{succ}(n)$. To be able to rewrite $f(n)$ in some other form, we give next the *computation rule* for function types, which clarifies the elimination rule. In this case, it is

$$\frac{x : A \vdash \Phi : B \qquad a : A}{(\lambda(x : A).\Phi)(a) \equiv \Phi[a/x] : B}$$

The notation $\Phi[a/x]$ means the result of replacing each occurrence of the variable x in Φ with a. The computation rule intuitively states that a function specified by lambda abstraction, when applied to its input, acts in the manner specified earlier — substitution of the input into the expression Φ following the '$.$' . Therefore, we can now say $f(n) \equiv \text{succ}(n) : \mathbb{N}$, so our function behaves exactly as expected. For function types, there is actually one more rule, called a *uniqueness rule*, which states

that any function f is equal, by definition, to the function mapping x to $f(x)$, which is obviously desirable.

$$\frac{f : A \to B \qquad a : A}{f \equiv (\lambda x.f(x))}.$$

We have now fully defined functions in type theory, so anything else we could say about functions would have to be justified by the given rules.

Remark. *Instead of writing $\lambda(p : P).\Phi : P \to Q$, we often write $\lambda p.\Phi : P \to Q$. This is not ambiguous, because to say that the function is a term of $P \to Q$ is to say that the input p must be of type P.*

What about functions of several variables? One common technique is called *currying*, named after Haskell Curry. To define a function of two variables, say of types A and B with codomain C, we provide a function-valued function f of type $A \to (B \to C)$. Thus $f(a)$ is function accepting an input of type B, and $f(a)(b)$ is a term of C.

Remark. *Very commonly, we write $f(a)(b)$ as just $f(a, b)$, which should not create any confusion. Also, we usually write $A \to (B \to C)$ as just $A \to B \to C$, the convention being that we associate to the right. Thus $A \to B \to C \to D$ means $A \to (B \to (C \to D))$, and $f(a, b, c, d)$ means $f(a)(b)(c)(d)$.*

It may seem that λ abstraction is a limited method for specifying functions, since one can perhaps imagine many functions for which we could not find a description of the appropriate form. Actually, the technique is quite flexible. We will soon demonstrate this by defining an addition function on the natural numbers.

Having defined function types, we may provide the elimination and computation rules for the type $\mathbb{N}$. Oftentimes, as in the case of $\mathbb{N}$, the elimination rule for a type provides a way to defining functions whose domain is the type. The elimination rule presented here is called the non-dependent elimination rule, to distinguish it from the later (more general) dependent version. While the strange notation may

give the rule a complicated appearance, the rule is conceptually simple. Recall that these rules are to be taken at face value, so we read all of the expressions literally.

$$\frac{C : \mathcal{U} \qquad c_0 : C \qquad c_s : \mathbb{N} \to (C \to C) \qquad n : \mathbb{N}}{\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(n) : C}$$

There are two computation rules for $\mathbb{N}$. They are

$$\frac{C : \mathcal{U} \qquad c_0 : C \qquad c_s : \mathbb{N} \to (C \to C)}{\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(0) \equiv c_0 : C}$$

$$\frac{C : \mathcal{U} \qquad c_0 : C \qquad c_s : \mathbb{N} \to (C \to C) \qquad n : \mathbb{N}}{\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(\mathrm{succ}(n)) \equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(n)) : C}$$

The elimination rule, which we say embodies the *recursion principle* for $\mathbb{N}$, gives us a powerful way to define functions on $\mathbb{N}$. To define a function g of natural numbers, we provide a few things in advance: a type C, an object c_0, and a function c_s which is effectively a function of two variables (one of $\mathbb{N}$ and one of C). C, of course, is our intended codomain. c_0 is the object to which we intend to map 0. The function c_s is used to define g on the input $\mathrm{succ}(n)$ in terms of n and $g(n)$. Specifically, the value of $g(\mathrm{succ}(n))$ will be defined as $c_s(n, g(n))$, so we call c_s the "next step" function for g. The recursion principle states that if we have all of these things, then given any $n : \mathbb{N}$, the expression $\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(n)$ is a term of the intended codomain C of g. In conjunction with the introduction rule for function types, we can say that

$$\lambda(x : \mathbb{N}).\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(x) : \mathbb{N} \to \mathbb{N}$$

This is the function which we will call g. The reader can verify (somewhat informally) that the recursion principle for $\mathbb{N}$ supplies us with the right kind of judgment necessary to introduce a term of type $\mathbb{N} \to C$, since we can infer $x : \mathbb{N} \vdash \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(x) : C$ from it. That is, we can infer that, assuming $n : \mathbb{N}$, the expression $\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(n)$ is a term of C. Taken together, the computation rules for $\mathbb{N}$

and function types give our function g the desired computational behavior, which the reader should verify. We say that g is defined by *primitive recursion*.

The elimination rules represents a certain minimality principle for $\mathbb{N}$, since it permits us to define functions on $\mathbb{N}$ by supplying only enough information to define them on 0 and its successors. We will later use the elimination rule to *prove* the minimality of $\mathbb{N}$ by constructing a function associating an *arbitrary* term n of $\mathbb{N}$ to a proof that n is one of the desired numbers.

For practice, let us use the rules given so far to define addition of natural numbers — this is a natural candidate for a function specified by primitive recursion, since it is convenient to define addition of larger numbers in terms of addition on smaller numbers. Of course, addition is a function of two variables, whereas the recursion principle gives us a way to define functions of a single natural number. To overcome this, as described above, we use currying and define addition as a function of type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. We will map each term n to the function sending m to the usual $n + m$. Thus, we first define our codomain as $C \equiv \mathbb{N} \to \mathbb{N}$. What is our c_0, the term to which we map the number 0? Clearly we should map 0 to the function mapping each number n to $n + 0 = n$, in other words the identity function. Thus, we define $c_0 \equiv \lambda(x : \mathbb{N}).x : \mathbb{N} \to \mathbb{N}$, which is obviously well-typed. What function is c_s? It must be a function of type $\mathbb{N} \to [(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})]$. In this case, we define $c_s \equiv \lambda(x : \mathbb{N}).\lambda(f : \mathbb{N} \to \mathbb{N}).\lambda(n : \mathbb{N}).\mathrm{succ}(f(n))$, which is evidently of the appropriate type.[13] Now c_s takes two inputs, a natural number x and a function f of natural numbers, and returns a function sending n to the usual number $1+f(n)$. We lay down the definitional equality $\mathrm{add}(x, y) \equiv \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s)(x, y)$ and see that add is a effectively function of two natural numbers, as intended. Then the computation rules for $\mathbb{N}$ can be seen to give us these desired equalities:

$$add(0, n) \equiv n$$

[13]Notice how we can notationally chain together functions in this manner. We are not actually introducing new notation, since the reader should be able to verify that our expression is in accordance with the rules for lambda-abstraction.

$$\mathrm{add}(\mathrm{succ}(n), m) \equiv \mathrm{succ}(\mathrm{add}(n, m)).$$

To match our usual notation, we introduce a symbol $+$ and define $m+n$ to mean $\mathrm{add}(m, n)$. Then, for example, we could introduce a function like $(\lambda(x : \mathbb{N}).x + x) : \mathbb{N} \to \mathbb{N}$, which doubles its input.

If this all seems to be too much work to justify using type theory, recall that we are being unusually formal and detailed. A comparable definition of addition on the natural numbers in ZFC would also look unappealing. As with set theory, sufficient exposure to type theory will allow us to disregard this much detail in our definitions. Moreover, this approach is much more convenient (with practice) to formalize directly into a computer environment, where it can be done just once and then saved into a library of definitions. When we later define $\prod$ types, we will see in fact that we could have bypassed much of this work and defined addition with just one line.

Remark. *As a word of caution, let us point out the following, for example: Meta-theoretically, it appears that the function type* $\mathbb{N} \to \mathbb{N}$ *is a countable collection of terms — The only functions we introduce must be computable, and it is a basic fact that the set of such functions is only countably infinite. This is a misunderstanding from applying too much meta-theoretical set-theoretic reasoning to types. Within type theory, the type* $\mathbb{N} \to \mathbb{N}$ *is the collection of all functions sending the natural numbers to themselves. Type theory is not internally "aware" that we are only considering constructions. As stated above, we do not necessarily need to be able to list every term of* $\mathbb{N} \to \mathbb{N}$ *to call it a type — what is important is that we know what properties an arbitrary term has, and in this case an arbitrary term is method for mapping* $\mathbb{N}$ *to* $\mathbb{N}$*. It happens that we only introduce (and manipulate) the terms which are computable, which is distinct from saying that "all" of its terms, internally to the theory, are computable functions.* [14]

[14] Robert Harper discusses this topic as it relates to the related system of *extensional* type theory in post entitled "Church's Law." It can be found at `http://existentialtype.wordpress.com/2012/08/09/churchs-law/`

Our presentation thus far has been much more formal than some other accounts, with the advantages of precision, exposure to formality, and reduced length. The caveat is of course technical density. We recapitulate what has been said, discarding formalities.

Usually, a type (or a type-forming operation) is defined by specifying four rules: formation, introduction, elimination, and computation. Their usual purpose is as described in the table, for a type P and terms p.

Table 3.1: Explanation of rules for types

Rule ...	Answers ...
Formation	When can I say that P is a type?
Introduction	How can I make a term p of P?
Elimination	What terms of other types can I make with a general term p?
Computation	(Controls the elimination rule)

The given ordering of the rules is sensible, since each rule often implicitly depends on the rules above it. For instance, the formation rule usually specifies that P is a type only when some other things are types. Then the introduction rule often states that p is a term of P only if symbols appearing in the expression p are themselves terms of the types in the premise of the formation rule.

The elimination rule is often used to specify when one can define a function with domain P (hence, "eliminating" a term p to produce a term of some other kind). Recall that the premise for introducing a term of $P \to Q$ is that one knows that some expression Φ is a term of type Q under the assumption that some variable in Φ represents a general term of P — such a judgment would often be inferred by appealing to the elimination rule for P. Of course, elimination for P is not the *only* method of defining a function on P, as earlier we directly defined a function $n \mapsto \mathrm{succ}(n)$ on $\mathbb{N}$ without resorting to primitive recursion.

The computation rule controls the elimination rule — elimination usually gives rise to a function with domain P, and computation specifies the values of that function by laying down appropriate judgmental (i.e. computational) equalities.

These rules are not set perfectly in stone. Some types have additional rules (like the uniqueness rule for function types). Some types lack a given rule (the empty type **0** has no introduction rules). Also, each "rule" might actually be given by several rules (like the two computation rules for $\mathbb{N}$). But, somewhat remarkably, this general scheme works for defining any kind of type. Here is effectively what we have said of function types and the natural numbers.

Table 3.2: Rules for function types

Rule ...	Effectively states ...
Formation	$A \to B : \mathcal{U}$ when $A : \mathcal{U}$ and $B : \mathcal{U}$
Introduction	If $x : A$ implies $\Phi : B$, then there is a function called $\lambda(x : A).\Phi : A \to B$
Elimination	If f is a term of $A \to B$ and $a : A$, then $f(a) : B$
Computation	If $f \equiv \lambda(x : A).\Phi$, then $f(a)$ is the result of substituting a into Φ.

Table 3.3: Rules for natural numbers type

Rule ...	Effectively states ...
Formation	$N : \mathcal{U}$
Introduction	$0 : \mathbb{N}$ and all successors are terms
Elimination	Given a domain C, a first value c_0, and a "next step" function c_s, then there is a function $f : \mathbb{N} \to C$
Computation	The value of $f(0)$ is c_0 and $f(\text{succ}(n))$ is $c_s(n, f(n))$

Our descriptions of the type-forming operations $\times$ and $+$ discard lengthy formalities and instead directly address the underlying significance of the defining rules.

Cartesian Product

Formation and introduction work exactly as expected — $A \times B$ is a type when A and B are types, and (a, b) is a term when $a : A$ and $b : B$.

