

Homotopy Type Theory and The Univalent Foundations of Mathematics

A Thesis

submitted to

Indian Institute of Science Education and Research, Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Varun Prasad

Registration No. : 20111028



Indian Institute of Science Education and Research, Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2017

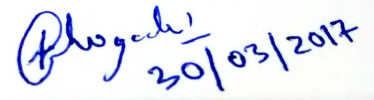
Supervisor: Dr. Amit Hogadi

© Varun Prasad 2017

All rights reserved

Certificate

This is to certify that this dissertation entitled Homotopy Type Theory and The Univalent Foundations of Mathematics towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Varun Prasad at Indian Institute of Science Education and Research under the supervision of Dr. Amit Hogadi, Associate Professor, Department of Mathematics, during the academic year 2016-2017.

A handwritten signature in blue ink that reads "Amit Hogadi" followed by the date "30/03/2017".

Dr. Amit Hogadi

Committee:


Dr. Amit Hogadi

Dr. Steven Spallone

This thesis is dedicated to the empty type.

Declaration

I hereby declare that the matter embodied in the report entitled Homotopy Type Theory and The Univalent Foundations of Mathematics are the results of the work carried out by me at the Department of Mathematics, Indian Institute of Science Education and Research, Pune, under the supervision of Dr. Amit Hogadi and the same has not been submitted elsewhere for any other degree.



30 / 03 / 2017

Varun Prasad

Acknowledgments

I would like to acknowledge, first of all, my guide and mentor Dr. Amit Hogadi whose guidance and presence was invaluable for this project. I would also like to thank Dr. Shane D'Mello with whom I have had many interesting and insightful discussions on the subject. I would like to express my gratitude towards Dr. Steven Spallone, Dr. Vivek Mallick and Dr. Manish Mishra, who along with Dr. Amit Hogadi and Dr. Shane D'Mello, were a part of the seminar series conducted in IISER, Pune on Homotopy Type Theory.

I thank our joint community of fifth year and PhD student(s) at the mathematics department of IISER, especially, Visakh, Ajith, Chris and Neeraj, who were my constant companions in exploring the profound dimensions and scope of mathematics and this project.

Lastly, I would also like to take this opportunity to express my gratitude towards Prof. Vladimir Voevodsky and all the others working in the Univalent Foundations program, without whose insights and efforts, this project would not exist today.

Abstract

Homotopy Type Theory is a new interpretation of Martin-Löf’s intentional, constructive type theory into abstract homotopy theory. Type theories refer to a class of formal languages which were proposed as foundations of constructive mathematics and which have since been studied and developed by theoretical computer scientists because of their desirable computational properties. In homotopy type theory, types are spaces upto homotopy, propositional equalities are homotopies and type isomorphisms are homotopy equivalences. Logical constructions in type theory are then homotopy-invariant constructions on spaces. This interpretation has many desirable properties including a natural axiomatization of higher categorical thinking.

The Univalent Foundations of mathematics is a comprehensive, computational foundations of mathematics based on homotopy type theory. Vladimir Voevodsky’s univalence axiom relates propositional equality of types in a universe with homotopy equivalence of small types. The Univalent Foundations program is currently being implemented in automated proof assistants like *Coq*. This thesis is an introduction to this program.

Contents

Abstract	xi
1 Introduction	1
1.1 Type Theory	1
1.2 The Holy Trinity	2
1.3 Homotopy Type Theory	2
1.4 The Univalent Foundations and Implementation in Coq	3
2 Formal Type Theory	5
2.1 Contexts	7
2.2 Structural Rules	8
2.3 Type Universes	10
2.4 Dependent Function Types	12
2.5 Dependent Pair Types	14
2.6 Coproduct Types	16
2.7 The Empty Type	17
2.8 The Unit Type	18
2.9 The Natural Number Type	19

2.10	Identity Types	20
2.11	Homotopy Type Theory	22
3	Homotopy Type Theory and the Univalence Axiom	25
3.1	Types as Sets	26
3.2	Types as Propositions	27
3.3	The Homotopy Interpretation of Type Theory	29
3.4	Function Extentionality	35
3.5	Univalence Axiom	39
4	Sets and Logic	43
4.1	Sets and n -type	43
4.2	Intuitionistic Logic	46

Chapter 1

Introduction

Homotopy Type Theory is a newly emerging field at the interface of mathematics, logic and theoretical computer science. It is an interpretation of Martin-Löf's intentional, constructive type theory using abstract homotopy theory. Vladimir Voevodsky's Univalent Foundations is a computational foundation of mathematics based on Homotopy Type Theory which is an alternative to the current set theoretic foundations (ZFC).

One of the primary motivations for formalizing the informal mathematics we do in this foundation is the possibility of building viable proof checkers, i.e., softwares which take mathematical proofs as input and check their validity. This has become a necessity today considering the lengths of proofs in modern mathematics and the severe limitations of the peer review process.

1.1 Type Theory

Type Theory was initially proposed by Bertrand Russell to resolve paradoxes in naive set theory. It has now developed into a branch of mathematical logic and theoretical computer science with major contributions from Alonzo Church and Per Martin-Löf, among others. Here, we consider Martin-Löf's intentional constructive type theory as a foundation of math-

ematics based on intuitionistic logic.

The primitive notion in type theory is that of a type which is similar to data types in programming languages. The study of this system begins with the basic judgment $x : A$ read as ‘the term x is of type A ’. The deductive system of type theory consists of rules of forming new judgments from pre-existing ones. This is the same as constructing a term of a type in a given context (i.e. under certain assumptions). These rules therefore specify elaborately structured types which are classified into type formers like the dependent function type, dependent pair type, coproduct type, booleans, natural numbers and the identity type.

1.2 The Holy Trinity

At the most fundamental level, this subject is a study of the deep correspondence that has been discovered between Type Theory (Programming), Proof Theory (Logic) and Category Theory (Mathematics). Consider the basic judgment $x : A$. This is interpreted as ‘the term x is of type A ’ in Type Theory; ‘ x is a proof of proposition A ’ in Proof Theory and ‘ x is an object of category A ’ in Category Theory. The rules of type theory then correspond to rules of logic in Proof Theory and universal constructions in Category Theory.

Therefore, types, propositions and categories are essentially the same objects expressed in different theories. In particular, propositions and sets (which are categories with only identity morphisms), the primitive notions of the current foundations, are types in type theory.

1.3 Homotopy Type Theory

In Homotopy Type Theory, the above correspondence is modified to replace Category Theory with abstract Homotopy Theory, i.e, types are interpreted as spaces upto homotopy (or

∞ -groupoids) instead of categories. The judgment $x : A$ is now interpreted as ‘ x is a point in space A ’ and rules of type theory correspond to homotopy invariant constructions on spaces.

This interpretation explains some of the features of type theory like the identity type much more naturally than the category theoretic interpretation. The connection between Type Theory and Homotopy theory was recently discovered by Vladimir Voevodsky, Steve Awodey and Michael Warren independently and is currently the focus of intense investigation.

1.4 The Univalent Foundations and Implementation in Coq

Voevodsky arrived at Homotopy Type Theory by showing how to model Type Theory using Kan simplicial sets ^[3]. The simplicial model of type theory satisfies an additional property called Univalence which is not usually assumed in Type Theory. Homotopy Type Theory along with the Univalence axiom is known as the Univalent Foundation of mathematics. The Univalence axiom basically allows isomorphic structures to be formally identified which was not possible till now in the ZFC based foundations. This has many far reaching consequences which are yet to be fully understood.

Coq is an interactive proof management system. It is designed to develop mathematical proofs and write formal specifications, programs and verify that programs are correct with respect to their specifications. Coq provides a specification language called GALLINA which is based on an intentional dependent type theory called the Calculus of Inductive Constructions (CIC). The Martin-Löf type theory can be seen as a fragment of CIC. Hence, mathematics formalized in the Univalent Foundations can be written in Coq and libraries created so far include topics as diverse as K-theory, p-adics, category theory, real numbers and topology.

The primary reference for this thesis is the book “Homotopy type theory: Univalent foundations of mathematics”^[1]. Since all definitions and theorems are from this reference, in order to avoid citing the same reference repeatedly all along the text, I would like to cite it right at the beginning.

Chapter 2

Formal Type Theory

Instead of first developing mathematics informally in this new foundations, called the Univalent Foundations, and then describing a formal theory of this foundation, we undertake a more rigorous treatment of the foundation right at the beginning. In this chapter, we introduce formally the theory of dependent types and then extend it to include additional axioms of the Univalent Foundations.

When formalizing set theory, we first introduce a deductive logic framework, the first-order predicate logic, and then introduce ZFC (Zermelo-Frankael axioms with choice) as a particular theory in this framework. Type theory, in contrast to set theory, is its own deductive system.

In type theory, there is only one basic notion, namely, types. In contrast, in set theory we have two basic notions : sets and propositions. Thus, set theory is not only about sets, but about sets and propositions and how they interact with each other. In type theory, sets and propositions are both types.

The basic notion that is studied in logic is a proposition or a well-formed assertion. If A is a proposition, then A is true, A is false, A has a proof are judgments about the proposition A . A deductive system or framework is a set of rules for deriving judgments from other

judgments.

