An overview of Homotopy type theory and Formal methods

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

Harsha Nyapathi

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

April, 2020

Supervisor: Dr. Siddhartha Gadgil

Certificate

This is to certify that this dissertation entitled An overview of Homotopy type theory and Formal methods 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 Harsha Nyapathi at Indian Institute of Science under the supervision of Dr. Siddhartha Gadgil, Professor, Department of Mathematics , during the academic year 2019-2020.

Dr. Siddhartha Gadgil

Committee:

Dr. Siddhartha Gadgil

Dr. Amit Hogadi

This thesis is dedicated to my well wishers

Declaration

I hereby declare that the matter embodied in the report entitled An overview of Homotopy type theory and Formal methods are the results of the work carried out by me at the Department of Mathematics, Indian Institute of Science, Bangalore, under the supervision of Dr. Siddhartha Gadgil and the same has not been submitted elsewhere for any other degree.

Harsha Nyapathi

Acknowledgments

I would firstly like to acknowledge my supervisor, Prof Siddhartha Gadgil for constantly encouraging me to do better and providing the required guidance. I was able to push my boundaries and learn a lot of new things during this year with his assistance. I would like to thank Prof Amit Hogadi for assuaging my concerns and guiding me during stressful situations. I express my gratitude to the people on Zulip chatroom, especially Alex J. Best, who helped me with my doubts in Lean. I would also like to thank the mathematics fifth year committee for supporting me with the project.

I am grateful for my family and friends, especially my roommates for their support during this challenging year. Lastly, I am indebted to IISER for making the last 5 years memorable.

Abstract

In this project, I have studied the basics of type theory, as a foundation for mathematics and a basis for theorem provers, using which we can formalize mathematical objects and proofs, and check their validity as well. I have taken up an example of representing graphs in type theoretic language to understand the formalization better. I have also studied homotopy type theory as an extension of Intuitionistic type theory, with the addition of univalence axiom and higher inductive types.

Contents

Abstract **xi**

1 Type Theory **3**

1.1 History and Variations 3

1.2 Type theory vs Set theory 5

1.3 Context and Structural rules 6

1.4 Type Universes 9

1.5 Dependent Types 10

1.6 Inductive types 12

2 Logic and Proof **21**

2.1 Propositions as types 21

2.2 Propositional logic 22

2.3 Predicate Logic 26

2.4 Classical vs Constructive logic 29

3 Example - Formalization of graphs **33**

4 Homotopy Type Theory **37**

4.1 Types as higher groupoids 38
4.2 Function extensionality and Univalence axiom 40
4.3 Sets and Logic 43

Introduction

The original idea of type theory was given by Bertrand Russell, to resolve paradoxes in mathematical foundations. It has developed over time with contributions from people like Alonzo Church and Per Martin-Löf. Currently, there exist many variations of type theory, important ones being Calculus of Constructions(CoC) and Intuitionistic type theory(ITT). They are based on a primitive notion of type, which have rules specifying how to introduce objects and use them. They have a vast structure to formalize mathematical objects and proofs and be regarded as a foundation for mathematics. Type theory also has important applications in computer science and programming languages, and more importantly in proof assistants.

Proof assistants and theorem provers give logical and computational methods to express claims formally, prove them, and give a verification for the proofs. Type theory is the basis of many proof assistants because of its computational rules guiding construction of types and behavior of their objects. It is also the result of a natural correspondence between proofs and programs, as discovered by Haskell Curry and William Howard. Lean theorem prover is a bridge between automated and interactive theorem proving,i.e., it provides tools to assist in finding proofs and it focuses on verification of these proofs as well by justifying every step against prior axioms. It is based on CoC.

Homotopy type theory is a recent idea, developed independently by the works of Awodey, Warren and Voevodsky. It explores the intimate connections between abstract homotopy theory, type theory, and category theory. Spaces can be thought of as groupoids with points as objects and paths as morphisms. The non trivial structure of identity type motivates a connection between types and spaces. The rules for identity type show that regarding elements of a type as points and proofs of equality between them as paths is consistent with

the structure of weak ∞- groupoid. It gives rise to further connections between functions and functors, dependent families and fibrations and so on. Hence homotopy type theory gives a different interpretation of type theory and extends it by introducing univalence axiom and higher inductive types. This provides powerful tools for formalizing constructive mathematics and is still an active area of research. A number of proof assistants like Coq are based on homotopy type theory implementation. Even Lean has a kernel supporting the theory.

I have studied the concepts of type theory in this project, with a main focus on formal verification. To understand it practically, I have used the Lean theorem prover and attempted to formalize graphs and connectivity problem. The goal was to learn to represent objects in type theoretic language, and also to understand computability in the case of finite and infinite graphs. I have also gained a brief introduction to homotopy type theory by learning the concepts necessary to understand the statement of univalence axiom and incorporation of classical logic in homotopy type theory. The primary references for this project are the tutorial on " Theorem proving in Lean" and " Homotopy type theory : Univalent foundations of mathematics".