The elimination rule has the effect that a function g of type $A \times B \to C$ may be defined by providing a function $f : A \to (B \to C)$, in other words a function of two variables, one of type A and one of B. More formally, the premise of the elimination rule is that Φ is a term of C under the assumption that the variable x

Table 3.4: Rules for product types

Rule ...	Effectively states ...
Formation	If $A : \mathcal{U}$ and $B : \mathcal{U}$ then $A \times B : \mathcal{U}$
Introduction	If $a : A$ and $b : B$, then $(a, b) : A \times B$
Elimination	If $x : A, y : B \vdash \Phi : C$, then one can define a function $(a, b) \mapsto \Phi[a, b \,/\, x, y]$
Computation	The value of the function is given by substitution of a and b into the variables of c.

is a term of A and variable y is a term of B. Computationally, $g((a, b))$ is evaluated by substituting a for x and b for y in the expression Φ, as expected. The notation $\Phi[a, b \,/\, x, y]$ means the result of replacing x with a and y with b.

The recursion principle for $\mathbb{N}$ was that functions with domain $\mathbb{N}$ can be defined by primitive recursion, and moreover we only need to define such functions on 0 and successors. The analogue here is that functions can be defined on $A \times B$ by treating the components of a pair (a, b) separately, and moreover we only need to define functions on pairs. For instance, we can define the left and right projections: $\pi_1((a, b)) \equiv a$ and $\pi_2((a, b)) \equiv b$.

Disjoint Sum

Table 3.5: Rules for sum types

Rule ...	Effectively states ...
Formation	If $A : \mathcal{U}$ and $B : \mathcal{U}$ then $A + B : \mathcal{U}$
Introduction	If $a : A$ then $\mathrm{inl}(a) : A$. If $b : B$ then $\mathrm{inr}(b) : B$.
Elimination	If $f_0 : A \to C$ and $f_1 : B \to C$, then one can define a map $g : A + B \to C$
Computation	The value of that map is given in cases. If p came from $a : A$ then $g(p)$ is $f_0(a)$. If p came from $b : B$ then $g(p)$ is $f_1(b)$.

The disjoint sum has two introduction rules, one for A and one for B. The rules state that, for both types, each of its terms x yields a term $\mathrm{in}*(x)$ of the union. $*$ is either l or r, keeping tracking of which type A or B the term x came from. The two $\mathrm{in}*$ "operations" are effectively the two induced (left and right) injections of sets

A and B into their disjoint union. More generally, they correspond to the left and right injections into a coproduct in a category.

Notice that there is no term common to both A and $A + B$, since a and $\text{inl}(a)$ are judgmentally distinct terms. In other words, the injection may be thought of as making copies of each term in A, rather than allowing for the two types to overlap. The same is true for terms of B.

The analogue to the recursion principle here is *case analysis*. A function is defined on $A + B$ as soon as we specify its values on (the copies of) A and B, and to define a function f with domain $A + B$, we are allowed to split our definition of $f(p)$ into two cases: one where p is a copy of a term of A, another one where p is a copy of a term of B. That is, our definition is allowed to "know" what kind of term p corresponds to and proceed accordingly.

A type of lists

For even more practice with type theory, we will define a type of *lists* of natural numbers. This type would usually be more interesting to computer scientists than general mathematicians, but demonstrates the power of type theory as a tool for computer science. After this definition, we move on to dependent types, which we will ultimately see should be very interesting to mathematicians.

Rule ...	Effectively states ...
Formation	If $\mathbb{N} : \mathcal{U}$ then $\text{List}(\mathbb{N}) : \mathcal{U}$
Introduction	nil:List($\mathbb{N}$). If l :List($\mathbb{N}$) and $n : \mathbb{N}$ then $n :: l$:List($\mathbb{N}$)
Elimination	Given a domain C, a "nil" value $l_0 : C$, and a "next step" function $l_s : (\text{List}(\mathbb{N}) \to C \to C)$ then there is a certain $f : \text{List}(\mathbb{N}) \to C$
Computation	$f(\text{nil}) = l_0$. For $n : \mathbb{N}$ and $l : \text{List}(\mathbb{N})$, $f(n :: l) = c_s(l, f(l))$

3.3 Dependent Type Theory

Much of the expressive power of intuitonistic type theory is found in its notion of *dependent* types, making it one kind of *dependent* type theory. Before we define the notion, let us first explain the notation $P : \mathcal{U}$. $\mathcal{U}$ is, as the notation indicates,

itself a type. It may also be written as $\mathcal{U}_0$. It is the type of all (small) types, very loosely like the universe V of all sets. It is called the (first) *universe*, and its terms are precisely the types that we have been examining thus far. Although a type, $\mathcal{U}$ cannot be a term of itself (or we encounter contradictions similar to Russell's paradox. See [ML98]). Instead, we describe a hierarchy of universes and say $\mathcal{U}_0 : \mathcal{U}_1$, $\mathcal{U}_1 : \mathcal{U}_2$, etc. While the judgment $P : \mathcal{U}$ may be read as "P is a type," technically is a typing judgment that P is a term of the universe $\mathcal{U}$. With this in mind, although we described four forms of judgment above, we are actually concerned just with two — typing judgments and definitional equality between terms (possibly of a universe). Universes do not concern us here and we shall not discuss them at length. All we need to know is that $\mathcal{U}$ is a technically a type, one whose terms are themselves types.

A dependent type P over a term A, despite the name, is a *term* of the type $A \to \mathcal{U}$, in other words a function mapping each term $a : A$ to a term $P(a) : \mathcal{U}$, where the term $P(a)$ is a term of the universe, hence a type. We can see that a dependent type really just a *family of types* indexed by some base type A, quite analogous to an indexed family of sets.

Dependent Function Types

The dependent function types, which we also call $\prod$ types ("pi types"), generalize the more basic function types. A term of a $\prod$ type is a function whose codomain may vary based on its input, and naturally we call these dependent functions.

To define a type of dependent functions with domain A, we first need a dependent type $B : A \to \mathcal{U}$. B keeps track which inputs should be mapped to which codomains. The term a will be mapped to a term of $B(a)$.

Given such a family B, we may form the type $\prod_{(x:A)} B(x)$. A term is any method of mapping an arbitrary term $x : A$ to a term of the type $B(x)$. As with ordinary function types, these functions may be introduced with lambda abstraction — if we know Φ is a term of $B(x)$ in the context that x is any term of A, then the function mapping $a : A$ to the substituted expression Φ' is a dependent function with domain A and codomain family B.

Remark. *Notationally, we scope over the entire expression following the* $\prod_{x:A}$ *unless delimited with punctuation. That is,* $\prod_{(x:A)} B(x) \to A \to \mathbb{N}$ *would be read as* $\prod_{(x:A)}[B(x) \to A \to \mathbb{N}]$ *rather than* $[\prod_{(x:A)} B(x)] \to A \to \mathbb{N}$. *Additionally, it is customary to abbreviate an expression like* $\prod_{(x:A)} \prod_{(y:A)} B$ *to simply* $\prod_{(x,y:A)} B$.

Table 3.6: Rules for Π types

Rule ...	Effectively states ...
Formation	$\prod_{x:A} B(x) : \mathcal{U}$ when $A : \mathcal{U}$ and $B : A \to \mathcal{U}$
Introduction	If $x : A$ implies $\Phi : B(x)$, then $\lambda(x : A).\Phi : \prod_{x:A} B(x)$
Elimination	If f is such a function and $a : A$, then $f(a) : B(a)$
Computation	If $f \equiv \lambda(x : A).\Phi$, then $f(a)$ is the result of substituting a into Φ.

If B is actually a constant function (so that we can informally say B is just a type itself, rather than a term of a function type) then clearly our $\prod$ type reduces to just the regular function type $A \to B$. In fact we can retroactively define the non-dependent functions as dependent functions with a constant codomain. [15]

$\prod$ types are extremely useful. For instance, we can use them to "package" the elimination rules for other types. Consider $\mathbb{N}$, whose elimination rule specifies that a function f from the natural numbers into C is defined by giving a value for $f(0)$ and providing a function c_s such that we may define $f(\text{succ}(n)) \equiv c_s(n, f(n))$. This elimination rule is equivalent to the existence of a certain dependent function which can be called the *recursor* for $\mathbb{N}$, specifically

$$\text{rec}_{\mathbb{N}} : \prod_{C:\mathcal{U}} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$$

[15] The very careful reader may notice a circularity here, since we have said that one forms a dependent function by first giving a dependent type $B : A \to \mathcal{U}$, which itself is a function. This would apparently imply that we must define ordinary functions before defining dependent functions. Actually, the formal premise in the formation rule of pi-types is not that $B : A \to \mathcal{U}$ but instead that $x : A \vdash B : \mathcal{U}$. In practice, type theorists tend to blur the distinction between the judgments $p : P \vdash q : Q$ and $q : P \to Q$, or even the *rule* that $q : Q$ under the premise $p : P$. This does not create problems but initially may be confusing, since formally these notions are different.

(recall all of the scoping conventions we have specified) which satisfies the equations

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, 0) \equiv c_0$$

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) \equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n)).$$

Then we can just define

$$\mathrm{add} \equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \lambda n.n, \lambda n.\lambda f.[\lambda m.\mathrm{succ}(f(m))]) : \mathbb{N} \to C$$

We offer a brief analysis of this definition, although it only uses familiar ideas so far. Addition is a function of two natural numbers, so we must use the recursion principle to map the number n to a function, namely the function which sends a number m to the sum $m + n$. Thus we feed the recursor the following information: what kind of term n should be mapped to (a function from the numbers to the numbers), where to map 0 (the identity map), and where to map $n + 1$ assuming we know where to map n (simply to the map sending m to $n + m + 1$). A good exercise would be to examine why this new definition of "add" is correct. While the definition may seem complicated, with practice such definitions become rather intuitive.

In addition to defining a function which encapsulates another type's elimination rule, $\prod$ types are used to *generalize* the elimination rules. Recall that we have been using elimination rules for a type P to specify how to construct new terms (of a provided type) from the terms p of P. Effectively, elimination rules provide a way to define functions. We want to generalize this idea, and in fact we are required to — the more general dependent functions are introduced by a more general premise (allowing for a variable codomain), so we need to "upgrade" the basic elimination rules in order to allow for this. It is very straightforward to generalize the elimination rules we have given so far — instead of requiring a codomain $C : \mathcal{U}$ to be given, we require that C be a family $C : P \to \mathcal{U}$. Any other adjustments should be obvious after we give a specific example for $\mathbb{N}$.

A more general eliminator for $\mathbb{N}$, which may be called its *induction principle*, requires a dependent type $C : \mathbb{N} \to \mathcal{U}$ and leads us to a function $\prod_{(n:\mathbb{N})} C(n)$. Whereas the premise of the basic elimination rule required a term $c_0 : C$, naturally we now require a term $c_0 : C(0)$. We also require our "next step" function c_s to be a dependent function of type $\prod_{(n:\mathbb{N})} C(n) \to C(\mathrm{succ}(n))$, [16] matching our intention to define $f(\mathrm{succ}(n))$ as $c_s(n, f(n))$. Similarly to the recursion principle, the induction principle can be conveniently packed into a dependent function called the *inductor* for $\mathbb{N}$. It accepts C, c_0, and c_s to return a term of $\prod_{n:\mathbb{N}} C(n)$. It is a term

$$\mathrm{ind}_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}} C(0) \to \left[\prod_{n:\mathbb{N}} C(n) \to\ C(\mathrm{succ}(n))\right] \to \prod_{n:\mathbb{N}} C(n)$$

subject to the familiar computational equalities

$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, 0) \equiv c_0$$

$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) \equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n)).$$

The elimination for each type can be generalized to a dependent version, and in general we say the dependent rule represents the "induction principle" for the type. For instance, when one invokes the elimination rule of sum types to define a function on $A + B$ by case analysis, then one is invoking "induction on sum types." The motivation for this terminology will be explained shortly when we discuss propositions-as-types in section 3.4.