In formal type theory, there are three kinds of judgments :

1. $\Gamma \text{ ctx}$
2. $\Gamma \vdash a : A$ and
3. $\Gamma \vdash a \equiv a' : A$

The basic judgment of type theory is that $a : A$, which is read as ‘the term a is of type A ’. If the type A is interpreted as a set then this judgement is similar to the judgment ‘ $a \in A$ is true’. If A is a proposition, then $a : A$ corresponds to the judgment ‘ a is a proof of proposition A ’.

However, a is a proof of a proposition A is only meaningful under a list of assumptions or what we call an ambient context. This is denoted by Γ . Thus, $\Gamma \vdash a : A$ is read as ‘in the context Γ , a is a term of type A ’.

$\Gamma \vdash a \equiv a' : A$ is read as ‘in the context Γ , the terms a and a' of type A are *definitionally equal* which means that a is equal to a' by definition. There is another notion of equality in type theory which we will encounter later called propositional equality. Note that a definitional equality is not a proposition. It does not make sense, for example, to negate such a equality and assume that a is not definitionally equal to a' as a hypothesis to prove any theorem.

However, both these kinds of judgments are justified only in a valid list of assumptions or a well-formed context. This is captured in the first kind of judgment ‘ $\Gamma \text{ ctx}$ ’ which is read as Γ is a well-formed context.

Judgements are derived from other judgments using inference rules. A typical inference

rule is of the form :

$$\frac{\mathcal{J}_1 \quad \dots \quad \mathcal{J}_n}{\mathcal{J}} \text{ NAME}$$

A derivation of a judgment is a tree that is constructed from inference rules, with the judgment at the root of the tree.

We now look at the various rules of inference which define the various kinds of types in type theory and describe their behaviour.

2.1 Contexts

A context is an ordered list of judgments of the form $x_1 : A_1, x_2 : A_2, x_3 : A_3, \dots, x_n : A_n$ where x_i 's are distinct variables and each x_i is assumed to have the type A_i . It is important to note that a context is an ordered list since A_i could depend on variables x_1, \dots, x_{i-1} . The list could be empty. Generally, contexts are denoted using letters Γ, Δ , etc.

The judgement $\Gamma \text{ ctx}$ expresses the fact that Γ is a well-formed context. The formation of contexts are governed by the following inference rules :

1.

$$\frac{}{. \text{ ctx}} \text{ ctx-EMP} \tag{2.1}$$

We can always derive the judgment that the empty context is well-formed. This is how the rule ctx-EMP is read or translated into English.

2.

$$\frac{x_1 : A_1, x_2 : A_2, x_3 : A_3, \dots, x_{n-1} : A_{n-1} \vdash A_n : \mathcal{U}_i}{(x_1 : A_1, x_2 : A_2, x_3 : A_3, \dots, x_n : A_n) \text{ ctx}} \text{ ctx-EXT} \tag{2.2}$$

In ctx-EXT, the variable x_n must be distinct from the variables x_1, \dots, x_{n-1} . The hy-

pothesis of the rule says that if in the context x_1, \dots, x_n, A_n is a type in the universe \mathcal{U}_i , (which essentially means that A_n is a well-defined type; see section 2.3 for more details), then the context $(x_1 : A_1, x_2 : A_2, x_3 : A_3, \dots, x_n : A_n)$ is well-formed.

2.2 Structural Rules

These are a class of inference rules which pin down the behaviour of contexts, variables and terms (substituting terms for variables and introducing redundant variables) and definitional equality or judgmental equality.

$$1. \quad \frac{(x_1 : A_1, x_2 : A_2, x_3 : A_3, \dots, x_n : A_n) \quad ctx}{x_1 : A_1, x_2 : A_2, x_3 : A_3, \dots, x_n : A_n \vdash x_i : A_i} \quad \text{Vble} \quad (2.3)$$

The rule Vble ensures that the judgments that occur in any well-formed context are, indeed, assumptions or hypotheses.

The next four inference rules are the rules for substituting variables and introducing new ones.

$$2. \quad \frac{\Gamma \vdash a : A \quad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]} \quad \text{Subst}_1 \quad (2.4)$$

Here, the notation $t[a/x]$ is read as the term a is substituted for the variable x in the expression t . Subst_1 then simple states that if a is a term of A then it can be substituted for x in the judgment $\Gamma, x : A, \Delta \vdash b : B$ by substituting all free occurrences of x in the expressions Δ, b and B to obtain the judgment $\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]$.

3.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, \Delta \vdash b : B}{\Gamma, x : A, \Delta \vdash b : B} \text{Wkg}_1 \quad (2.5)$$

Wkg_1 tells us how to introduce a assumption $x : A$ to weaken the context of the judgment $\Gamma, \Delta \vdash b : B$. Note that x is distinct from all the variables that occur in Γ and Δ .

The substitution and weakening rules corresponding to definitional equality which are judgments of the kind $a \equiv b : A$ are stated similarly.

4.

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A, \Delta \vdash b \equiv c : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \equiv c[a/x] : B[a/x]} \text{Subst}_2 \quad (2.6)$$

5.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, \Delta \vdash b \equiv c : B}{\Gamma, x : A, \Delta \vdash b \equiv c : B} \text{Wkg}_2 \quad (2.7)$$

Next, we need to introduce rules which state that definitional equality is indeed an equality, i.e., it is reflexive, symmetric and transitive.

6.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \quad (2.8)$$

Given $a : A$, we can deduce that a is definionally equal to itself as terms of A , i.e., definitional equality is reflexive.

7.

$$\frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A} \quad (2.9)$$

If a and b are terms of A and a is definitionally equal to b , then b is definitionally equal to a , i.e., definitional equality is symmetric.

8.

$$\frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A} \quad (2.10)$$

The above rules states that definitional equality is transitive. Additionally, we also have the following two rules where $A \equiv B : \mathcal{U}_i$ can be read as A and B are well-defined types which are definitionally equal :

9.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a : B} \quad (2.11)$$

10.

$$\frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a \equiv b : B} \quad (2.12)$$

2.3 Type Universes

In type theory, types can be terms of other types. This leads us to the question as to whether there is a type of all types. Assuming a type of all types will give rise to paradoxes similar to

the Russell's paradox. In order to take care of this, we assume a *countably infinite cumulative hierarchy* of type universes :

$$\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \dots$$

that is, a countably infinite hierarchy of type universes where each universe is contained in the next and every type in \mathcal{U}_i is also a type in \mathcal{U}_{i+1} . Every type A in type theory which is well-defined belongs to some universe $s\mathcal{U}_i$.

The following rules pin down the notion of Type universes :

1.
$$\frac{\Gamma \quad ctx}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}} \quad \mathcal{U}\text{-INTRO} \quad (2.13)$$

Each universe is contained in the next, that is, there is a hierarchy of type universes.

2.
$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}} \quad \mathcal{U}\text{-CUMUL} \quad (2.14)$$

Every type in \mathcal{U}_i is also a type in \mathcal{U}_{i+1} , that is, the hierarchy is cumulative.

Notation : A is a type will be denoted as $A : \mathcal{U}$ The index i of the type universe is suppressed with the understanding that the hierarchy of universes that we are dealing with in a particular theory can be consistently indexed. This is also called typical ambiguity.

From this point onwards, we shall follow a general pattern for introducing and characterizing new types. A new type is specified by giving the following rules :

1. A **formation rule** which tells us when and how to form a new type of this kind.
2. **Introduction rules** which tell us how to construct or introduce terms of this type. These are also called the type's constructors.
3. **Elimination rules** or an induction principle, which tell us what one can do with terms of this type. These are also called the type's eliminators. Eliminators can also be thought of, in some cases, as describing functions or maps in and out of the type.
4. **Computation rules** tell us what happens when the eliminators act on the constructors, i.e., what happens when the elimination rules are applied to the terms that are introduced using the introduction rules.
5. Optional **uniqueness principles**, which are judgmental equalities which explain how every element of that type is uniquely determined by applying the elimination rules to it. In a certain sense, it tells us what happens when the constructors act on eliminators.

2.4 Dependent Function Types

Dependent function types or Π -types are a generalization of the notion of functions in mathematics. In set theoretic mathematics, a function $f : A \rightarrow B$ has two sets associated with it, namely, the domain A and the codomain B . Every point in A is mapped uniquely to a point in B . A dependent function can be thought of as a function in which the codomain varies with the point chosen in the domain. So, for every point in the domain, we first specify a codomain and then map the point to a point in the appropriate codomain.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_i} \quad \Pi\text{-FORM} \quad (2.15)$$

$\Gamma \vdash A : \mathcal{U}_i$ is read as in the context Γ , A is a type in the universe \mathcal{U}_i . The judgment $\Gamma, x : A \vdash B : \mathcal{U}_i$ tells us that under the context $\Gamma, x : A$, we have $B : \mathcal{U}_i$. Thus, for every x in A , we have a specific choice of a type $B(x)$ (i.e. B depends on x). This gives us a family of types, which we'll call B , that is parametrized by the type A .

The formation rule is then read as : If A is a type and B is a type family parametrized by A , then we can form the type of dependent functions $\prod_{(x:A)} B$.