Chapter 1

Type Theory

1.1 History and Variations

Type theory is a formal system which can serve as foundation of mathematics. It emerged as an alternative to set theory because of contradictions such as Russell's paradox. So, in simple terms, types form a hierarchy, hence avoiding self reference. There are many versions of type theory, well known among those are λ calculus and its generalizations, calculus of constructions and intuitionistic type theory.

- **Simple type theory** consists of untyped and simply typed λ calculus. λ calculus was created by Church to deal with recursion and computation in halting problem.

  Untyped λ calculus uses λ terms (or terms) as primitives instead of sets and elements. If x is a variable and E is an arbitrary term, then any term is of the form

  $$\mathrm{E} := \mathrm{x} \mid \lambda x.\mathrm{E} \mid (\mathrm{EE}) \mid ..$$

  The terms can be reduced using α-conversion and β-reduction operations. Strategies like call by name and call by value can be applied during reduction. However, if a term reaches normal form, the form is independent of the strategy.

  Simply typed λ calculus is based on λ calculus and uses the same syntax but it is a higher order logic system. There is a set of base types and new types can be built

using type constructor $\rightarrow$ or $\times$. Hence, given any arbitrary variable x, term E, types α and β, and base type γ, any type or term is of the form

$$\alpha, \beta := \gamma \mid \alpha \rightarrow \beta \mid \alpha \times \beta$$

$$\text{E} := \text{x} \mid \lambda x : \alpha.\text{E} \mid (\text{EE}) \mid (\text{E,E})$$

Typing judgements and rules are used to derive well formed and well typed terms.

- **Calculus of constructions** (CoC) is an impredicative type theory. When a definition quantifies over a set containing the entity which is being defined, it is impredicative. CoC is a generalized type system, meaning judgements over types are allowed. There is a type T (type of large types) and P (Prop or type of all propositions, or small types), which can be treated as terms as well. Hence, given variable x and arbitrary term E, any term is of the form

  $$\text{E:=}\ \text{T} \mid \text{P} \mid \text{x} \mid \text{EE} \mid \lambda\ \text{x:}\ \text{E.E} \mid \forall\ \text{x:E.E}$$

  CoC can be thought of as an extension of Curry Howard isomorphism, where types are interpreted as predicates and terms as proofs. We can also build basic data types such as bool, nat, product and union within CoC. Hence CoC is the basis of many proof assistants.

- **Intuitionistic type theory** (or constructive type theory, or Martin-Löf type theory) is, like CoC, a generalized type system, but it is predicative. Philosophically, it is a foundation for constructive mathematics as it gives a general idea of what constructive mathematical objects and proofs are. Homotopy type theory is a variant which employs ideas from ITT and homotopy theory. ITT has an extensional and intensional version which differ in the distinction between judgemental and propositional equality.

In the upcoming sections, the basic concepts of intuitionistic type theory (which is a version of Martin Löf's intensional type theory) will be explained. However, I have also learned formal proving in Lean theorem prover, which is based on Calculus of Inductive constructions. Hence the corresponding concepts of Lean will be mentioned wherever applicable.

1.2 Type theory vs Set theory

Type theory emerged as an alternative to set theory. Although type is in some ways similar to set, there are some important differences.

- Any foundation based on set theory is built on top of deductive systems such as first order logic. The axioms of a particular theory are formulated inside the system. Hence it has two layers, sets (and interplay between them) and propositions.

  However, type theory is a deductive system in its own right. It has a basic notion of types and propositions can be interpreted as particular types.

- Judgements are statements which are derived from the rules of deductive system. In set theory, first order logic has only one type of judgement- that any given proposition has proof. In type theory, there are three kinds of judgements, the basic one being "a : A", meaning term "a" has type "A". When "A" represents a proposition, it is analogous to saying "a" is a proof of "A".

  If we look at types as sets, then a : A corresponds to a $\in$ A. However, the important difference between these statements is that first is a judgement whereas second is a proposition. We cannot disprove a : A. Another way to think is, given an element "a", it may or may not belong to A in set theory. However in type theory, "a" cannot exist in isolation.

- In set theory, equality is a proposition. In type theory, equality is a type, since propositions are types. Hence a = b being inhabited implies "a" and "b" are propositionally equal. However there also exists a judgement of definitional equality, a $\equiv$ b : A, which is absent in set theory. It implies that the terms are equal by definition. Definitional equality is useful to derive judgements of the first kind, i.e, given a : A and A $\equiv$ B, we derive a : B.

Traditional type theories (excluding homotopy type theory) consist of rules without axioms, hence ensure computational properties by being procedural. In set theory, there are rules of first order logic but behavior of sets is described by axioms.

1.3 Context and Structural rules

The formal description of type theory includes giving syntax and a set of inference rules for deriving well formed judgements. We can have different presentations for the system, here we will describe one in the style of natural deduction.