The notation $\prod$ seems reminiscent of multiplication, whereas mathematicians know that forming a collection of functions is more akin to exponentiation. Actually, the operator $\prod$ is recognizable as a kind of product, the Cartesian product indexed over the base type A. A $\prod$ type is in fact a "product of types over A" whose individual terms — the dependent functions with domain A — are themselves lists of terms (of various types) indexed by A. Effectively, each dependent function is an element of the product of the types $B(x)$ as x ranges over A.

[16] Again, recall our scoping conventions. $c_s(n)$ is a term of $C(n) \to C(\mathrm{succ}(n))$

Dependent Pair Types

Dependent pair types, or $\sum$ types ("sigma types"), generalize the more basic product types in the same way that $\prod$ types generalize function types. They are collection of ordered pairs (a, b), except we allow type of the second term b to vary according to the first term a. Naturally we call these *dependent pairs*.

To define a $\sum$ type with left domain the type A, again we first require a dependent type $B : A \to \mathcal{U}$. A dependent pair is introduced by pairing some $a : A$ with a term b of $B(a)$. Again, the special case for which B is a constant family just reduces the $\sum$ type to the usual $A \times B$, and this special case may be taken as the definition of the usual product type.

Table 3.7: Rules for Σ types

Rule ...	Effectively states ...
Formation	$\sum_{(x:A)} B(x) : \mathcal{U}$ when $A : \mathcal{U}$ and $B : A \to \mathcal{U}$
Introduction	If $a : A$ and $b : B(a)$, then $(a, b) : \sum_{(x:A)} B(x)$
Elimination	Given a family $C : \prod_{(x:A)} \prod_{(b:B(x))} \mathcal{U}$ and assuming $\Phi : C(x, y)$ whenever $x : A$ and $y : B(x)$, there is a function from the sigma type with codomain family C.
Computation	The function is evaluated by substitution

Notice that we have given the dependent elimination rule for $\sum$ types, for full generality. This rule is an immense generalization of earlier non-dependent elimination rule for basic product types. Here, we allow the codomain to depend on *both* components a and b of the ordered pairs, and moreover the type of b may depend on A.

Dependent pairs are useful for representing mathematical structures like groups. For simplicity, let us examine how to define a simple structure called a magma. In algebra, a magma is a set M equipped with a binary operation $M \times M \to M$. For us, it is a type M paired with a function $m : M \to M \to M$, in other words an ordered pair with the first component M and second component the function m. Since the type of m depends on M — magmas with different underlying sets have

operations of different types — we need sigma types to present (M, m) as a pair. The type of all magmas can thus be given as

$$\sum_{M:\mathcal{U}} M \to M \to M.$$

$\sum$ types, as their notation suggests, have a connection to summation, even while their definition seems more like that of a (binary) Cartesian product. A $\sum$ type is precisely the indexed disjoint union of the types $B(x)$ as x varies over the type A. They are effectively a "disjoint union over A," where the left component of a pair keeps track of the type of the right component.

3.4 Propositions as Types

So far we have described the universe of type theory as being something like a universe of sets, with the notable difference that the sets are disjoint and each set is of a specific nature. Give some basic types, like one for the natural numbers, there is a method for forming function types, Cartesian products, and disjoint unions. We have defined dependent types, the type-theoretic analogue to indexed families of sets. We have even introduced two fairly exotic concepts, the $\prod$ and $\sum$ types, which generalize functions and products while also acting like indexed products and coproducts respectively. It may even be somewhat plausible at this point that we can recover some aspects of regular mathematics, since we have provided a possible type-theoretic definition of a common algebraic structure. The biggest thing we are missing at this point is logic. What is a "proof" in type theory?

There are actually two things which may be called "proofs" here, which can lead to some confusion — one which we have yet to formally explain, and one which can already be understood. We have been defining type theory by giving a collection of rules establishing which sorts of judgments are permitted under which conditions. This gives us one notion of proof, which we call a *derivation*. One can "prove" that a given judgment is permitted by explicitly appealing to the rules we have given

— beginning with judgments taken for granted, inferring a series of judgments that ultimately conclude with the judgment being asserted as permissible. For illustrative purposes, here is a formal derivation of the judgment $\lambda(n : \mathbb{N}).\text{succ}(n) : \mathbb{N} \to \mathbb{N}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \text{ ctx}}\ \textsc{ctx-emp}}{\cdot \vdash \mathbb{N} : \mathcal{U}_0}\ \mathbb{N}\ \text{FORM}}{n : \mathbb{N}\ \text{ctx}}\ \textsc{ctx ext}}{n : \mathbb{N} \vdash n : \mathbb{N}}\ \textsc{Vble}}{n : \mathbb{N} \vdash \text{succ}(n) : \mathbb{N}}\ \mathbb{N}\ \text{INTRO}}{\cdot \vdash \lambda(n : \mathbb{N}).\text{succ}(n) : \mathbb{N} \to \mathbb{N}}\ \Pi\ \text{INTRO}$$

Since we have not given a fully formalized type theory, we are not in a position to understand entirely what each step on the derivation tree represents. The intent, though, is clear. In this case, we begin with no initial judgments. Below each line is the judgment being inferred from the previous one. On the side of each horizontal line is the rule being appealed to which justifies the judgment. In this sense, the derivation is a "proof" that the final judgment is permissible. Because of the computational character of type theory, we could even describe this derivation to a proof checker to verify that we have followed all of the rules.

A derivation is similar to the notion of a formal proof in ZFC — not an object inside the theory but external to it, used to establish truths about the objects in the theory. But this notion of proof is distinct from the internal notion of a proof-object, a term of a type representing a proposition. These are the terms which we have previously said may be called *witnesses*, although we slightly disregard this terminology below and refer to them just as proofs. These objects are not quite familiar to classical mathematicians — while we will use them in a familiar manner, they represent a notion of "proof" which is internal to our system of mathematics. Ordinarily, proofs reside in the *meta-theory*. To illustrate the meaning of this phrase, consider for instance why one could not apply a function to a proof in a classical setting. Whereas any internal mathematical objects — numbers, groups, functions,

sets, spaces, etc. — are subject to general mathematical operations, proofs themselves are ordinarily not. Proofs are generally a tool used to *examine* mathematics, existing at a fundamentally different level than the objects being examined.

So far we have mostly developed the meta-theory for type theory, describing the kinds of rules governing which judgments are valid. Associated to the meta-theory is the more familiar notion of "proof," which we called a derivation, exemplified above — a list of consecutive judgments, culminating in a judgment whose validity we wish to establish, validated through its explicit appeal to the meta-theoretical rules of valid judgments. The proof-objects we now describe are — much like numbers, groups, and functions — objects to be constructed, collected, evaluated, and otherwise subject to general mathematical operations, in accordance with the rules specified in the meta-theory.

In the following, we explain how to incorporate logic into type theory through this new, internal notion of proof. The basic idea, as described earlier, is to represent propositions as the type of their proofs. The fascinating observation here is that we have done most of the work already — the type-forming operations described above actually provide for us something reminiscent of first-order logic, including a way to represent quantification and predicates. All we have to provide below is the interpretation which makes this insight clear. With a method for describing propositions and proofs in place, we will then extend type theory with an internal notion of equality. This rather subtle conception of equality will lead us directly into homotopy theory and higher category theory, which will prepare the reader to consult the technical literature for homotopy type theory and conclude our treatment.

The above notions may be interpreted in this manner:

This correspondence is based on intuitionistic rather than classical logic. To explain how it works, we will visit each logical operation and explain its constructive (i.e. proof-relevant) interpretation. For each operation, we define a "new" type forming operation and then show that it is equivalent to one already defined. To match the order in which the type forming operations were introduced above, we

Table 3.8: Logical interpretation of types

Type Theory	Logical Interpretation
$A : \mathcal{U}$	A is a proposition
$a : A$	a is a proof of A
$A \to B$	The proposition "A implies B"
$A \times B$	The proposition "A and B"
$A + B$	The proposition "A or B"
$A \to \mathbf{0}$	The proposition "A is false"

begin with constructive implication, which corresponds to function types. For just this section, as we describe how to define a system of constructive logic using type theory, let us take "type" and "proposition" as synonymous, and so with "proof" and "term." Recall the cornerstone of the intuitionistic notion of a proposition — we explain the meaning of a proposition by explaining what one must construct in order to prove it.

Implication

We can define a type forming operation $\Longrightarrow$ to build "implication types", with the type $A \Longrightarrow B$ representing the proposition that A implies B. Obviously, one speaks of implication $A \Longrightarrow B$ only in the context where A and B are known to be propositions, so this will be the implication formation rule.

Classically, to say that $A \Longrightarrow B$ is to say that from the truth of A one may infer the truth of B. Since the constructive notion of truth corresponds to inhabitation of the type representing some proposition, we constructively translate "A implies B" as "Given a proof of A, one can produce a proof of B." Therefore the terms (proofs) of $A \Longrightarrow B$ are demonstrations that an arbitrary proof of A yields a proof of B. But we also confine ourselves to operations which have some sort of computational interpretation — to say one "can" do something means one has an *algorithm* for doing it. Thus altogether, a proof of $A \Longrightarrow B$ is an exact computational procedure which accepts an arbitrary proof of A and returns a proof of B.

How do we use a proof of $A \implies B$? Clearly, if we have a proof f of $A \implies B$ and a proof a of A, then we can apply f to a and say that $f(a)$ is a proof of B. Which proof of B? Obviously the one returned by the method defining f.

Implication is, of course, exactly the notion of a function type $A \to B$. After some reflection, the reader will conclude that the formation, introduction, elimination, and computation rules for $A \implies B$ are exactly those of $A \to B$. For just this type, we show the two rule-tables for the types side-by-side, so that the reader may see clearly how similar these notions are. For functions, $\mathcal{U}$ is thought of as the universe of all types. For implication, it is thought of as the universe of all propositions.

Table 3.9: Function types versus implication types

(Functions)	Effectively states ...
Formation	$A \to B : \mathcal{U}$ when $A : \mathcal{U}$ and $B : \mathcal{U}$
Introduction	If $x : A$ implies $\Phi : B$, then there is a function called $\lambda(x : A).\Phi : A \to B$
Elimination	If f is a term of $A \to B$ and $a : A$, then $f(a) : B$
Computation	If $f \equiv \lambda(x : A).\Phi$, then $f(a)$ is the result of substituting a into Φ.

(Implication)	Effectively states ...
Formation	$A \implies B : \mathcal{U}$ when $A : \mathcal{U}$ and $B : \mathcal{U}$
Introduction	If Φ is a proof of B, making use of an unspecified proof x of A, then $\lambda(x : A).\Phi : A \implies B$
Elimination	If f proves $A \implies B$ and a proves A, then $f(a)$ proves B
Computation	If $f \equiv \lambda(x : A).\Phi$, then $f(a)$ is the result of substituting a into Φ.

Conjunction

We define a new type forming operation representing conjunction, forming $A \,\&\, B$ to represent the proposition "A and B." Obviously our formation rule is that one speaks of conjunction only in the context where A and B are known to be propositions.

Constructively, $A \,\&\, B$ is proved by proving A and proving B, individually. Thus our introduction rule is that (a, b) is a proof of $A \,\&\, B$ when a proves A and b proves B.

What can one "do" with a proof of $A \,\&\, B$? In other words, what sorts of implications does a proof of the proposition have? Constructively, since one proves $A \,\&\, B$ only by proving A and B individually, then for any proof of the conjunction we may "break it apart" into its two components. Therefore we can prove from $A \,\&\, B$ whatever can be proved assuming one has individual proofs of both. Or in other words, $A \,\&\, B$ implies C when one has a method of proving C in the context that one has a proof of A and a proof of B.

It should be obvious that $A\&B$ is just $A \times B$ — the rules correspond exactly.