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \prod_{(x:A)} B} \quad \text{\Pi-INTRO} \quad (2.16)$$

If for every $x : A$, we are given a specific choice of term $b(x)$ in the type $B(x)$, then we can form or construct a term called $\lambda(x : A).b$ of the type $\prod_{(x:A)} B$. This is the only way to introduce terms of a dependent function type. Here, $\lambda(x : A).b$ is a primitive constant that is being introduced.

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]} \quad \text{\Pi-ELIM} \quad (2.17)$$

The elimination rule tells us what to do with a dependent function, namely, act it on a point in the domain to obtain a point in the appropriate codomain. It is read formally as if in the context Γ we have a term $f : \prod_{(x:A)} B$ and $a : A$, then in the same context Γ , we obtain a term $f(a) : B$. Again, $f(a)$ is a primitive constant that is being introduced here.

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)(a) \equiv b[a/x] : B[a/x]} \quad \Pi\text{-COMP} \quad (2.18)$$

The computation rule tells us what happens when the terms introduced in the introduction rule act according to the elimination rule. So, if we have the judgment $\Gamma, x : A \vdash b : B$, then we have $\Gamma \vdash \lambda(x : A).b : \prod_{(x:A)} B$ because of Π -INTRO. So, if in Π -ELIM, we replace f with $\lambda(x : A).b$, then we get $\lambda(x : A).b(a) : B[a/x]$. The computation rule says that this is just $b[a/x]$ as we would expect.

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B}{\Gamma \vdash f \equiv (\lambda x.f(x)) : \prod_{(x:A)} B} \quad \Pi\text{-UNIQ} \quad (2.19)$$

The uniqueness principle tells us that the constructors of $\prod_{(x:A)} B$ are essentially the only terms of this type. Or in other words, if we know the value of the dependent function at every point in the domain, then we have uniquely determined the dependent function.

In the special case, we the type family B is the constant family, i.e., all the terms in the domain have the same codomain, we denote $\prod_{(x:A)} B$ by $A \rightarrow B$ and call it the type of non-dependent functions from A to B .

2.5 Dependent Pair Types

Dependent pair types or Σ -types are the generalization of the binary cartesian product in mathematics. It is denoted by $\sum_{(x:A)} B$. Terms of this type are pairs (a, b) where $a : A$ and $b : B(a)$.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \sum_{(x:A)} B : \mathcal{U}_i} \quad \Sigma\text{-FORM} \quad (2.20)$$

The formation rule states that if A is a type and B is a type family parametrized by A , then we can form the type of dependent pairs $\sum_{(x:A)} B$.

$$\frac{\Gamma, x : A \vdash B : \mathcal{U}_i \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum_{(x:A)} B} \quad \Sigma\text{-INTRO} \quad (2.21)$$

If for some $a : A$, we are given a specific choice of term b in the type $B(a)$, then we can introduce the term called (a, b) in the type $\sum_{(x:A)} B$. Here, (a, b) is a primitive constant that is being introduced.

$$\frac{\Gamma, z : \sum_{(x:A)} B \vdash C : \mathcal{U}_i \quad \Gamma, x : A, y : B \vdash g : C[(x, y)/z] \quad \Gamma \vdash p : \sum_{(x:A)} B}{\Gamma \vdash \text{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, p) : C[p/z]} \quad \Sigma\text{-ELIM} \quad (2.22)$$

The elimination rule can be thought of as describing functions out of the dependent pair type. However, since, we are in type theory, we consider the general case of dependent functions instead of just functions. So, if C is a type family parametrized by $\sum_{(x:A)} B$ and we have a dependent function such that we know its value for every (x, y) , then we have determined the function on all of $\sum_{(x:A)} B$. To elaborate, if for every $x : A$ and $y(x) : B(x)$, we have the value of the function given by $g : C[(x, y)]$, then we have $\text{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, p) : C[p/z]$, the value of the function at a arbitrary term p .

$$\frac{\Gamma, z : \sum_{(x:A)} B \vdash C : \mathcal{U}_i \quad \Gamma, x : A, y : B \vdash g : C[(x, y)/z]}{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]} \quad \Sigma\text{-COMP} \quad (2.23)$$

$$\Gamma \vdash \text{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, (a, b)) \equiv g[a, b/x, y] : C[(a, b)/z]$$

The computation rule tells us what the function we defined in the elimination rule is when applied to pairs (a, b) .

These various kinds of types that we are introducing are also referred to as type formers.

2.6 Coproduct Types

Coproducts are a notion in type theory which correspond to the notion of a disjoint union in set-theoretic mathematics. It is notated as in $A + B$. The rules of inference follow a trend similar to the above type formers.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A + B : \mathcal{U}_i} \quad +\text{-FORM} \quad (2.24)$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash B : \mathcal{U}_i \quad \Gamma \vdash a : A}{\Gamma \vdash \text{inl}(a) : A + B} \quad +\text{-INTRO}_1 \quad (2.25)$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash B : \mathcal{U}_i \quad \Gamma \vdash b : B}{\Gamma \vdash \text{inr}(b) : A + B} \quad +\text{-INTRO}_2 \quad (2.26)$$

$$\begin{array}{c}
\Gamma, z : (A + B) \vdash C : \mathcal{U}_i \quad \Gamma, x : A \vdash c : C[inl(x)/z] \quad \Gamma, y : B \vdash d : C[inr(y)/z] \\
\Gamma \vdash e : (A + B) \\
\hline
\Gamma \vdash ind_{A+B}(z.C, x.c, y.d, e) : C[e/z]
\end{array}
\quad \begin{array}{l}
+\text{-ELIM} \\
(2.27)
\end{array}$$

$$\begin{array}{c}
\Gamma, z : (A + B) \vdash C : \mathcal{U}_i \quad \Gamma, x : A \vdash c : C[inl(x)/z] \quad \Gamma, y : B \vdash d : C[inr(y)/z] \\
\Gamma \vdash a : A \\
\hline
\Gamma \vdash ind_{A+B}(z.C, x.c, y.d, inl(a)) \equiv c[a/x] : C[inl(a)/z]
\end{array}
\quad \begin{array}{l}
+\text{-COMP}_1 \\
(2.28)
\end{array}$$

$$\begin{array}{c}
\Gamma, z : (A + B) \vdash C : \mathcal{U}_i \quad \Gamma, x : A \vdash c : C[inl(x)/z] \quad \Gamma, y : B \vdash d : C[inr(y)/z] \\
\Gamma \vdash b : B \\
\hline
\Gamma \vdash ind_{A+B}(z.C, x.c, y.d, inr(b)) \equiv d[b/y] : C[inr(b)/z]
\end{array}
\quad \begin{array}{l}
+\text{-COMP}_2 \\
(2.29)
\end{array}$$

2.7 The Empty Type

In all the type formers that we have introduced so far, we have never, postulated the existence of a type. The first example of this kind is the empty type which exists in every context as the formation rule indicates. It has no introduction rules as expected.

$$\frac{\Gamma \quad ctx}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i} \quad \mathbf{0}\text{-FORM} \quad (2.30)$$

$$\frac{\Gamma, x : \mathbf{0} \vdash C : \mathcal{U}_i \quad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathit{ind}_{\mathbf{0}}(x.C, a) : C[a/x]} \quad \mathbf{0}\text{-ELIM} \quad (2.31)$$

The elimination rule says that if in some context, the empty type is inhabited, then every type in the universe we are working in is inhabited. Thus, the empty type can also be interpreted as the proposition ‘False’ and if we have a proof of ‘false’, then we have a proof every other proposition. For more details, see Section 3.2.

2.8 The Unit Type

The unit type has only one term, a formal object \star and every map out of it is determined if we know its value on \star . The rules of inference follow a trend similar to the above type formers.

$$\frac{\Gamma \quad ctx}{\Gamma \vdash \mathbf{1} : \mathcal{U}_i} \quad \mathbf{1}\text{-FORM} \quad (2.32)$$

$$\frac{\Gamma \quad ctx}{\Gamma \vdash \star : \mathbf{1}} \quad \mathbf{1}\text{-INTRO} \quad (2.33)$$

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathcal{U}_i \quad \Gamma \vdash c : C[\star/x] \quad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathit{ind}_1(x.C, c, a) : C[a/x]} \quad \mathbf{1}\text{-ELIM} \quad (2.34)$$

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathcal{U}_i \quad \Gamma \vdash c : C[\star/x]}{\Gamma \vdash \mathit{ind}_1(x.C, c, \star) \equiv c : C[\star/x]} \quad \mathbf{1}\text{-COMP} \quad (2.35)$$

2.9 The Natural Number Type

We also introduce the type of natural numbers. The terms of this type are of the form $0, \mathit{succ}(0), \mathit{succ}(\mathit{succ}(0)), \dots$.

$$\frac{\Gamma \quad \mathit{ctx}}{\Gamma \vdash \mathbb{N} : \mathcal{U}_i} \quad \mathbb{N}\text{-FORM} \quad (2.36)$$

$$\frac{\Gamma \quad \mathit{ctx}}{\Gamma \vdash 0 : \mathbb{N}} \quad \mathbb{N}\text{-INTRO}_1 \quad (2.37)$$