Note that when we were treating type theory as something like a constructive set theory, the elimination rule for a type P was being used to explain how one defines functions with domain P. In the present logical context, an elimination rule for P explains what kinds of implications P has. But we have already seen that implication corresponds exactly to function types!

Disjunction

We define a type forming operation of disjunction, forming the proposition $A \vee B$, where A and B are propositions, to represent the proposition "A or B."

What is a proof of $A \vee B$? Constructively, it is a proof either of A or B. This is subtly but significantly different from the classical definition, where a proof of $A \vee B$ is a proof that *one* of the two disjuncts A or B holds, without it necessarily being clear which. We return the topic below. For now, the introduction rule of $A \vee B$ is such that any proof of A, and any proof of B, gives rise to a proof of $A \vee B$.

What can one do with a proof of $A \vee B$? We know an implication $A \vee B \implies C$ is a method f for eliminating a proof p of the disjunction and producing a proof $f(p)$ of C. The method must be general, in other words it must work for any arbitrary p. The operational principle here is *case analysis*. This means there are two things which must be given before we may say $A \vee B$ implies C, since there are two cases to

consider: the case that p of $A \vee B$ was introduced by proving A, and the case where p came from a proof of B. But since our method must be universally applicable to any p, to break f into two cases is implicitly assuming that, given a general p, we can somehow know which proposition A or B is proved by it. But this is justified in constructive logic since we would never have concluded $p : A \vee B$ in the first place unless we knew this information. All proofs of disjunction must come attached to one of A or B.

The reader should realize that $A \vee B$ is just $A + B$, as the disjunction is just the disjoint union of all proofs of A and all proofs of B. In both cases, case analysis permits the introduction of an operation on terms by supplying two separate methods, since we can take even any term and determine which "half" of the disjoint union it corresponds to.

Falsity

Constructively, "A is false" is taken to mean that A is contradictory. That is, a proof of A's truth can be used to derive a contradiction, particularly a term of the empty type $\mathbf{0}$. Therefore, to know that A is false is to have a method which coverts a proof of A into a term of 0 — a function of type $A \to \mathbf{0}$. We may write $A \to \mathbf{0}$ as simply $\neg A$.

The reader might wonder how we could form a term of $A \to \mathbf{0}$, since $\mathbf{0}$ has no introduction rules. This represents a slight misunderstanding which can be rectified by a simple example. Indeed, we should *not* be able to construct a term of $\mathbf{0}$. What we are doing is capturing the *meaning* of the judgment that A is false. In this case, our definition is such that *if* A is false, and if A is also true, then one can derive a contradiction. Our definition captures exactly that, since we *can* construct a term of the type $(A \,\&\, \neg A) \to \mathbf{0}$. A somewhat verbose construction goes like this:

- Suppose $A \,\&\, \neg A$, i.e. suppose $p : A \times (A \to \mathbf{0})$.
- By induction on ordered pairs, we can split p into its components, forming the judgments $\pi_1(p) : A$ and $\pi_2(p) : A \to \mathbf{0}$.

- By function application, $\pi_2(p)(\pi_1(p)) : \mathbf{0}$.
- By induction on ordered pairs, there is a function of type $(A \;\&\; \neg A) \to \mathbf{0}$.
- In other words, for any A it is false that A and $\neg A$.

Notice that we did not *actually* construct a term of $\mathbf{0}$. When we said $\pi_2(p)(\pi_1(p)) : \mathbf{0}$, we were really looking at a hypothetical judgment — there is an implicit context that $p : A \times (A \to \mathbf{0})$. But the hypothetical judgment, while a valid judgment, is not the same thing as forming a term of $\mathbf{0}$. Instead, the informal derivation implicitly constructs an unnamed term of type $(A \;\&\; \neg A) \to \mathbf{0}$, which is to say of type $(A \times [A \to \mathbf{0}]) \to \mathbf{0}$, by demonstrating how to produce a term of $\mathbf{0}$ in a certain *context*. If we believe that the rules of type theory are consistent, then what we take away from this construction is that we will never find terms of both A and its negation, rather than concluding that there is a term of $\mathbf{0}$.

Now, the logic we have developed is only *propositional*. First-order logic is more expressive than classical propositional logic because it permits the notion of propositional functions (predicates). Recall that a predicate of valence one is an expression $P(x)$ which becomes a proposition after one supplies a specific object to be substituted for the variable x — that is, a predicate is a family of propositions. In addition to substituting specific objects into a predicate and forming a proposition that way, we can also create a proposition from a predicate by quantification — making a statement about the entire family of propositions at once. This gives rise to such propositions as "For all x, $P(x)$" and "There exists some x such that $P(x)$." We build up predicate logic in type theory with this interpretation:

Table 3.10: Logical interpretation of dependent type formers

Judgment	Logical Interpretation
$B : A \to \mathcal{U}$	B is a predicate on the objects of A
$\prod_{x:A} B(x)$	$\forall x \in A$, $B(x)$ is true
$\sum_{x:A} B(x)$	$\exists x \in A$ such that $B(x)$ is true

Remark. *The type A is being used somewhat flexibly here. More so than a proposition, A is being used as a type in the constructive set-theoretic sense described earlier. In fact it is being treated as a type in the classic sense, the range of significance of a propositional function. Let us here call A a "set" and write $x : A$ as $x \in A$, to emphasize this interpretation of A. We will return to topic below by answering the question "Which types are propositions?" The $\prod$ and $\sum$ types, for their part, are being treated as propositions whose terms are its proofs.*

Predicates

If B is a dependent type, in other words a function of type $A \to \mathcal{U}$, then B associates to each term a of A a type $B(a) : \mathcal{U}$. But types are propositions, so B assigns to each $a : A$ a corresponding proposition. Thus dependent types are predicates.

For all

We could define a new type forming operation $\forall$. The formation rule is that $\forall x \in A\, B(x)$ is a proposition when A is a (set-theoretic) type and $B : A \to \mathcal{U}$. A proof of $\forall x \in A\, B(x)$ is a proof of $B(x)$ in the context that x is some general term of A. It is thus a kind of implication, namely that $x \in A$ implies $B(x)$. Or we could say a proof is a method accepting input x of type A and returning output a proof of $B(x)$. The elimination and computation rules are the obvious ones of function application and evaluation. It should be clear that $\forall x : A\, B(x)$ is just the type $\prod_{x:A} B(x)$.

There exists

We could define a type forming operation $\exists$, the formation rule being that $\exists x \in A\ B(x)$ is a type when A is a set and B is a predicate on A. Constructively, existence is proved by explic*it* demonstration, so a proof of $\exists x \in A\, B(x)$ ought to contain information about *which* such x exists. Moreover, since $B(x)$ is only true if we offer a proof of $B(x)$, a proof of $\exists x \in A\, B(x)$ should contain this information too. Therefore a term of $\sum_{x:A} B(x)$ is an ordered pair $(a, b(a))$ where $a \in A$ and $b(a) : B(a)$. Evidently, $\exists x \in A\, B(x)$ is just $\sum_{x:A} B(x)$.

3.5 Proof-Relevant Logic

In general, the universe of type theory contains types like $\mathbb{N}$ which are rather set-like. But in the above, we were interpreting the type forming operations as being applied to types representing propositions. Which are which? When is a term interpreted as a proof, and when is a term more like an element? Actually, the answer is more complicated — usually, both interpretations are valid at the same time to some extent. Any type A, for example, can be interpreted as the proposition that A is non-empty, i.e. that there exists some term of type A. Proofs of such a proposition are precisely terms of A. On the other hand, a type being thought of as a proposition is still set-like, since a proposition is like the set of its proofs. In practice, we emphasize whichever interpretation is most useful. Ultimately, all types are on equal ground and follow the rules of types, with the interpretations assisting in our analysis of them.

To see the overlap between the logical and set-theoretic interpretations, we recall the induction principle of $\mathbb{N}$, specifying how to define (dependent) functions on $\mathbb{N}$ by recursion.

$$\mathrm{ind}_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}} C(0) \to \left[\prod_{n:\mathbb{N}} C(n) \to\ C(\mathrm{succ}(n))\right] \to \prod_{n:\mathbb{N}} C(n)$$

Under a logical reading, with $\prod$ as $\forall$, $\to$ as $\implies$, and C as a predicate on $\mathbb{N}$, then we can interpret $\mathrm{ind}_{\mathbb{N}}$ as a proof of the proposition that, for any predicate P over the natural numbers,

$$[P(0) \text{ and } (\forall n \in \mathbb{N}\,[P(n) \implies P(n+1)])] \implies \forall n \in \mathbb{N}\, P(n)$$

Remark. *To be more suggestive, we have made slight adjustments between the logical interpretation and the actual type, translating the first $\to$ as "and" while adjusting the scope of the pi-types. It should be clear why a direction translation works as well.*

Here we see the motivation for calling dependent elimination the "induction principle" for a type, since in the case of $\mathbb{N}$ we have just recovered the principle of mathematical induction. The logical reading of the inductor for $\mathbb{N}$ says to prove something true for all n, we simply prove it true for 0 and describe how to convert a proof of $P(n)$ into a proof of $P(n+1)$.

In general, the logic represented by the interpretation we have given is not the usual logic of mathematics. For one thing, the logic is *proof-relevant*, meaning roughly that we always make explicit reference to the proof used to establish a proposition. It is also different in an immediately striking sense. For instance, disjunction is a particularly simple but profound example of where our logic may be unusual to mathematicians.

We have said that $A + B$ is proved constructively by demonstrating that A is true or B is true. This is a very strong interpretation of disjunction and thus it has different behavior than in classical logic. There, it is sufficient to show that either A or B must be true, even if it is not clear which one. For instance, consider the following classic proof of the proposition P that there exists a rational number which can be written as an irrational number raised to an irrational exponent. Call such numbers "peculiar."

- Let $x = \sqrt{2}$.
- It is well known that x is irrational. Let $y = x^x$.
- Let $z = y^x = 2$, a rational number.
- y is rational iff it is peculiar.
- If y is not rational, then z is peculiar.
- Therefore P is true, since one of y or z is a peculiar number.

With our logic, this proof does not go through. The problem is with the disjunction $Y \vee Z$ that either y or z is peculiar, which was proved without proving either

of those propositions individually. With classical logic, a disjunction takes on a life of its own, apart from the individual disjuncts, so to speak.

The real source of the problem is the earlier disjunction that y is either rational or not. While that proposition appears to be true by definition, it invokes the *principle of excluded middle* (PEM), the logical notion that any proposition is true or false, necessarily one and not both. With PEM as a rule of logic, the proposition $A \vee B$ is equivalent to $(\neg A) \implies B$, which may be proved without proving A or B on their own. Our proof above makes use of this equivalence.

The principle of excluded middle is not implied by our interpretation of propositions as types whose terms are proofs. Recall that our mathematics is rooted in computation and defines all other ideas on top of it, using type theory to classify different kinds of computations. For the principle of excluded middle to be true is to have a term of the disjunction

$$A + [A \to \mathbf{0}]$$

for each and every type A. This would give us a method for, given any type A whatsoever, either proving A by finding a term of it or disproving A by proving it to be contradictory. In effect, one would have a computational method which can prove or disprove everything in mathematics. Of course, there is not such a thing to be found.

Although we are presently concerned with mathematics and not philosophy, it may be helpful to address a few points here. The intuitionistic approach is not necessarily associated with a rejection of PEM. Instead, the question is recategorized as a philosophical principle and not which can be proved with logic (since logic is an application of computation). Perhaps it is true in some sense that every proposition is either true or false, but this is an ontological claim, rather than one which can be demonstrated mathematically (computationally).

Our type theory does not reject excluded middle. In fact, type theory refutes the refutability of PEM, because $\neg\neg[A + A \to \mathbf{0}]$ is an inhabited type for all A. While a

satisfactory proof of PEM cannot be given, the imposition of PEM as an additional rule is consistent with type theory. The downside is that one then sacrifices the nicer meta-theoretical properties of type theory, since those nice properties depend on a restriction to computation. However, PEM is perhaps not as essential for mathematicians as it might seem — with the proper rephrasing of arguments, one can often avoid invoking the principle. Additionally, PEM is true for some kinds of types, just not true in general.

There is one more mathematically relevant argument for not needlessly incorporating PEM, which is of major interest to many mathematicians. Just as with group theory, number theory, or even set theory, one can speak of different "models" of type theory. Especially compelling is the correspondence between different kinds of categories (in category theory) and different variants of type theory, which is a topic of significant current interest. Imposing the principle of excluded middle restricts the theory to a narrow range of models. This situation is analogous to the observation that imposing more than the general axiom for rings restrict one's models to less general structures like integral domains and fields. The correspondence between type theories and category theory is a topic which we will only briefly visit later.

The lack of a PEM is not the only unfamiliar behavior mathematicians may find in type theory. For instance, consider the axiom of choice (AC), the somewhat infamous axiom extending ZF set theory to ZFC.

$$\forall X\,[\emptyset \notin X \implies \exists f : X \to \cup X(\forall A \in X\,(f(A) \in A))]$$

Recalling that all sets are sets of sets, an English translation may be given thusly:

> For all sets X, if all elements of X are non-empty, there is a function mapping X to the set of elements of elements of X, with the property that f maps each element A of X to an element of A.

The axiom of choice can equivalently be stated as the axiom that a Cartesian product, possibly infinite, of non-empty sets is non-empty. The axiom cannot be

proved from the other (ZF) axioms of set theory. The basic problem is that sets are non-empty precisely when they have elements, but how does one find an element of a general infinite Cartesian product? The finite case can be proved inductively, but proving the infinite case requires making an infinite number of "choices," one for each component, and there is no obvious way to justify such a construction without implicitly invoking the axiom of choice. In fact, combined results of Gödel and Cohen prove that it cannot be done — some models of ZF set theory do not satisfy the axiom of choice, while others do, proving that those axioms cannot be used to prove or refute AC.[17]

Instead of set membership $\in$, which recall is a just a binary predicate on the universe V of material sets, we could more generally write the axiom of choice for a binary predicate R. Our general axiom, for a choice of X, implies the proposition below.

$$\forall x \in X \; \exists y \in B \text{ such that } R(y,x) \implies \exists f : A \to B \; (\forall x \in X \; R(f(x),x))$$

Remark. *We restrict our consideration to a choice function on a particular but arbitrary set X, only for convenience.*

For type theory, we can represent a relation R as a a function $X \to Y \to \mathcal{U}$, a function accepting two inputs x, y and returning the proposition that y is related to x. According to our interpretation of logic, the above formula is then translated most literally as the type:

$$\left[\prod_{x:X} \sum_{y:Y} R(y,x)\right] \to \sum_{f:X\to Y} \prod_{x:X} R(f(x),x)$$

The terms of this type are analogous to choice functions on X. The proposition represented by the type may be read, "If we know each element x of X is related to

[17] Technically these results assume ZF is consistent. If it is not, then *all* propositions are theorems in the system.

at least one element y of Y, then there is a function $f : X \to Y$ such that $f(x)$ is related to x." This type, however, is inhabited by the following term:

$$\lambda \left(g : \prod_{x:X} \sum_{y:Y} R(y,x) \right) . [\lambda x.\mathrm{p}_1(g(x)) \ , \ \lambda x.\mathrm{p}_2(g(x))]$$

This term is a proof that the axiom of choice, or at least the most direct translation of it, is a theorem in type theory. Of course, presumably constructive mathematics would not ordinarily permit such a thing. This strange behavior stems from our strict interpretation of "existence." In set theory, the premise of the axiom of choice (restricted to a set X) is that each element of X is non-empty. The conclusion is that there is a choice function on X, i.e. a function sending each element of X to one of its own elements. Now with proof-relevant logic, the premise translated as a term of a proposition, in this case a function on X sending each x to an ordered pair: The first component is a term y in Y, and the second is a proof that y is related to x. Such a term is necessary to witness the fact that each term of X is "non-empty," but we can see that, in effect, the hypothesis itself is a kind of choice function.

There is one very important idea which have deliberately not addressed yet: our logical interpretation so far has made no mention of equality. More specifically, we have not described a type representing the proposition of equality between objects. Currently, we only have judgmental equality, which is meta-theoretical and hence is quite distinct from the sort of equality used most often in mathematics. To be perfectly clear, we will explain again why judgmental equality is not useful for many purposes.

Because (judgmental) equality is a judgment, it is completely distinct from a proposition. We could not, for instance, prove nor disprove such an equality in the sense of propositions-as-types. Recall that a proposition, whether considered as "external" object to the theory (as with first-order logic), or as an internal object within the theory (as with type theory), is a statement we consider to have a truth

value. Usually equality is taken as a proposition, and certainly so in first-order logic. For instance, we can formulate the proposition $x = y$ and then appeal to the rules of predicate logic (maybe in conjunction with axioms like those of set theory) to find a proof or disproof of it. If we find a proof, we form the judgment that "The proposition $x = y$ is true (according to the rules)."

Judgmental equality is a judgment in itself. We use this notion to keep track of which expressions, by our own intention, represent the same object, hence the common name "definitional equality." For instance, $f(n)$ means the result of inputing the number n into the function f. 2 by definition means the resulting of applying the successor operation to $0 : \mathbb{N}$, twice. We introduced the symbol + by stating that $m + n$ by definition means $\mathrm{add}(m, n)$. By its own computation rules, $\mathrm{add}(m, n)$ means the resulting of inputting m and n into the program recursively calculating the output of addition. Generally speaking, two expressions are judgmentally equal because there is a sense in which we mean for them to be equal. Naturally, judgmentally equal expressions may be freely substituted for one another in all contexts.

We could *not* use judgmental equality to ask, for instance, whether $f(0) = f(1)$ for some function f. The two expressions should probably not be equal judgmentally because they are not intended to be equal as expressions. Judgmental equality is only meant to represent the idea that $f(0)$ by definition means inputting 0 into f. The kind of equality represented in the question above is propositional — the sort of thing to be proved or refuted, or hypothesized. Another application of propositional equality would be to define a notion of injectivity, defining "f is injective" to mean that $f(x) = f(y) \implies x = y$ is a true proposition. Or, we may want to examine (in type-theoretic terms) the transitivity of equality, perhaps proving a basic proposition like $(a = b)\ \&\ (b = c) \implies (a = c)$. In keeping with propositions-as-types, we therefore introduce a new kind of type, called an identity type, whose terms are proofs that two objects are equal.

3.6 Identity Types

Formation

Given a type A and two terms $a : A$ and $b : B$, there is a type called $Id_A(a, b)$, which we write as $a =_A b$. Such a type will represent the proposition that a and b are equal terms of A (not necessarily because of their symbolic forms).

Introduction

For any $a : A$, there is a term $\text{refl}_a : a =_A a$. This term is called reflexivity, and it represents the proof that a is equal to a in a canonical sense. This is the only introduction rule we give, but it emphatically does *not* imply that reflexivity is the only possible proof of equality between two terms. It is for this reason that we have been careful to emphasize that type theory should be taken at face value — saying that reflexivity is a proof of equality, even when no other introduction rules are given, does not imply that there are no other proofs of equality. So far, we have not even shown that only the only terms of $\mathbb{N}$ are what can be obtained by 0 and the successor operation.

Elimination and Computation

What is elimination? The most obvious approach is to simply infer judgmental equality. We could impose a rule that, given a proof p that a and b propositionally equal, that a and b are then judgmentally equal, like so:

$$\frac{p : a =_A b}{a \equiv b : A}$$

There is a variant of Martin-Löf's type theory, namely the *extensional* variant, which does impose such a rule. Moreover, the theory is useful for mathematics. However, the theory turns out to have poor meta-theoretical properties. In particular, type-checking becomes undecidable. This means that, with a such a rule in place, we could not in general determine mechanically whether the assertion $p : P$ is correct.

To maintain the desirable properties of type theory, we impose a different elimination rule for identity types. Our rule is the following:

Given two functions

$$D : \prod_{x,y:A} \prod_{x=_A y} \mathcal{U}$$

$$d : \prod_{x:A} D(x, x, \mathrm{refl}_x)$$

Then there is a function

$$\mathrm{ind}_A(D, d) : \prod_{x,y:A} \prod_{x=_A y} D(x, y, p)$$

And the computation rule is that

$$\mathrm{ind}_A(D, d)(x, x, \mathrm{refl}_x) \equiv d(a)$$

This elimination rule is tremendously subtle. Regarding terms of $\mathcal{U}$ as propositions, we see that D is a dependent function of several variables and returns a proposition. Effectively, D associates any proof p of equality between any x and y of A to some (not necessarily true) proposition. D is a predicate depending on proofs of equality between terms of A.

d is a function accepting a term x in A and returning a proof of the proposition D associates to reflexivity on x. d is a proof of D in the special case that x and y are identical (judgmentally) and p is the canonical equality between them.

The conclusion of the elimination rule is this: there is a function accepting proofs p of equality between arbitrary terms of A (much like D) and returning proofs of the proposition that D associates to p. The computation rule is just bookkeeping and can be ignored for right now.

We offer a logical interpretation, albeit a strange one that does not quite translate into first-order logic. In particular, the logic here is distinctly proof-relevant, since it explicitly refers to proofs of equality. D may be read $\forall x \in A\ \forall y \in A\ [x = y \implies \Phi]$. Most of the strangeness is that Φ in general is a predicate depending on the exact

proof that $x = y$ (recall that hypothesizing $x = y$ means hypothesizing a *proof* that $x = y$, so the implication is a dependent function). d is a proof that $\forall x \in A\ \Phi[\text{refl}_x]$ where $\Phi[\text{refl}_x]$ means the proposition associated to reflexivity on x. The conclusion is that D is true in general.

What does such an interpretation mean? First, let us consider a special case of it. Above, D depends on two terms x, y in A and a proof p of equality between the terms. Suppose we ignore the proof relevance, letting D depend entirely on the terms x, y and not on p. Then the interpretation works quite well: D is a binary predicate associating all pairs x, y of propositionally equal terms of A to a proposition $D(x, y)$. d is a proof of the proposition $D(x, x)$. The conclusion is that $D(x, y)$ is true. In other words, to prove something about terms x and y which are known to be (propositionally!) equal, it suffices to consider the case where y is judgmentally equal to x. Whereas the induction principle for $\mathbb{N}$ encapsulates the notion of proof by induction, the induction principle for identity types is that equals may be substituted for equals.

As a simple demonstration, we offer a proof of symmetry: that $[a = b] \implies [b = a]$. While ordinarily, symmetry may be taken as part of the definition of equality, here we need to establish the property by constructing an actual proof. Informally, it works like so: Suppose $a = b$, so we want to prove $b = a$. By induction on equality, it suffices to prove symmetry for $a = a$. But $a = a \implies a = a$, so symmetry is true. A more formal proof might look like this:

- Let D be the function defined by $D(a, b, p) \equiv b =_A a$. D is a function mapping any terms a, b in A such that $a =_A b$ (and the accompanying *proof* that $a =_A b$!) to the proposition $b =_A a$.
- To prove symmetry, we seek a proof that for any a, b and $p : a =_A b$, the type $D(x, y, p)$ is inhabited.
- Let d be the function defined by $d(x) \equiv \text{refl}_x$. d is a term of $\prod_{x:A} D(x, x, \text{refl}_x)$, in other words a proof of $D(x, y, p)$ in the restricted case that x is judgmentally equal to y and p is reflexivity. In such a case, $D(x, y, p)$ is judgmentally equal

to the proposition that $x =_A x$, and d proves the proposition by returning the reflexivity term.

- By the elimination rule for identity types, there is a proof of $D(x, y, p)$ in the *general* case where y and p are arbitrary. We happen to call this proof $\mathrm{ind}_A(D, d)$. Specifically, this proof is a function sending an arbitrary $p : x =_A y$ to $q : y =_A a$. We could denote this q by p^{-1}.