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathit{succ}(n) : \mathbb{N}} \quad \mathbb{N}\text{-INTRO}_2 \quad (2.38)$$

$$\frac{\Gamma, x : \mathbb{N} \vdash C : \mathcal{U}_i \quad \Gamma \vdash c_0 : C[0/x] \quad \Gamma, x : \mathbb{N}, y : C \vdash c_s : C[\text{succ}(x)/x]}{\Gamma \vdash n : \mathbb{N}} \quad \text{N-ELIM}$$

$$\Gamma \vdash \text{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, n) : C[n/x]$$

(2.39)

The elimination rule of natural numbers corresponds to the principle of mathematical induction. When we interpret proposition as types, later, this is what a proof by induction will correspond to.

$$\frac{\Gamma, x : \mathbb{N} \vdash C : \mathcal{U}_i \quad \Gamma \vdash c_0 : C[0/x] \quad \Gamma, x : \mathbb{N}, y : C \vdash c_s : C[\text{succ}(x)/x]}{\Gamma \vdash \text{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, 0) \equiv c_0 : C[0/x]} \quad \text{N-COMP}_1$$

(2.40)

$$\frac{\Gamma, x : \mathbb{N} \vdash C : \mathcal{U}_i \quad \Gamma \vdash c_0 : C[0/x] \quad \Gamma, x : \mathbb{N}, y : C \vdash c_s : C[\text{succ}(x)/x]}{\Gamma \vdash n : \mathbb{N}} \quad \text{N-COMP}_2$$

$$\Gamma \vdash \text{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, \text{succ}(n))$$

$$\equiv c_s[n, \text{ind}_{\mathbb{N}}(x.C, c_0, x.y.c_s, n)/x, y] : C[\text{succ}(n)/x]$$

(2.41)

2.10 Identity Types

An important feature of type theory which does not have any set theoretic analogue is the Identity types. In the interpretation of proposition as types, if we have $a, b : A$, then

the proposition a is equal to b as terms of type A , is itself a type denoted by $a =_A b$. This equality is called propositional equality and is distinct from definitional equality that we have encountered so far.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathcal{U}_i} \text{=-FORM} \quad (2.42)$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash a : A}{\Gamma \vdash \text{refl}_a : a =_A a} \text{=-INTRO} \quad (2.43)$$

The introduction rule just introduces a constant refl_a and says that propositional equality is always reflexive.

$$\frac{\Gamma, x : A, y : A, p : x =_A y \vdash C : \mathcal{U}_i \quad \Gamma, z : A \vdash c : C[z, z, \text{refl}_z/x, y, p] \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash p' : a =_A b}{\Gamma \vdash \text{ind}_{=A}(x.y.p.C, z.c, a, b, p') : C[a, b, p'/x, y, p]} \text{=-ELIM} \quad (2.44)$$

The elimination rule for identity type is also called path induction and plays a critical role in homotopy type theory (See Chapter 3).

$$\frac{\Gamma, x : A, y : A, p : x =_A y \vdash C : \mathcal{U}_i \quad \Gamma, z : A \vdash c : C[z, z, \text{refl}_z/x, y, p] \quad \Gamma \vdash a : A}{\Gamma \vdash \text{ind}_{=A}(x.y.p.C, z.c, a, a, \text{refl}_a) \equiv c[a/z] : C[a, a, \text{refl}_a/x, y, p]} \text{=-COMP} \quad (2.45)$$

2.11 Homotopy Type Theory

In homotopy interpretation of type theory (see chapter 3), the following additional axioms are assumed. We state this here for the sake of completeness and go into a detailed discussion of these in Chapters 3 and 4.

1. Function Extentionality :

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \quad \Gamma \vdash g : \prod_{(x:A)} B}{\Gamma \vdash \text{funext}(f, g) : \text{isequiv}(\text{happly}_{f,g})} \quad \Pi\text{-EXT} \quad (2.46)$$

2. Univalence Axiom :

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash \text{univalence}(A, B) : \text{isequiv}(\text{idtoeqv}_{A,B})} \quad \mathcal{U}_i\text{-UNIV} \quad (2.47)$$

3. The circle :

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbb{S}^1 : \mathcal{U}_i} \quad \mathbb{S}^1\text{-FORM} \quad (2.48)$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \text{base} : \mathbb{S}^1} \quad \mathbb{S}^1\text{-INTRO}_1 \quad (2.49)$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \text{loop} : \text{base} =_{\mathbb{S}^1} \text{base}} \quad \mathbb{S}^1\text{-INTRO}_2 \quad (2.50)$$

$$\frac{\Gamma, x : \mathbb{S}^1 \vdash C : \mathcal{U}_i \quad \Gamma \vdash b : C[\text{base}/x] \quad \Gamma \vdash l : b =_{\text{loop}}^C \quad \Gamma \vdash p : \mathbb{S}^1}{\Gamma \vdash \text{ind}_{\mathbb{S}^1}(x.C, b, l, p) : C[p/x]} \quad \mathbb{S}^1\text{-ELIM} \quad (2.51)$$

$$\frac{\Gamma, x : \mathbb{S}^1 \vdash C : \mathcal{U}_i \quad \Gamma \vdash b : C[\text{base}/x] \quad \Gamma \vdash l : b =_{\text{loop}}^C}{\Gamma \vdash \text{ind}_{\mathbb{S}^1}(x.C, b, l, \text{base}) \equiv b : C[\text{base}/x]} \quad \mathbb{S}^1\text{-COMP}_1 \quad (2.52)$$

$$\frac{\Gamma, x : \mathbb{S}^1 \vdash C : \mathcal{U}_i \quad \Gamma \vdash b : C[\text{base}/x] \quad \Gamma \vdash l : b =_{\text{loop}}^C}{\Gamma \vdash \mathbb{S}^1 - \text{loopcomp} : \text{apd}_{(\lambda y. \text{ind}_{\mathbb{S}^1}(x.C, b, l, y))}(\text{loop}) = l} \quad \mathbb{S}^1\text{-COMP}_2 \quad (2.53)$$

The axioms corresponding to \mathbb{S}^n for all natural numbers n can be stated similarly and is assumed.

Chapter 3

Homotopy Type Theory and the Univalence Axiom

Consider the basic judgment in type theory $x : A$. In type theory, this is read as *the term x is of type A* . If we interpret types as sets, then this judgment would be interpreted as $x \in A$, i.e., *the element x belongs to the set A* . If we interpret types as propositions, then the same judgment would be interpreted as *x is a proof of proposition A* .

Thus, $x : A$ is a formal statement in the formal theory of types and each of its interpretations that we have seen above is the translation of this statement into various languages like that of sets and propositions. Another way of stating this point is that $x \in A$, for example, is the interpretation of $x : A$ in the set-theoretic model of types where types are modelled using sets. Similarly, we have a model of types using propositions.

What do the constructions of all different kinds of types (in Chapter 2) in formal type theory correspond to in these various interpretations of type theory? This is outlined in brief in Table 1.

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof of proposition	element of set	point in space
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	\perp, \top	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \rightarrow B$	$A \implies B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists x : A, B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall x : A, B(x)$	product	space of sections
Id_A	equality =	$\{(x, x) x \in A\}$	path space A^I

Table 1 : Comparing points of view on type-theoretic operations

Notations Used in Table 1 :

\perp : the formal proposition ‘False’ ; \top : the formal proposition ‘True’ ; $*$: contractible space;
 $A \vee B$: A or B ; $A \wedge B$: A and B ; $A \implies B$: A implies B .

3.1 Types as Sets

In the set-theoretic interpretation of types, a type A is a set. $x : A$ means that $x \in A$ and $x \equiv y : A$ is interpreted as x is definitionally equal to y as elements of the set A . The interpretations of various type constructions is clear from Table 1. In fact, this interpretation is what is used to motivate the various type constructions in Chapter 2. For example, the dependent function types are a generalization of functions on sets, the dependent product types are a generalization of cartesian products on sets, the empty type corresponds to the empty set and the unit type to the singleton, the type of natural numbers to the set of natural numbers and so on.

3.2 Types as Propositions

In the interpretation of types as propositions, $x : A$ is interpreted as x is a proof of proposition A . Therefore, if a proposition A is true in the context Γ , then in the type theoretic setting, this means that the type A is inhabited in the context Γ .

Considering propositions as types is a very important step in this foundation as it places propositions (and theorems in particular) and proofs of propositions in the same footing as any other mathematical object like natural numbers, sets or functions. Thus, unlike in current mathematics, where a proof is just a way of talking about (or establishing) properties of mathematical objects that we study such as groups, topological spaces, etc.; in this foundation, a proof is itself an abstract mathematical object.

To take an example of a proof-theoretic interpretation of types, consider say the non-dependent function type, denoted as $A \rightarrow B$. A non-dependent function is our usual notion of functions. In the type theoretic setting, it is just a term of a dependent function type $\prod_{(x:A)} B$ where B is a constant family over A .