An extremely formal proof would be a formal derivation of the typing judgment

$$\lambda x.\lambda y.\lambda p.\mathrm{ind}_A\left[(\lambda x.\lambda y.\lambda p.y =_A x), (\lambda x.\mathrm{refl}_x)\right](x, y, p) : \prod_{x,y:A} \prod_{p:x=_A y} y =_A x$$

Obviously we would not regularly use nearly such formality, just as we would not be so formal in set theory. In a computer environment, the proof assistant can expedite the process of deriving the formal judgment by guiding the user through the steps and "filling in the gaps" as much as possible.

Here is an informal proof that all terms of $\mathbb{N}$ are 0 or a successor.

- By the induction principle for $\mathbb{N}$, to define a function f on *all* natural numbers, it suffices to define $f(0)$ and to define $f(\mathrm{succ}(n))$ for a general n, assuming $f(n)$ is defined.
- Let the codomain C depend on n, defining $C(n) \equiv n =_{\mathbb{N}} n$.
- Define $f(0) \equiv \mathrm{refl}_0 : 0 =_{\mathbb{N}} 0$.
- Define $f(\mathrm{succ}(n)) \equiv \mathrm{refl}_{\mathrm{succ(n)}} : \mathrm{succ}(n) =_{\mathbb{N}} \mathrm{succ}(n)$.
- By induction, f has been defined on *all* natural numbers. f sends an arbitrary natural number to a proof of equality between it and either 0 or a successor.

With practice, reasoning in type theory becomes as straightforward as everyday reasoning in classical logic. The key idea that we must become acquainted with is proof-relevance, automatically understanding that a proposition is true precisely when one has an appropriate proof.

Now, for a fixed a, b of A, the identity type $a =_A b$ is a type like any other, and therefore *it* has an associated family of identity types: for $p, q : a =_A b$ we may

consider $p =_{a=_A b} q$, which itself has an associated family of identity types, and so on. The mysterious aspect of the elimination rule we have given is that, in general, we cannot prove that these types have non-trivial structure. We cannot even prove that all terms of $a =_A a$ are equal to reflexivity. We could try a similar approach that we used for $\mathbb{N}$, mapping each $p : a =_A a$ to a proof that p equals reflexivity. But the elimination rule for identity types does not obviously let us define such an operation — unlike the rules for $\mathbb{N}$, the induction principle here does not tell us how to build a function with domain an identity type $a =_A b$ for a fixed a and b in A. A quick glance at ind_A shows that identity elimination returns a map whose codomain depends not on a general $p : a =_A b$ for a *fixed* a and b, but a map out of A which depends on general p of a *general* $x, y : A$.

We could perhaps use the elimination rule for identity types one dimension higher. That is, we could let D be a term of type $\prod_{p,q:a=_A b} \prod_{\alpha:p=_{a=_A b} q} \mathcal{U}$ to produce a function whose domain is a fixed type $a =_A b$. But this would only let us construct proofs regarding the terms of $a =_A b$ in the context where they are already known to be equal, which does not help us.. In sum, it is not obvious how to construct a function whose domain is a fixed $a =_A b$ and which returns a proof that all $p : a =_A b$ are equal to reflexivity (whether propositionally or judgmentally). Apparently, identity types in general can have some sort of non-trivial structure. Indeed, Hoffman and Streicher proved this to be the case by providing a model type theory for which the identity types were not trivial.

We are now ready to introduce the punchline which defines homotopy type theory: Types are topological[18] spaces whose terms are points in the space, and terms of identity types are paths in the space. The elimination rule above says that all paths in a space are homotopic to constant map if we allow the endpoints to move freely. But if we *fix* the endpoints, which corresponds to examining a fixed type $a =_A b$, then of course not all paths are homotopic. That is why we cannot prove that identity types have trivial structure — they need not.

[18]This is not, strictly speaking, correct. We will be more precise below.

CHAPTER 4

HOMOTOPY THEORY AND HOMOTOPY TYPE THEORY

We begin this section with a cursory review of the basic concepts of homotopy theory, which we presume are already familiar to the reader. From here, we motivate the connection between type theory and homotopy theory, and explain that the connection is not simply a useful analogy — through abstract homotopy theory, one can see that types and topological spaces are indeed very closely connected ideas. Admittedly, this section is the most informal and brief, for a few reasons. First, there is much literature being produced with explains the ideas more thoroughly, for instance [AW09]. Second, a more thorough examination of these topics very quickly would lead us far beyond the scope of this paper, and we cannot presume that readers have sufficient backgrounds in homotopy theory and category theory.

4.1 Classical homotopy theory

Homotopy theory, not seemingly related to type theory, is the branch of mathematics that studies continuous deformations. Given two maps (continuous functions) f, g from a (topological) space X to a space Y, a *homotopy* between f and g is a "two-dimensional" map $H : [0,1] \times X \to Y$ such that $H(0,x) = f(x)$ and $H(1,x) = g(x)$. Intuitively, H is a continuous deformation of f into g. We say f and g are homotopic, written $f \sim g$. The property of being homotopic forms an equivalence relation on the collection of maps from X to Y, and as usual we denote the equivalence class of f as $[f]$.

Two topological spaces are said to be *homotopy equivalent* if there exist continuous maps $f : X \to Y$, $g : Y \to X$, such that $(g \circ f)$ is homotopic to the identity

map $\mathrm{id}_X : X \to X$ and $(f \circ g)$ is homotopic to $\mathrm{id}_Y : Y \to Y$. As the terminology indicates, this is an equivalence relation. Two homotopy equivalent spaces need not be homeomorphic, i.e. isomorphic as spaces. Instead, homotopy equivalence indicates that two spaces are "equal enough," insofar as many topological properties of one can be translated into properties of the other.

A *path* in a space X is a map $I \to X$, where I is the unit interval $[0, 1]$. A *loop* is a path such that $f(0) = f(1)$, and this value is called the basepoint of the loop. In the special case of loops, which may be thought of as a pictures of a circle in X, we call maps homotopic only if there is a homotopy between them which is constant on their common basepoint, i.e. such that $H(t, 0) = f(0) = g(0)$ for all $t \in I$.

Given any two loops f and g with a common basepoint, their concatenation $f \cdot g$ is defined as the loop which traces f and then traces g, reparameterized as to be defined on the unit interval I. Concatenation respects homotopy equivalence: $f_0 \sim g_0 \ \& \ f_1 \sim g_1 \implies [f_0 \cdot f_1] = [f_1 \cdot g_1]$

The fundamental group of X at the basepoint $x_0 \in X$ is defined as follows: the underlying set is the collection of homotopy classes of loops based at x_0. The group operation $*$ is concatenation: $[f] * [g] = [f \cdot g]$, which is well defined by the above comment. The inverse of a class $[f]$ is $[f^{-1}]$ where f^{-1} is the loop which traces f "backwards." The identity element is the constant loop at the basepoint x_0. The fundamental group is designated $\pi_1(X, x_0)$, and its algebraic structure is a primitive indication of some of the "holes" present in a space.

The fundamental group evidently satisfies the axioms of a group. It is important to note that the group laws are satisfied only when we consider equivalence classes of loops. That is, the axioms of group theory are satisfied only "up to homotopy." For instance, the axioms postulate that for all classes $[f]$ of loops based at x_0, we must have $[f] * [f^{-1}] = [\mathrm{id}_{x_0}]$. The equality holds because we consider homotopy classes, but one can see that $f \cdot f^{-1}$ is usually not literally equal to the constant map at the basepoint. Instead, the concatenation of the loops is merely homotopic

to the constant map (intuitively, the concatenated loop may be continuously shrunk to the point x_0 without "snagging" on the holes in X).

The fundamental group $\pi_1(X, x_0)$ may be generalized to the fundamental *groupoid* of X, written as simply $\pi_1(X)$. A groupoid is formally defined as a category such that all morphisms are isomorphisms. In this light, a group can be called a groupoid with one object (with morphisms the group elements). In effect, a groupoid may be thought of as a group with a partially defined composition operation. This definition would require separate identity elements such that for any element x there are elements i_s, i_t satisfying $i_\mathrm{s} \cdot x = x$ and $x \cdot i_\mathrm{t} = x$ (these identity elements need not be uniquely determined by x).

To define $\pi_1(X)$, we remove our dependence on x_0 and instead consider homotopy classes of *all* paths in X (where the homotopies now fix both endpoints rather than the one basepoint). For a class of paths $[f]$, the local identity element i_s is the class $[c_{f(0)}]$ of the constant map at the "starting point" $f(0)$ of f. Similarly, i_t is the class $[c_{f(1)}]$ of the constant map at the endpoint. Notice that, again, the groupoid laws are satisfied only because we consider the paths up to homotopy. The fundamental groupoid is only slightly more general than the fundamental group at x_0 and may be thought of as a "horizontal" generalization of it.

The fundamental groupoid can be generalized "vertically" as well, yielding the fundamental (weak) infinity groupoid $\prod_\infty(X)$ of X. Instead of satisfying the usual group(oid) laws by considering equivalence classes of paths, we weaken the usual groupoid requirements in such a way that we can consider actual paths. In terms of category theory, we let the objects be the points in X and morphisms the paths. Then we allow for a collection of "2-morphisms" and weaken the groupoid axioms to require that the paths, which we now call the 1-morphisms, satisfy the usual laws only up to 2-morphisms. In this case, the 2-morphisms are the homotopies (fixing endpoints) between paths in X. Then we can see that indeed, the usual groupoid equalities (associativity and the usual laws of inverses) are satisfied up to 2-morphisms, which is verified in precisely the same way we show that the fundamental

group (with elements as classes of loops) satisfies the strict group laws. But we do not stop with 2 morphisms — we allow for 3 morphisms and require that 2 morphisms satisfy the groupoid laws up to *them*. In this case, the 3-morphisms are the homotopies between homotopies. In general, we allow for an infinite ladder of n-morphisms (n a natural number) and impose the axioms that the groupoid laws at each level are "coherent," i.e. satisfied up to the morphisms at the next higher level.

The connection to type theory is that the elimation rule for identity types *equips each type with its own ∞-groupoid structure*. We think of terms a, b in A as "points in the space A" and terms of $a =_A b$ as "paths" in the space. Now, the usual properties of equality — reflexivity, symmetry, transitivity — correspond to groupoid laws of identities, inverses, and composition of morphisms. Specifically, fixing a type A, one can construct the following terms by appealing to the rules of identity types:

Table 4.1: Properties of propositional equality

Reflexivity	$R : \prod_{x:A} x =_A x$.
Symmetry	$S : \prod_{x:A} \prod_{y:A} (x =_A y) \to (y =_A x)$.
Transitivity	$T : \prod_{x:A} \prod_{y:A} \prod_{z:A} (x =_A y) \to (y =_A z) \to (x =_A z)$.

R is obviously the map sending x to refl_x. The symmetry term S was constructed above as an example of how to use identity elimination. Transitivity T can be constructed similarly. We write $p^{-1} : y =_A x$ for $S(x, y, p)$ where $p : x =_A y$. We write $p \cdot q : x =_A z$ to mean $T(x, y, z, p, q)$. Now, notice the parallel between the above propositions and the following notions from homotopy theory. Given a space A, we can prove the following:

In terms of categories, the analogous statements for an infinity groupoid G are thus:

Now, the term $p \cdot p^{-1}$ need not be judgmentally equal to the term $\mathrm{refl}_x : x =_A x$. Nor does $p^{-1^{-1}}$ need to be equal to p. And $p \cdot (q \cdot r)$ need not be the same as $(p \cdot q) \cdot r$. We can, however, form the following terms:

Table 4.2: Properties of paths in a space

Constant Path	Given x in A, there is a constant path at x.
Inversion	For $f : I \to A$, there is an inverse path f^{-1}.
Concatenation	For two paths f, g if $f(1) = g(0)$, there is a path $f \cdot g$ from $f(0)$ to $g(1)$.

Table 4.3: Properties of morphisms in a groupoid

Local Identity	For an object x in G, there is an identity morphism id_x .
Inverses	For each (iso)morphism f, there is an inverse morphism f^{-1} .
Composition	For morphisms f, g, if the target of f is the source of g, there is a composed morphism $g \circ f$.

Table 4.4: Properties of operations on identity types

Cancellation	$U : \prod_{x,y:A} \prod_{p:x=_A y} p \cdot p^{-1} =_{x=_A x} \mathrm{refl}_x$.
Involution	$V : \prod_{x:A,y:A} \prod_{p:x=_A y} {p^{-1}}^{-1} =_{x=_A x} p$
Associativity	$W : \prod_{x,y,z,w:A} \prod_{p:x=_A y} \prod_{q:y=_A z} \prod_{r:z=_A w} p \cdot (q \cdot r) =_{x=_A z} (p \cdot q) \cdot r$

Type theoretically, the terms above indicate that the concatenation and inversion of propositional equalities satisfy the usual laws of such operations, *up to propositional equality* at the level of equalities between equalities. In the homotopy interpretation, higher propositional equalities are higher homotopies, and the above types then correspond directly to the fact that the infinity groupoid structure on a space A is weak, i.e. the groupoid laws are satisfied only up to higher morphisms.

Thus, there is a very strong analogy between types and spaces. In fact, the analogy extends to dependent types as well: Families of types correspond to fibrations, with type-theoretic analogues of all of the usual lifting properties from homotopy theory. Such strong analogies suggest that we should explore the analogy through more formal methods. For this, we use *abstract* homotopy theory.

Table 4.5: Types, Propositions, and Spaces

Type Theory	Logic	Homotopy
$A : \mathcal{U}$	A is a proposition	A is a space
$a : A$	a is a proof of A	a is a point of A
$A \to B$	The proposition A implies B	Function space
$A \times B$	The proposition "A and B"	Product space
$A + B$	The proposition "A or B"	Coproduct
$\mathbf{0}$	False	Empty set
$\mathbf{1}$	True	Singleton
$B : A \to \mathcal{U}$	B is a predicate on A	B is a fibration over A
$\prod_{x:A} B(x)$	$\forall x \in A$, $B(x)$	Space of sections
$\sum_{x:A} B(x)$	$\exists x \in A$ such that $B(x)$	Total Space

Table 4.6: Types, Spaces, and Groupoids

Type Theory	Homotopy Theory	∞ groupoids
A type	A space	A groupoid
$a : A$	$a \in A$	a an object of A
Reflexivity	Constant loops	Identity morphisms
Symmetry	Inverses of paths	Inverse morphisms
Transitivity	Concatenation	Composition
Functions on A	Continuous functions on A	Functors from A

4.2 Abstract homotopy theory

In general, one can "do homotopy theory" in more settings than just the usual topological spaces. For instance, *simplicial sets* are a combinatorial/algebraic representation of spaces for which there is not a notion of "topology" (open sets, convergence, etc.), but still notions of continuous maps, fundamental groups, fibrations, etc. As another example, instead of considering topological spaces themselves, we can consider simply the category of all weak ∞ groupoids and still describe the usual contructions of homotopy theory. The study of homotopy theory in settings other than topological spaces is what is meant by "abstract homotopy theory."

A common device in abstract homotopy theory is that of Quillen model categories. Any category satisfying the model category axioms can be considered a representation of homotopy theory sufficient for the usual constructions. This is any bicomplete[1] category with three subcategories $\mathcal{F}$ ("fibrations"), $\mathcal{C}$ ("cofibrations"), and $\mathcal{W}$ ("weak equivalences") subject to a few conditions regarding factorizations and lifts of morphisms (see [AW09] or [War08]).

The principle result of homotopy type theory is that Quillen model categories are models of type theory. Technically there are "coherence" issues to be resolved regarding the strictness of the interpretation, but the combined results in the field permit a very strong sense in which types can be regarded as abstract spaces subject to homotopy-theoretic reasoning. Type theory itself can be considered an "internal language" for model categories, or even the "logic of homotopy theory."

For instance, we can prove the proposition in type theory that $x =_A y \implies f(x) =_B f(y)$, for any $f : A \to B$. The proof is entirely obvious from a type-theoretic perspective — by induction on equality, we can reduce to the case that $x \equiv y$, which makes the proposition trivial. From a homotopy-theoretic perspective, the result is that all functions are continuous insofar as they map paths to paths. From a categorical perspective, functions are functors mapping objects to

[1] A bicomplete category is one with all limits and co-limits

objects and morphisms to morphisms. If we generalize our considerations to dependent functions, we see that dependent functions have all of the usual properties of fibrations.

We can also define a notion of equivalence between types, corresponding to equivalence of functions and spaces. Actually, there are several approaches to defining such a notion in type theory. For functions of $A \to B$, we can define

$$(f \sim g) \equiv \prod_{x:A} f(x) =_B g(x).$$

For equivalence between spaces, one useful definition is

$$\text{isequiv}(f) \equiv \left[\sum_{g:B\to A} (f \circ g \sim \text{id}_B)\right] \times \left[\sum_{h:B\to A} (h \circ f \sim \text{id}_A)\right].$$

From here, we can define $A \simeq B$ as $\sum_{(f:A\to B)} \text{isequiv}(f)$, which behaves similarly to a conventional notion of equivalence between spaces A and B.

4.3 Univalence

Homotopy type theory is actually an extension of the type theory, incorporating new principles which are motivated by the homotopy interpretation. In particular, homotopy type theory imposes an axiom: the univalence axiom. So far we have not described what it means to have an axiom in type theory, since everything we have said so far has been in terms of rules. In type theory, an axiom is just a judgment which we take for granted, rather than derive. Type theory by itself has no axioms.

The univalence axiom was originally noted by Voevodsky to be a property of the interpretation of type theory into simplicial sets. The basic principle of the axiom is to "identify equality and equivalence" is a certain precise sense. For a universe $\mathcal{U}$, like any other type, there is a type family of equalities between its terms, the small types. That is, $\mathcal{U}$ has an identity type. There is also the type of $A \simeq B$ of *equivalences* between its terms, defined above. Now for any types $A : \mathcal{U}$, $B : \mathcal{U}$, there are types $A \simeq B$ and $A =_{\mathcal{U}} B$. As would be expected, equality is a kind of

equivalence, so there is a canonical map of the type $(A =_{\mathcal{U}} B) \to (A \simeq B)$. The univalence axiom is the judgment that this map itself is an equivalence, so that $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$ is itself an occupied type.

Through the additional imposition of the univalence axiom, one can define new types which behave like ordinary spaces in homotopy theory. Moreover, facts about these spaces can be formally represented and verified in a computer environment, especially the interactive proof assistant COQ. For instance, one can construct a verified proof that the fundamental group of S^1 is (isomorphically) $\mathbb{Z}$ (see [LS13]). The univalent foundations program is the effort led by Voevodsky to formalize homotopy theory and other mathematics in a computer environment in this manner, associated with a special year held from 2012–2013 at the Institute for Advanced Study. So far, researchers have been able to formalize and verify a variety of proofs along the main lines of classical homotopy theory. Such proofs include the Freudenthal suspension theorem, van Kampen's theorem, and the calculations of the homotopy groups of spheres. Moreover, much as with homotopy theory, homotopy type theory proves useful as a foundation for category theory, circumventing the intrinsic awkwardness of a set-theoretic formalization of it. Results in this direction have included the development of most definitions and theorems of general category theory, including Yoneda's lemma.

CHAPTER 5

CLOSING REMARKS

We should emphasize in closing that homotopy type theory need not be taken as a competitor to set theory, necessarily. Most of everyday set theory can be incorporated directly into the theory (with some machinery in place to support this), and homotopy type theory can be modeled in ZFC, so the systems are broadly compatible in a certain sense. However, the computational advantages and richness of the system, particularly the intrinsic support for homotopy theory and category theory, would seem to indicate that it should be not be relegated as simply a small convenience over classical foundations.

Due to the scope of this paper, much of the practical uses of the theory have not been addressed. While we have indicated that type theory can support general mathematics, we have not developed such ideas at length here. An interested reader would find these topics covered in the free resource [Uni13], effectively a tutorial on how homotopy type theory can contribute to exactly such topics. There, a reader may find the development of homotopy theory, category theory, a working formulation of set theory, and the construction of the real numbers. For a more detailed examination of the deeper underlying theory of homotopy type theory — the topic of Quillen model categories and the higher category theory of types — suggested resources are [War08] and the nLab[1]. Other useful resources may be found through the nLab.

Below, we include a few appendices which may be of general interest. The first documents in the axioms of ZFC, with some commentary. The second documents some examples of formal type theory, which may be useful to appreciate some of

[1]The nLab is an online community project concerned with higher category theory and its applications to mathematics and physics. `http://ncatlab.org`

its nuances. The third is a trivial but entertaining proof of a basic logical identity, written as a script which many be executed in the proof assistant COQ. This computer system is being used by the univalent foundations project, based on a system similar to the type theory we have explored here.

.1 Axioms of ZFC

This appendix simply the axioms of ZFC, with some light commentary about what each axiom represents. They are written with some use of ordinary mathematical notion, rather than a purely formal language, for convenience. Two of the "axioms" are actually axiom *schemas*. That is, they are ways of inductively defining many axioms at once. While nine statements are given below in total, the two schemas actually mean that there are infinitely many axioms of ZFC. For this to be rigorous, we have to ensure that there is a mechanical procedure for examining a given formula and determining whether it is an axiom. This is the case with ZFC.

Extensionality

$$\forall x \forall y [\forall z (z \in x \iff z \in y) \iff x = y]$$

Sets are determined only by their elements.

Pairing

$$\forall x \forall y [\exists z (z \in x \land z \in y]$$

For sets x and y, there is a set containing both as elements.

Union

$$\forall \mathcal{F} \, \exists A \, \forall Y \, \forall x [(x \in Y \land Y \in \mathcal{F}) \Rightarrow x \in A]$$

Unions exist. That is, for a family $\mathcal{F}$ of sets, there is a set whose elements are the elements of the sets in $\mathcal{F}$.

Power Set

$$\forall x \exists y \forall z [z \subseteq x \Rightarrow z \in y]$$

Power sets exist. That is, for any set X, there is a set who elements are the subsets of X.

Infinity

$$\exists X\,[\emptyset \in X \wedge \forall y(y \in X \implies S(y) \in X)]$$

There exists a set with $\emptyset$ as an element and closed under the successor operation (under a fair definition of such a function, like the ones given earlier).

Regularity

$$\forall x[\exists a(a \in x) \Rightarrow \exists y(y \in x \wedge \neg\exists z(z \in y \wedge z \in x))]$$

There is no infinite descending sequence of sets (ordered by $\in$). In particular, there is no set which is an element of itself.

Schema of Restricted Comprehension

$$\forall z\,\forall w_1 \forall w_2 \ldots \forall w_n\,\exists y \forall x[x \in y \Leftrightarrow (x \in z \wedge \phi(x, w_1, \ldots, w_n)]$$

This is one an example of an axiom schema, which can be considered shorthand notation for infinitely many axioms. We allow for n to be any natural number (in the usual sense, not a formal set-theoretic sense). For a given n, $\phi(x, w_1, \ldots, w_n)$ represents an arbitrary formula of first-order logic whose free variables can be any among x, $w_1, \ldots w_n$. For a given choice of w_1 through w_n, $\phi(x, w_1, \ldots, w_n)$ is a predicate on the set z. The particular instance of the axiom schema then says that there is a set whose elements are those x in z such that the predicate is true (often called the extension of the predicate). This axiom schema justifies the usual set builder notation, since it allows for the definition of subsets by specifying a property its elements satisfy.