In the proof-theoretic interpretation, where A and B are propositions, $A \rightarrow B$ is just the proposition $A \implies B$ (A implies B). The terms of $A \rightarrow B$ are just proofs of the proposition $A \implies B$. Now, consider the introduction rule for non-dependent types which is obtained from the corresponding rule for dependent types:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \rightarrow B} \quad \text{\Pi-INTRO} \quad (3.1)$$

This just says that if for every proof x of A , we have a proof b of B , then we have a proof of $A \implies B$ called $\lambda(x : A).b$. This is what it means to prove an implication, i.e., whenever A is true, B must be true.

Let's also look at the elimination rule :

$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \quad \Pi\text{-ELIM} \quad (3.2)$$

This is read as, if we have a proof f of $A \implies B$ and we have a proof a of the proposition A , then we have a proof of B called $f(a)$. In simpler words, if $A \implies B$ is true and A is true, then B is true. This is the rule of logic called *Modus ponens*.

In this manner, the various types that we have described earlier and their rules of inference can be interpreted in the proof-theoretic framework according to Table 1. Notice that the mathematics this will give rise to is proof-relevant. Proofs in this setting are abstract mathematical objects and with this comes the notion of an algebra of proofs as we have encountered in the above discussion.

The set-theoretic model of types is not a satisfactory one as there are a lot of types and type constructions like the identity type, which do not have any natural or canonical interpretation in sets. The model of types as propositions is by itself, obviously, insufficient to describe the informal mathematics that we do.

The recent development that has lead to a flurry of activity in this area is the observation that the interpretation of types as ∞ -groupoids or spaces upto homotopy is a very good model of types. Moreover, it can capture various aspects of modern mathematics like category theory very easily and therefore is a good candidate for a new computational foundation of mathematics.

3.3 The Homotopy Interpretation of Type Theory

In the homotopy interpretation of types (also called Homotopy Type Theory), types are regarded as spaces upto homotopy. Terms of types are then points in the space. But, why to model types using homotopy theory? What has homotopy theory to do with the foundations of mathematics?

Before getting into the details , let us first consider a brief philosophical answer to the above question. Homotopy type theory is a geometric foundations of mathematics. One way to look at sets is as the simplest geometric objects, namely a disjoint union of points. In set theoretic mathematics, we assume sets as primitive undefined and then describe everything else, including homotopy theory, using it. In homotopy type theory, we assume not just sets but higher geometric objects such as spheres aslo as primitve undefined notions. This enables us to capture higher mathematics such as category-level mathematics very easily as compared to the set-theoretic foundations.

A more direct answer to the initial question comes from the analysis of the identity type in homotopy type theory. In the homotopy theoretic interpretation, a type A is a space upto homotopy. $a : A$ is interpreted as a is a point in the space A . Now, given two points $a, b : A$, we can form the identity type $a =_A b$. We interpreted this type as the proposition ‘ $a = b$ as terms of type A ’. In homotopy type theory, $a =_A b$ is the space of all paths between a and b and a term of this type, $p : a =_A b$ is a path from a to b . Thus, the equality of terms of a type correspond to the existence of a path between the points in the homotopy interpretation.

But, now consider two paths p and q from a to b , i.e., $p, q : a =_A b$. Since, $a =_A b$ is itself a type, we can form the type $p =_{(a=_A b)} q$. This is then the space of paths between the paths p and q or in other words, it the the space of homotopies between the paths p and q . So, terms $r : p =_{(a=_A b)} q$ of this type are homotopies. Now we can continue this (consider $r, s : p =_{(a=_A b)} q$) to obtain homotopies between homotopies and so on. Thus, we obtain the following correspondence :

Type Theory		Homotopy Theory
A is a type	\longleftrightarrow	A is a space upto homotopy
$a : A$	\longleftrightarrow	a is a point of A
$p : a =_A b$	\longleftrightarrow	p is a path in A from a to b
$r : p =_{(a=_A b)} q$	\longleftrightarrow	r is a homotopy from path p to path q
	\vdots	

It is important to note that when we say that types are spaces upto homotopy, we assume no information about the topology of the space and only consider spaces upto homotopy equivalence. Also, points, paths, homotopies, homotopies between homotopies, etc. are all primitive undefined notions. So, for example, in this interpretation a path is not a collection of points, it itself is a primitive notion.

3.3.1 Types as ∞ -groupoids

A groupoid is a category where all morphisms are isomorphisms. A 2-category is a category which consists of objects, 1-morphisms between objects and 2-morphisms between 1-morphisms satisfying some additional axioms. Category of categories is an example of a 2-category with objects as categories, 1-morphisms as functors between categories and 2-morphisms as natural transformations between functors.

We can extend this notion to an n -category which would consist of objects, 1-morphisms, \dots , n -morphisms. However, no algebraic definition of an n -category exists because of the difficulty in stating the additional axioms. It is easy to see how this notion can be further extended to that of an ∞ -category. An ∞ -category in which all the morphisms can be inverted is an ∞ -groupoid.

Now, a topological space is an ∞ -groupoid with objects as the points of the space, 1-morphisms as paths between points, 2-morphisms as homotopies between paths, 3-morphisms

as homotopies between homotopies and so on. The composition of morphisms is the concatenation of paths and since all paths are invertible with respect to concatenation, every topological space has an ∞ -groupoid structure associated to it.

Moreover, an idea or philosophy that can be traced back to Alexander Grothendieck says that every ∞ -groupoid is “essentially” a topological space in the above sense.

Therefore, every type can also be interpreted as an ∞ -groupoid as follows :

<u>Type</u>	\longleftrightarrow	<u>∞-groupoid</u>
A is a type	\longleftrightarrow	A is ∞ -groupoid
$a : A$	\longleftrightarrow	a is an object of A
$p : a =_A b$	\longleftrightarrow	p is a 1-morphism from a to b
$r : p =_{(a=_A b)} q$	\longleftrightarrow	r is a 2-morphism from p to q
	\vdots	

We now show a bit more rigourously that types are indeed groupoids with respect to the axioms and rules of formal type theory stated in Chapter 2.

The first proposition in this direction ascertains that all morphisms are invertible. Note that we need to prove the invertibility of 1-morphisms only as 2-morphisms of A are 1-morphisms of the type $a =_A b$ and so on.

Proposition 3.3.1. *For every type A and every $x, y : A$, there is a function from $(x = y) \rightarrow (y = x)$ written as $p \mapsto p^{-1}$, such that $\text{refl}_x^{-1} \equiv \text{refl}_x$ for every $x : A$. We call p^{-1} the **inverse** of p .*

Since, this is the first proposition or theorem that we are encountering, let us see what it corresponds to in formal type theory. Recall that every proposition is identified with a

type in type theory and proving the proposition corresponds to constructing a term of the associated type.

Using the correspondence stated in Table 1, it is easy to see that the above proposition corresponds to the type :

$$\prod_{(A:\mathcal{U})} \prod_{(x,y:A)} (x = y) \rightarrow (y = x)$$

The proof of Proposition 3.3.1 will therefore consist of inhabiting the above type, i.e., deriving the judgment $f : \prod_{(A:\mathcal{U})} \prod_{(x,y:A)} (x = y) \rightarrow (y = x)$ for some f .

We first prove this theorem informally and then convert it into a formal proof.

Proof. Let A be a type, then for every $x, y : A$ and $p : x = y$, we need to construct a term $p^{-1} : y = x$. By path induction of identity types (which is the elimination rule of identity types), it suffices to do this in the case when y is x and p is refl_x . But, when y is x , then both $(x = y)$ and $(y = x)$ are $(x = x)$ and if p is refl_x , then we can define refl_x^{-1} to be refl_x itself as we need a term of $(x = x)$. We have thus constructed p^{-1} for the “reflexivity” case and the general case follows by path induction and $\text{refl}_x^{-1} \equiv \text{refl}_x$ by construction.

A More Formal Proof : The proof is using path induction of identity types which is :

$$\frac{\Gamma, x : A, y : A, p : x =_A y \vdash C : \mathcal{U}_i \quad \Gamma, z : A \vdash c : C[z, z, \text{refl}_z/x, y, p] \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash p' : a =_A b}{\Gamma \vdash \text{ind}_{=_A}(x.y.p.C, z.c, a, b, p') : C[a, b, p'/x, y, p]} \quad =\text{-ELIM} \quad (3.3)$$

Let $A : \mathcal{U}$ and let $C : (\prod_{(x,y:A)} (x = y) \rightarrow \mathcal{U})$ be the type family defined by $C[x, y, p] := (y = x)$. C is a function which assigns to any $x, y : A$ and $p : x = y$ a type, namely $(y = x)$. Now, given $z : A$, we have a term of the type $C[z, z, \text{refl}_z]$ namely, refl_z as $C[z, z, \text{refl}_z]$ is

simply ($z = z$).

Therefore, by path induction, we have the term $ind_{=A}(x.y.p.C, z.refl_z, x, y, p) : C[x, y, p]$ or simplifying notation we have an element $ind_{=A}(C, refl_z, x, y, p) : (y = x)$ for each $p : x = y$. Thus, $p^{-1} \equiv ind_{=A}(C, refl_z, x, y, p)$.