Schema of Replacement

$$\forall A \forall w_1 \forall w_2 \ldots \forall w_n \big[\forall x(x \in A \Rightarrow \exists! y\,\phi) \Rightarrow \exists B\ \forall x\big(x \in A \Rightarrow \exists y(y \in B \wedge \phi)\big)\big]$$

Replacement is another axiom schema, and a rather subtle one at that. Effectively, the schema says that if f is a "function" (not in a strict set-theoretic sense, but a more general logical one) whose domain is a set, then the image of the function is also a set.

Choice

$$\forall X\,[\emptyset \notin X \implies \exists f : X \to \cup X(\forall A \in X\,(f(A) \in A))]$$

Choice functions exist for all sets X.

.2 Some common objects in formal type theory

We present here the formal rules for three kinds of objects in type theory: dependent function types ($\prod$ types), $\mathbb{N}$, and the identity types. We also offer a few words about formal type theory in general.

In formal type theory, all objects are terms. "Types" are simply those terms of a universe type $\mathcal{U}_i$, and each universe is itself a term of a next higher universe $\mathcal{U}_{i+1}$. We avoid a lengthy discussion of universes — the point is that so far we have really only considered typing judgments (e.g. $p : P$, $A \times B : \mathcal{U}$, $(a, b) : A \times B$) and judgments of equality between terms (e.g. $p \equiv q : P, f(n) \equiv \Phi', 2 \equiv \mathrm{succ}(\mathrm{succ}(0))$).

There is one other kind of judgment that we have left out: that some expression Γ is a well-formed context for making hypothetical judgments like $\Gamma \vdash \Phi : B$. We do not fully describe contexts here — the idea is just that Γ is a well-formed context when the expression consists of a list of typing judgments regarding variables, possibly ones appearing on the right side of the turnstile $\vdash$. What we need to know is that the other two kinds of judgments are technically *always* formulated in some context Γ, even if the context is empty (and hence can be omitted in everyday practice). Therefore, the formal rules below presume some background context Γ.

To the right of each rule is a name, which is useful for keeping track of which rules are being invoked in a formal derivation of some judgment.

Dependent function types

$$\frac{\Gamma \vdash A \,:\, \mathcal{U} \qquad \Gamma,\, x : A \vdash B \,:\, \mathcal{U}}{\Gamma \vdash \prod_{x:A} B \,:\, \mathcal{U}} \;\Pi \text{ FORM}$$

(Notice that B is simply a "meta-variable," a placeholder for what might be a more complicated expression involving the (internal) variable x. Many of the symbols in the following rules are in fact meta-variables, as should be clear from context.)

$$\frac{\Gamma,\, x : A \vdash b \;:\; B}{\Gamma \vdash \lambda(x : A).\, b \;:\; \prod_{x:A} B}\ \Pi \text{ INTRO}$$

$$\frac{\Gamma \vdash f \;:\; \prod_{x:A} B \qquad \Gamma \vdash a \;:\; A}{\Gamma \vdash f(a) \;:\; B[a\,/x]}\ \Pi \text{ ELIM}$$

(The notation $B[a\,/x]$ indicates the result of replacing each occurrence of the variable x in the expression B with the term a.)

$$\frac{\Gamma,\, x : A \vdash b \;:\; B \qquad \Gamma \vdash a \;:\; A}{\Gamma \vdash (\lambda(x : A).\, b)(a) \equiv b[a\,/\,x] \;:\; B[a\,/x]}\ \Pi \text{ COMP}$$

$$\frac{\Gamma \vdash f \;:\; \prod_{x:A} B}{\Gamma \vdash f \equiv (\lambda(x : A).f(x) \;:\; \prod_{x:A} B}\ \Pi \text{ UNIQ}$$

The type $\mathbb{N}$ of natural numbers

$$\frac{\Gamma \text{ctx}}{\Gamma \vdash \mathbb{N} \;:\; \mathcal{U}}\ \mathbb{N} \text{ FORM}$$

$$\frac{\Gamma \text{ctx}}{\Gamma \vdash 0 \;:\; \mathbb{N}}\ \mathbb{N} \text{ INTRO}$$

$$\frac{\Gamma \vdash n \;:\mathbb{N}}{\Gamma \vdash \text{succ}(n) \;:\mathbb{N}}\ \mathbb{N} \text{ INTRO}$$

$$\frac{\begin{array}{c}\Gamma,\, x \;:\; \mathbb{N} \vdash C \;:\; \mathcal{U} \\ \Gamma,\vdash 0 \;:\; c_0 \;:\; C[0\,/\,x] \qquad \Gamma,\, x \;:\; \mathbb{N},\, :\, y : C \vdash c_s \;:\; C[\text{succ}(x)/x] \qquad \Gamma \vdash n \;:\mathbb{N}\end{array}}{\Gamma \vdash \text{ind}_{\mathbb{N}}(C, c_0, c_s, n) \;:\; C[n\,/x]}\ \mathbb{N} \text{ ELIM}$$

$$\frac{\begin{array}{c}\Gamma,\, x\,:\,\mathbb{N} \vdash C\,:\,\mathcal{U} \\ \Gamma,\vdash 0\,:\,c_0\,:\,C[0\,/\,x] \qquad \Gamma,\, x\,:\,\mathbb{N},:\,y : C \vdash c_s\,:\,C[\mathrm{succ}(x)/x] \qquad \Gamma \vdash n\,:\mathbb{N}\end{array}}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, 0) \equiv c_0\,:\,C[0\,/x]}\ \mathbb{N}\text{ COMP}$$

$$\frac{\begin{array}{c}\Gamma,\, x\,:\,\mathbb{N} \vdash C\,:\,\mathcal{U} \\ \Gamma,\vdash 0\,:\,c_0\,:\,C[0\,/\,x] \qquad \Gamma,\, x\,:\,\mathbb{N},:\,y : C \vdash c_s\,:\,C[\mathrm{succ}(x)/x] \qquad \Gamma \vdash n\,:\mathbb{N}\end{array}}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) \equiv c_s\,(n,\, ind_{\mathbb{N}}(C, c_0, c_s, n)\,/\,x, y))\,:\,C[\mathrm{succ}(n)\,/x]}\ \mathbb{N}\text{ COMP}$$

Identity Types

$$\frac{\Gamma \vdash A\,:\mathcal{U} \qquad \Gamma \vdash a\,:A \qquad \Gamma \vdash b\,:A}{\Gamma \vdash a =_A b : \mathcal{U}}\ \textsc{Id}\text{ FORM}$$

$$\frac{\Gamma \vdash A\,:\mathcal{U} \qquad \Gamma \vdash a\,:A}{\Gamma \vdash \mathrm{refl}_a\,:\,a =_A a}\ \textsc{Id}\text{ INTRO}$$

$$\frac{\begin{array}{c}\Gamma \vdash a\,:A \qquad \Gamma \vdash b\,:A \qquad \Gamma \vdash P\,a =_A b \\ \Gamma,\, x : A,\, y : A,\, p : x =_A y \vdash C : \mathcal{U} \qquad \Gamma,\, z : A,\vdash c : C[z, z, \mathrm{refl}_z\,/\,x, y, p]\end{array}}{\Gamma \vdash \mathrm{ind}_{=_A}(C, c, a, b, P)\,:\,C[a, b, P\,/\,x, y, p]}\ \textsc{Id}\text{ ELIM}$$

$$\frac{\begin{array}{c}\Gamma \vdash a\,:A \\ \Gamma,\, x : A,\, y : A,\, p : x =_A y \vdash C : \mathcal{U} \qquad \Gamma,\, z : A,\vdash c : C[z, z, \mathrm{refl}_z\,/\,x, y, p]\end{array}}{\Gamma \vdash \mathrm{ind}_{=_A}(C, c, a, a, \mathrm{refl}_a) \equiv c[a\,/\,z]C[a, a, \mathrm{refl}_a\,/\,x, y, p]}\ \textsc{Id}\text{ COMP}$$

.3 A computer-verified proof

The following simple proof is written for the freeware system COQ, which can be downloaded online. Much of the Univalent Foundations program consists of computer libraries of theorems formalized within this system. COQ is based on the Calculus of Inductive Constructions (CIC), which is similar to (but distinct from) Martin-Löf's type theory in many respects. A tutorial on COQ is outside the scope

of this paper, but full tutorials can be found on the main website [2], which is the source of the proof below.

The code is rather intuitive and motivates the general idea of computer-checked proofs based on type theory. COQ process each line of the proof sequentially, and a line-by-line explanation is given below.

```
Variables A B C : Prop.
Check (A -> B -> C) -> (A -> B) -> A -> C.
Goal (A -> B -> C) -> (A -> B) -> A -> C.
intro Hyp1.
intro Hyp2.
intro Hyp3.
apply Hyp1.
exact Hyp3.
apply Hyp2.
exact Hyp3.
Save ImplicationProposition.

Goal (A -> B -> C) -> (A -> B) -> A -> C.
apply ImplicationProposition.
```

Explanation

```
Variables A B C : Prop.
```

We introduce three variables A, B, C as terms of type Prop. COQ slightly departs from the proposition-as-types paradigm and instead has a type Prop whose terms represent propositions.

```
Check (A -> B -> C) -> (A -> B) -> A -> C.
Goal (A -> B -> C) -> (A -> B) -> A -> C.
```

The first line is a command to COQ to check the type of the term to the right. In this case, COQ tells us that the term is itself a proposition, which of course we knew. Naturally, the expression $- >$ is just the type former $\Longrightarrow$ (equivalently $\rightarrow$). The second line signals to COQ that we wish to prove the proposition on the right,

[2] `http://coq.inria.fr/`

which sends COQ into "proof mode." The system will now guide us in constructing a term of $(A \implies B \implies C) \implies (A \implies B) \implies (A \implies C)$ (in our usual type theory notation).

```
intro Hyp1.
intro Hyp2.
intro Hyp3.
```

The first line introduces a term Hyp1 of type $(A \implies B \implies C)$. Now COQ only needs for us to construct a term of $(A \implies B) \implies (A \implies C)$, possibly using Hyp1. That is, introducing Hyp1 is effectively to assume the hypothesis that $A \implies B \implies C$, keeping in mind that hypothesizing a proposition means hypothesizing a *proof* of it.

We next introduce Hyp2: $A \implies B$, after which COQ only needs us to construct a term of $A \implies C$. Next, we introduce $Hyp3 : A$. From here, we just need to construct a term of C using our collective hypotheses.

```
apply Hyp1.
```

We tell COQ that we will construct a term of C using the term of type $(A \implies B \implies C)$. Then COQ only needs for us to supply two terms, one of type A and the other of type B.

```
exact Hyp3.
```

Hyp3 is exactly a term of A! Now we just need a term of B.

```
apply Hyp2.
exact Hyp3.
```

We will construct a term of B using the term of $A \implies B$. Then we just need to supply a term of A, which again is exactly the term Hyp3. COQ now recognizes that the preceding lines of code together constitute a proof (i.e. term) of the original proposition.

```
Save ImplicationProposition.
Goal (A -> B -> C) -> (A -> B) -> A -> C.
apply ImplicationProposition.
```

We save our proof for possible future use. If we ever want to prove the proposition again, we just apply the proof we created.

BIBLIOGRAPHY

[AW09] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146:45–55, 2009.

[Gra08] Johan G. Granstrm. *Reference and Computation in Intuitionistic Type Theory*. PhD thesis, Uppsala University, 2008.

[LS13] Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *LICS 2013: Proceedings of the Twenty-Eighth Annual ACM/IEEE Symposium on Logic in Computer Science*, 2013.

[ML84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

[ML98] Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford University Press, 1998.

[SA13] Michael A. Warren Steve Awodey, lvaro Pelayo. Voevodsky's univalence axiom in homotopy type theory. *Notices of the American Mathematical Society.*, 60:1164–1168, 2013.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[War08] Michael A. Warren. *Homotopy Theoretic Aspects of Constructive Type Theory*. PhD thesis, Carnegie Mellon University, 2008.