The computation rule for identity types then gives us that $refl_x^{-1} \equiv refl_x$ as required.

$$\frac{\Gamma, x : A, y : A, p : x =_A y \vdash C : \mathcal{U}_i \quad \Gamma, z : A \vdash c : C[z, z, refl_z/x, y, p] \quad \Gamma \vdash a : A}{\Gamma \vdash ind_{=A}(x.y.p.C, z.c, a, a, refl_a) \equiv c[a/z] : C[a, a, refl_a/x, y, p]} =\text{-COMP} \quad (3.4)$$

□

Most informal proofs involving path induction of identity types can be formalized in a similar manner to that illustrated above. Hence, from now on, we shall write only the informal proofs keeping in mind that we can formalize the proofs, when required, using the rules stated in Chapter 2.

The next proposition defines the composition of morphisms in the ∞ -groupoid interpretation of types.

Proposition 3.3.2. *For every type A and every $x, y, z : A$, there is a function from $(x = y) \rightarrow (y = z) \rightarrow (x = z)$ denoted by $p \mapsto q \mapsto p \cdot q$, such that $refl_x \cdot refl_x \equiv refl_x$ for any $x : A$. $p \cdot q$ is called the **concatenation** or **composite** of p and q .*

Here, $(x = y) \rightarrow (y = z) \rightarrow (x = z)$ should be read as the type $(x = y) \rightarrow ((y = z) \rightarrow (x = z))$. A function of this type takes in two arguments $p : x = y$ and $q : y = z$ to give a term of the type $x = z$. Thus, $(x = y) \rightarrow (y = z) \rightarrow (x = z)$ can also be thought of as the type $((x = y) \times (y = z)) \rightarrow (x = z)$. This style of writing a function which takes multiple

arguments is called *Currying* after Haskell Curry.

Proof. We proceed again by using path induction. When x, y and z are all equal, say to x and p and q are both refl_x , then $(x = y) \rightarrow (y = z) \rightarrow (x = z)$ is just $(x = x) \rightarrow (x = x) \rightarrow (x = x)$ and $\text{refl}_x \mapsto \text{refl}_x \mapsto \text{refl}_x$ i.e. $\text{refl}_x \cdot \text{refl}_x \equiv \text{refl}_x$. The general case, then, follows from path induction. \square

The next proposition establishes refl_x as the identity morphism of x , verifies that p^{-1} is the inverse of p with respect to the identity and checks associativity.

Proposition 3.3.3. *Let $A : \mathcal{U}$ and $x, y, z, w : A$ and $p : x = y$, $q : y = z$ and $r : z = w$, then*

:

1. $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.
2. $p \cdot \text{refl}_y = p$ and $\text{refl}_x \cdot p = p$.
3. $p \cdot p^{-1} = \text{refl}_x$ and $p^{-1} \cdot p = \text{refl}_y$.
4. $(p^{-1})^{-1} = p$.

Proof. 1. By path induction, it is sufficient to assume that $x \equiv y \equiv z \equiv w$ and p, q, r are refl_x . In this case,

$$p \cdot (q \cdot r) \equiv \text{refl}_x \cdot (\text{refl}_x \cdot \text{refl}_x) \equiv \text{refl}_x \equiv (\text{refl}_x \cdot \text{refl}_x) \cdot \text{refl}_x \equiv (p \cdot q) \cdot r$$

Therefore, we have $\text{refl}_{\text{refl}_x}$ inhabiting $p \cdot (q \cdot r) = (p \cdot q) \cdot r$ in the above case.

2. Assume that $p \equiv \text{refl}_y$, then $p \cdot \text{refl}_y = p$ since we have $\text{refl}_{\text{refl}_y} : \text{refl}_y \cdot \text{refl}_y = \text{refl}_y$ by path induction. The other case is similar.
3. Assume that $p \equiv \text{refl}_x$, then $p^{-1} \equiv \text{refl}_x^{-1} \equiv \text{refl}_x$ and $\text{textrefl}_{\text{refl}_x} : \text{refl}_x \cdot \text{refl}_x^{-1} = \text{refl}_x$. Therefore, $p \cdot p^{-1} = \text{refl}_x$ by path induction. The other case is similar.

4. Assume that $p \equiv \text{refl}_x$. It is clear that $(p^{-1})^{-1} \equiv \text{refl}_x$ and the result follows from path induction.

□

To summarize, the notion of equality (or propositional equality to distinguish it from definitional equality) is interpreted in the homotopical and ∞ -groupoid view of types as follows :

Equality	Homotopy	∞ -Groupoid
Reflexivity	constant path	identity morphism
Symmetry	inversion of paths	inverse morphism
Transitivity	concatenation of paths	composition of morphisms

3.4 Function Extentionality

Let us now consider functions between types in homotopy type theory. A function $f : A \rightarrow B$ is interpreted as a continuous map from space A to space B or as an ∞ -functor from A to B . We will now establish that functions behave functorially on paths. What does this mean? We know that if $f : A \rightarrow B$ and $x : A$, then we have $f(x) : B$. Now, if $x = y$, i.e., we have $p : x = y$, then is $f(x) = f(y)$, i.e. is there some $f(p) : f(x) = f(y)$?

Proposition 3.4.1. *Suppose $f : A \rightarrow B$ is a function. Then, for any $x, y : A$, we have*

$$ap_f : (x =_A y) \rightarrow (f(x) =_B f(y))$$

such that for each $x : A$, we have $ap_f(\text{refl}_x) \equiv \text{refl}_{f(x)}$.

Proof. By path induction, it suffices to prove it for the case when $x \equiv y$ and p is refl_x . We can then define $ap_f(p) \equiv \text{refl}_{f(x)} : f(x) = f(x)$ and we are done. □

Proposition 3.4.2. *Given $f : A \rightarrow B$ and $g : B \rightarrow C$ and paths $p : x =_A y$ and $q : y =_A z$:*

$$1. \text{ ap}_f(p \cdot q) = \text{ap}_f(p) \cdot \text{ap}_f(q).$$

$$2. \text{ ap}_f(p^{-1}) = \text{ap}_f(p)^{-1}.$$

$$3. \text{ ap}_g(\text{ap}_f(p)) = \text{ap}_{g \circ f}(p).$$

$$4. \text{ ap}_{id_A}(p) = p.$$

Proof. Similar to above proofs and therefore, left to the reader. □

3.4.1 Type families as fibrations and Dependent functions as sections

A type family P parametrized by a type A is a function $P : A \rightarrow \mathcal{U}$ which assigns to every point $x : A$, a type $P(x)$. In the homotopy interpretation, P is a continuous map which attaches a space to every point of A . P is, therefore, a fibration with base space A .

Suppose $P : A \rightarrow \mathcal{U}$ is a fibration over A , then a dependent function $f : \prod_{(x:A)} P(x)$ is a function which assigns to every point $a : A$, a point in the fiber of a , that is $P(a)$. f is therefore a continuous section of the fibration. It is a continuous section since all functions in homotopy type theory correspond to continuous maps.

We think of the type family $P : A \rightarrow \mathcal{U}$ as a fibration over A , with $P(x)$ being the fiber over $x : A$ and with $\sum_{(x:A)} P(x)$ as being the total space of the fibration with first projection $\text{pr}_1 : \sum_{(x:A)} P(x) \rightarrow A$. It is easy to see why $\sum_{(x:A)} P(x)$ is the total space as every term of $\sum_{(x:A)} P(x)$ is a pair of the form (a, b) where $a : A$ and $b : P(a)$.

The following proposition then tells us that P respects equality.

Proposition 3.4.3. (*Transport*) Suppose P is a type family over A and $p : x =_A y$, then there exists function $p_* : P(x) \rightarrow P(y)$.

Proof. Formally, we need to construct a term of the type :

$$\prod_{(A:\mathcal{U})} \prod_{(P:A \rightarrow \mathcal{U})} \prod_{(x,y:A)} (x = y) \rightarrow (P(x) = P(y))$$

When $x \equiv y$ and $p \equiv \text{refl}_x$, then p_* can be defined as $p_* \equiv \text{id}_{P(x)}$, i.e. $p_*(z) \equiv z : P(x)$. This defines p_* in general by path induction. \square

Notation : p_* defined above is also denoted as $\text{transport}^P(p, -) : P(x) \rightarrow P(y)$.

Proposition 3.4.4. (*Path lifting property*). Suppose $P : A \rightarrow \mathcal{U}$ is a type family over A and suppose we have $u : P(x)$ for some $x : A$, then for any $p : x = y$, we have

$$\text{lift}(u, p) : (x, u) = (y, p_*(u))$$

in the type $\sum_{(x:A)} P(x)$, such that $\text{pr}_1(\text{lift}(u, p)) = p$.

Proof. Left to the reader. \square

Proposition 3.4.5. Suppose $f : \prod_{(x:A)} P(x)$ and $p : x = y$, then we have a map $p_* : P(x) \rightarrow P(y)$ as above. In this case, we have

$$\text{apd}_f(p) : p_*(f(x)) =_{P(y)} f(y)$$

More formally we have,

$$\text{apd}_f : \prod_{(p:x=y)} (p_*(f(x)) =_{P(y)} f(y))$$

Example 3.4.6. Fix $a : A$ and consider the type family $P : A \rightarrow \mathcal{U}$ defined by $x \mapsto (a =_A x)$. If $q : x =_A y$, then we have $q_* : (a = x) \rightarrow (a = y)$ such that $q_*(p) = q \cdot p$.

We now define when two dependent functions or two sections of a type family are homotopic.

Definition 3.4.7 (Homotopies between functions). Suppose $f, g : \prod_{(x:A)} P(x)$. A homotopy from f to g is a dependent function of the type

$$(f \sim g) :\equiv \prod_{(x:A)} (f(x) =_{P(x)} g(x))$$

Let us unravel this definition. Let $P : A \rightarrow \mathcal{U}$ be a type family over A such that $P(x)$ is the identity type $f(x) = g(x)$. A homotopy from f to g is a section of this family P . But, terms of $f(x) = g(x)$ are paths from $f(x)$ to $g(x)$. It is now easy to see how this corresponds to the definition of homotopy in homotopy theory.

So, f and g are homotopic if they are propositionally equal pointwise. But, we also have the equality between the terms f and g themselves as in the identity type $(f =_{(\prod_{(x:A)} P)} g)$. How are these two notions related? That is the content of the next proposition :

Proposition 3.4.8. Let $f, g : \prod_{(x:A)} P(x)$ be two sections of the type family P , then we have a map :

$$\text{happly} : (f = g) \rightarrow (f \sim g) \tag{3.5}$$

Proof. $(f \sim g) :\equiv \prod_{(x:A)} (f(x) =_{P(x)} g(x))$ and so we need to construct a term $\text{happly} : (f = g) \rightarrow \prod_{(x:A)} (f(x) =_{P(x)} g(x))$. This follows easily from path induction. \square

Now that we have defined homotopy between two maps, we move on to define the equivalence of two spaces A and B .

Definition 3.4.9. $f : A \rightarrow B$ is called an equivalence iff there exists a map $g : B \rightarrow A$ such that $f \circ g \sim id_B$ and $g \circ f \sim id_A$.

Theorem 3.4.10. *$f : A \rightarrow B$ is an equivalence iff there exist maps $g : B \rightarrow A$ and $h : B \rightarrow A$ such that $f \circ g \sim id_B$ and $h \circ f \sim id_A$.*

For reasons that we will not go into at this stage, we choose the latter as the definition of equivalence when working in homotopy type theory. So, we define the type $isequiv(f)$ as

$$isequiv(f) := \left(\sum_{(g:B \rightarrow A)} (f \circ g \sim id_B) \right) \times \left(\sum_{(h:B \rightarrow A)} (h \circ f \sim id_A) \right) \quad (3.6)$$

Thus, the type $isequiv(f)$ is inhabited precisely when f is an equivalence.

Now, Propostion 3.4.7 tells us that when two functions are equal, then they are equal pointwise (i.e., they are equivalent). But, we also want that if two functions are equal pointwise, then they must be equal. Here, all equalities are propositional. This does not follow from whatever we have stated so far. We, therefore, assume it as an axiom called function extensionality. This is the first additional axiom that we impose in homotopy type theory.

Axiom 3.4.11 (Function Extensionality). *For any types A, B and maps $f, g : \prod_{(x:A)} B$, the function $happly : (f = g) \rightarrow (f \sim g)$ that we defined in Proposition 3.4.7 is an equivalence.*

Formally, it corresponds to the rule of inference :

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \quad \Gamma \vdash g : \prod_{(x:A)} B}{\Gamma \vdash funext(f, g) : isequiv(happly_{f,g})} \quad \Pi\text{-EXT}$$

3.5 Univalence Axiom

Given the definition of when a function is an equivalence, when are two types A and B equivalent? A is equivalent to B when there exists an equivalence $f : A \rightarrow B$. Therefore,

$$(A \simeq B) := \sum_{(f:A \rightarrow B)} \text{isequiv}(f) \quad (3.7)$$

Recall that

$$\text{isequiv}(f) := \left(\sum_{(g:B \rightarrow A)} (f \circ g \sim \text{id}_B) \right) \times \left(\sum_{(h:B \rightarrow A)} (h \circ f \sim \text{id}_A) \right)$$

Just like in the case of function, given two types A and B , we may consider them as terms of some universe \mathcal{U} and so we have the type $A =_{\mathcal{U}} B$. How is this related to $(A \simeq B)$?

Theorem 3.5.1. *Given types $A, B : \mathcal{U}$, there is a function,*

$$\text{idtoeqv} : (A =_{\mathcal{U}} B) \rightarrow (A \simeq B) \quad (3.8)$$

defined by $p \mapsto \text{transport}^{(A \rightarrow A)}(p, -)$.

We would like to say that idtoeqv is an equivalence. But, just as in the case of happily for function types, this is not guaranteed from whatever we have developed so far. We, therefore, assume it as an axiom : Voevodsky's **Univalence Axiom**.

Axiom 3.5.2 (Univalence). *Given types $A, B : \mathcal{U}$, the function $\text{idtoeqv} : (A =_{\mathcal{U}} B) \rightarrow (A \simeq B)$ is an equivalence.*

Formally,

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash \text{univalence}(A, B) : \text{isequiv}(\text{idtoeqv}_{A,B})} \quad \mathcal{U}_i\text{-UNIV}$$

Another way to state the univalence axiom is that given $A, B : \mathcal{U}$, there exists a map

$$\text{ua} : (A \simeq B) \rightarrow (A = B)$$

such that :

1. $\text{transport}^{(A \rightarrow A)}(\text{ua}(f), -) = f$ and
2. $\text{ua}(\text{idtoeqv}(p)) = p$.

In particular, univalence means that equivalent types can be identified. This something that we have been doing in informal mathematics all along, for example, identifying isomorphic groups or homeomorphic topological spaces. We can now formally justify this.

Chapter 4

Sets and Logic

In this chapter, we continue exploring the ∞ -groupoid structure of a type. Although, we have a proposition-as-types interpretation, is the logic of type theory the same as the logic of set theory that we commonly use in our informal mathematics? Although, we can interpret types as sets, what precisely is the notion of a set in homotopy type theory? Are all type sets? These are some of the questions that we will be addressing in this chapter.

4.1 Sets and n -type

Let us first understand the notion of a set in homotopy type theory. Although, in homotopy type theory, types behave like ∞ -groupoids, there are a class of types which behave more like sets as in the traditional set-theoretic system. Categorically, sets can be thought of as discrete groupoids where we have a set of objects and only identity morphisms between them. Topologically, sets are spaces with the discrete topology on them.

Since, whatever we do in homotopy type theory is upto equivalence, sets can be thought of as a disjoint union of connected components with no higher homotopical information. This gives us the definition :

Definition 4.1.1. *A type A is a set if for all $x, y : A$, any two paths $p, q : x =_A y$ from x to*

y are homotopic, i.e., we have $p = q$.

Formally, the proposition $\text{isSet}(A)$ is defined as :

$$\text{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q) \tag{4.1}$$

Example 4.1.2. 1. *The empty type $\mathbf{0}$ is a set.*

2. *The unit type $\mathbf{1}$ is a set.*

3. *The type \mathbb{N} of natural numbers is also a set.*

In the definition of a set, all additional structure from 2-morphisms onwards is trivial. In other words, if we take a type and ignore all structure from 2-morphisms onwards, then we get a set. However, we can do this not just at the level of 2-morphisms, but after any level. This gives us the general definition of an n -type.

Definition 4.1.3. 1. *A type A is -1 -type if for all $x, y : A$, $x = y$.*

2. *A type A is a $(n + 1)$ -type if for all $x, y : A$, $x = y$ is an n -type, where n is an integer such that $n \geq -1$.*

In the homotopy interpretation of type theory, this translates to the following recursive definition :

- A type T is a (-2) -type if it is contractible.
- A type T is a $(n + 1)$ -type if for every $t_1, t_2 : T$, the path space in T from t_1 to t_2 is an n -type.

Let us unravel the above definition for some initial cases :

A type A is (-1) -type if for all $x, y : A$, $x = y$. So, there is no structure from the level of 1-morphisms itself.

Upto homotopy equivalence, there are only two (-1) -types : the empty type and the unit type because either it has no terms at all in which case the condition is vacuously satisfied or if there is atleast one term, all the terms are equal. These correspond to the booleans 0 and 1 or truth values *true* and *false*. Thus, all propositions in classical first order logic are equivalent to either true (\top) or false (\perp) and just as we defined $\text{isSet}(A)$, we can define :

$$\text{isProp}(A) :\equiv \prod_{(x,y:A)} (x = y) \tag{4.2}$$

Thus, propositional logic is essentially homotopy type theory at level -1 .

A type A is 0-type if for all $x, y : A$, $x = y$ is a (-1) -type, i.e., for all $x, y : A$ and $p, q : x = y$, $p = q$. Therefore, 0-types are precisely sets and what constitutes set-theoretic mathematics is essentially homotopy type theory at level 0.

A type A is 1-type if for all $x, y : A$ and $p, q : x = y$ and $r, s : p = q$, we have $r = s$. Thus, when interpreted as an *infty*-groupoid, there are objects of A and 1-morphisms between objects. All higher structure is trivial. Therefore, what constitutes as category-theoretic mathematics is essentially homotopy type theory at level 1.

Similarly we can define 2-types, 3-types, and so on. Mathematics at level 2 is 2-categorical mathematics, that at level 3 is 3-categorical, and so. This is how the univalent foundations captures and formalizes categorical and higher-categorical mathematics.

Lemma 4.1.4. *Let $A : \mathcal{U}$ be a type, then $\text{isProp}(\text{isSet}(A))$ is inhabited.*

Proposition 4.1.5. *Let $A : \mathcal{U}$ be a type, then if A is a -1 -type, then A is a 0 -type.*

Proof. We essentially need to construct a map of the type $\text{isProp}(A) \rightarrow \text{isSet}(A)$. Suppose $f : \text{isProp}(A) \equiv \prod_{(x,y:A)}(x = y)$. Fix an $x : A$ and define $g(y) := f(x, y)$. Here, $g : \prod_{(y:A)}(x = y)$.

Now consider some $p : y = z$, then we have $\text{apd}_g : p_*(g(y)) = g(z)$. But, $p_*(g(y)) = g(y) \cdot p$ (Example 3.4.6). So, $g(y) \cdot p = g(z)$. Now, if we consider some $q : y = z$, then we have $g(y) \cdot q = g(z)$. But, this implies that $p = q$. Since, y, z are arbitrary, we are done. □

Proposition 4.1.6. *Let $A : \mathcal{U}$ be a type, then if A is a 0 -type, then A is a 1 -type.*

Proof. Similar to the above proof. □

Most generally, we have the following theorem :

Theorem 4.1.7. *A is an n -type $\implies A$ is a $n + 1$ -type.*

4.2 Intuitionistic Logic

We have so far used the proposition-as-types philosophy to interpret and write down and prove informal propositions in formal type theory (See Table 1). But, does this notion of a proposition behave in the same way as whatever we call propositions in classical mathematics? The logic of propositions that arises out of the rules of inference of formal types is significantly different than the classical logic used in set theoretic mathematics. It is intuitionistic.

One of the key features of intuitionistic logic is that we do not have the law of excluded middle. This has many consequences, one of which is that we do not have the usual proof by contradiction as a valid proof technique in the logic of type theory.

Recall that in the interpretation of propositions as types, a term of the type corresponds to a proof of the proposition. Therefore, classically, a proposition is true if the type associated with it is inhabited. We now define the negation of a proposition in this setting :

Definition 4.2.1. *Let $A : \mathcal{U}$ be a type, then the negation of A is the type :*

$$\neg A :\equiv (A \rightarrow \mathbf{0}) \tag{4.3}$$

Thus, $\neg A$ is the type $A \rightarrow \mathbf{0}$ which corresponds to the proposition $A \implies \perp$, i.e., A implies *false*. Therefore, if $\neg A$ is inhabited, then A cannot be inhabited because if it is, then the empty type is inhabited which implies that all types are inhabited (or all propositions are true) leading to triviality.

We now show that proof by contradiction or the law of double negation elimination does not hold in general. We need the following lemma :

Lemma 4.2.2. *1. If $x, y : \mathbf{1}$, then $(x = y) \simeq \mathbf{1}$.*

2. \mathbb{N} is a set.

3. Consider the type $A + B : \mathcal{U}$. If $a : A$ and $b : B$, then $(inl(a) = inr(b)) \simeq \mathbf{0}$.

Theorem 4.2.3. *It is not the case that for all $A : \mathcal{U}$, we have $\neg\neg A \rightarrow A$.*

Proof. For all $A : \mathcal{U}$, $\neg\neg A \rightarrow A$ corresponds to the type $\prod_{(A:\mathcal{U})} (\neg\neg A \rightarrow A)$. We need to show that the above proposition is false, i.e., we need to construct a map from the above type to the empty type which is a term of the type :

$$\left(\prod_{(A:\mathcal{U})} (\neg\neg A \rightarrow A) \right) \rightarrow \mathbf{0}$$

Suppose $f : \prod_{(A:\mathcal{U})} (\neg\neg A \rightarrow A)$, then we need to produce a term of the empty type.

First, observe that for any $A : \mathcal{U}$, $\neg A$ is a proposition type or $\text{isProp}(\neg A)$ is inhabited. This is easy to see. If $u, v : \neg A \equiv (A \rightarrow \mathbf{0})$, then to show that $u = v$, it is enough to show that $u(x) = v(x)$ for all $x : A$ by function extentionality. But, $u(x) : \mathbf{0}$ is a term of the empty type. Thus, we can construct a term of any type using $u(x)$, in particular, the type $\prod_{(x:A)}(u(x) = v(x))$.

Let $\mathbf{2}$ be the type $\mathbf{2} := \mathbf{1} + \mathbf{1}$ which is like $\mathbf{1} \amalg \mathbf{1}$ and let $0_{\mathbf{2}} : \mathbf{2}$ and $1_{\mathbf{2}} : \mathbf{2}$.

Now, let $e : \mathbf{2} \rightarrow \mathbf{2}$ such that $e(0_{\mathbf{2}}) = 1_{\mathbf{2}}$ and $e(1_{\mathbf{2}}) = 0_{\mathbf{2}}$. Since, $e \circ e \sim \text{id}_{\mathbf{2}}$, e is an equivalence.

Note that we have a term of the type $\prod_{(x:\mathbf{2})} \neg(e(x) = x)$, i.e., for all $x : \mathbf{2}$, $e(x) \neq x$.

Since, e is an equivalence, by the univalence axiom, we have $\text{ua}(e) : \mathbf{2} = \mathbf{2}$. Let us call $\text{ua}(e)$ as p , then we have $p_*(f(\mathbf{2})) = f(\mathbf{2})$. Recall that $f(\mathbf{2}) : \neg\neg\mathbf{2} \rightarrow \mathbf{2}$. So, for every $u : \neg\neg\mathbf{2}$, we have $p_*(f(\mathbf{2}))(u) = f(\mathbf{2})(u)$.

But, $p_*(f(\mathbf{2}))(u)$ is also equal to $e(f(\mathbf{2})(u))$. This can be seen from the fact that $\text{transport}^{(A \rightarrow \neg\neg A \rightarrow A)}(p, (f(\mathbf{2}))) (u) = \text{transport}^{(A \rightarrow A)}(p, (f(\mathbf{2})(u)))$.

Therefore, $e(f(\mathbf{2})(u)) = f(\mathbf{2})(u)$. However, we know that $\prod_{(x:\mathbf{2})} \neg(e(x) = x)$ and $\neg(e(x) = x) := (e(x) = x) \rightarrow \mathbf{0}$. Thus, we have a term of the empty type as required.

□

Corollary 4.2.4. *It is not the case that for all $A : \mathcal{U}$, we have $A + (\neg A)$.*

So, the logic of proposition-as-types is not classical. However, if we consider only those types A for which $\text{isProp}(A)$ holds, then we are restricting ourselves to the classical case. Such types, which are basically, (-1) -types are also called *mere propositions*. Just to reiterate :

Definition 4.2.5. A type P is a *mere proposition* if for all $x, y : P$, we have $x = y$.

Although, the logic of type theory is not classical, but intuitionistic, to work in classical logic, we can restrict ourselves to working with mere propositions and add the additional axioms of the law of excluded middle and the law of double negation.

The formulation of the law of excluded middle in homotopy type theory is :

$$\text{LEM} := \prod_{(A:\mathcal{U})} (\text{isProp}(A) \rightarrow (A + \neg A)). \quad (4.4)$$

Similarly, the formulation of the law of double negation elimination in homotopy type theory is :

$$\text{LEM} := \prod_{(A:\mathcal{U})} (\text{isProp}(A) \rightarrow (\neg\neg A \rightarrow A)). \quad (4.5)$$

Therefore, the univalent foundations can be used as a foundations for both constructive as well as non-constructive mathematics.

To conclude, the univalent foundations of mathematics which is based on homotopy type theory is a much richer foundation of mathematics than set theory. In this thesis, we have provided a snapshot view of the current state of development of this new foundation. The subject is evolving rapidly and the way this theory will look a few years down the line might be very different from its initial stages of development that is presented here. We hope that many more people will see the merit in pursuing this project which has the potential to change the entire perspective and practise of this beautiful subject called mathematics.

Bibliography

- [1] Voevodsky, Vladimir. “Homotopy type theory: Univalent foundations of mathematics.” Institute for Advanced Study (Princeton), The Univalent Foundations Program (2013) Book version: first-edition- 1005-ge9c58d7.
- [2] Pelayo, Ivaro, and Michael Warren. “Homotopy type theory and Voevodsky’s univalent foundations.” *Bulletin of the American Mathematical Society* 51.4 (2014): 597-648.
- [3] Kapulkin, Chris, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. “The simplicial model of univalent foundations.” arXiv preprint arXiv:1211.2851(2012).
- [4] <http://homotopytypetheory.org/> : ‘This site collects and disseminates research, resources, and tools for the investigation of homotopy type theory, and hosts a blog for those involved in its study.
- [5] Voevodsky, Vladimir. “Univalent foundations project.” NSF grant application (2010).