Context is a list of assumptions of the form,

$$x_1 : A_1, x_2 : A_2, ..., x_n : A_n$$

which specifies that distinct $x_1, x_2, .., x_n$ have types $A_1, A_2, .., A_n$ respectively. Context can be empty as well.

There are three kinds of judgements:

- Γ ctx - It implies that Γ is a well formed context, meaning that each A_i is a type in $x_1 : A_1, x_2 : A_2, ..., x_i : A_i$ context.

- $\Gamma \vdash a : A$ - It implies that "a" has type A in the given context Γ.

- $\Gamma \vdash a \equiv b : A$ - It implies that "a" is judgementally equal to "b" and they have type A in the given context. Lean respects judgemental equality by having a definition command which is used to define objects.

An inference rule is of the form

$$\frac{\mathcal{J}_1 ..... \mathcal{J}_k}{\mathcal{J}} \text{ NAME}$$

It implies that given hypotheses (in the deductive system sense) $\mathcal{J}_1, .., \mathcal{J}_k$, we can derive $\mathcal{J}$.

The following are the rules to derive judgements.

- These rules are to derive well formed contexts, with an extra condition for second rule that x_n must be distinct from other variables.

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

$$\frac{x_1 : A_1, ..., x_{n-1} : A_{n-1} \vdash A_n : \mathcal{U}_i}{(x_1 : A_1, ..., x_n : A_n)ctx} \text{ ctx-EXT}$$

- This rule is to derive judgements of the second kind, only now the hypothesis and conditions are reversed compared to ctx-EXT.

$$\frac{(x_1 : A_1, ..., x_n : A_n)ctx}{x_1 : A_1, ..., x_n : A_n \vdash x_i : A_i} \text{ Vble}$$

- The following rules are to derive judgements of second and third kind, based on principles of substitution and weakening.

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]} \quad Subst_1$$

$$\frac{\Gamma \vdash A : U_i \qquad \Gamma, \Delta \vdash b : B}{\Gamma, x : A, \Delta \vdash b : B} \qquad Wkg_1$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash b \equiv c : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \equiv c[a/x] : B[a/x]} \quad Subst_2$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, \Delta \vdash b \equiv c : B}{\Gamma, x : A, \Delta \vdash b \equiv c : B} \quad Wkg_2$$

- Following rules state that judgemental equality is an equivalence relation.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A}$$

$$\frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A}$$

$$\frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a : B}$$

$$\frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash a \equiv b : B}$$

- In addition, there are rules to state that type constructors preserve definitional equality in all of their arguments.

We will follow a general pattern while introducing a new type by specifying:

- Formation rule - This tells us how we can form new types of a given kind.
- Introduction rule - This is to introduce new elements of the type. The rules are called type constructors as well.

- Elimination rules - They tell us how we can use elements of the type or construct functions out of the type. They are also called type's eliminators.
- Computational rule - This states how a function constructed by elimination rule acts when applied to an element constructed by introduction rule.
- Uniqueness principle (optional) - This tells us about uniqueness of functions in and out of the type.

1.4 Type Universes

Type universe represents the idea of having a type of all types. However, to avoid Russell's paradox, we have a hierarchy of universes,

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 ...$$

Universe is a type which contains types as elements. Here, each $\mathcal{U}_i$ is an element of $\mathcal{U}_{i+1}$ and if $A : \mathcal{U}_i$, then $A : \mathcal{U}_{i+1}$ as well. Hence universes are cumulative, which means elements don't have unique types.

This can be expressed formally as:

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

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

Type families ($B : A \to \mathcal{U}$) are functions,with codomain as universe, to represent collection of types which vary over a type A. When we write $\mathcal{U}$, it refers to some $\mathcal{U}_i$ without mentioning i explicitly. The subscript here doesn't refer to natural numbers, it is not a part of the theory.

Lean also has a hierarchy of type universes,

$$\text{Type } 1 : \text{Type } 2 : ...$$

When we declare A : Type, an implicit variable is created and Lean declares A : Type u.

The standard kernel for Lean contains a datatype Prop, which is the type representing all propositions. It is the syntactic sugar for Type 0 at the bottom of type hierarchy. All propositions, irregardless of the type of their arguments, are made to land in this type. Given a type A, it is allowed to form the type of all predicates on A, A $\to$ Prop, and propositions that quantify over A $\to$ Prop as well. This property of Prop makes CoC impredicative, on which the standard kernel of Lean is based on. The kernel supporting homotopy type theory doesn't have this feature, which is based on intuitionistic type theory.

1.5 Dependent Types

Elements of Π type or dependent function type are functions whose codomain is dependent on domain elements. If we are given a type $A : \mathcal{U}$ and a type family $B : A \to \mathcal{U}$ over A, then the type $\Pi_{(x:A)}B(x)$ represents type of functions which take $a : A$ as argument and return an element of the type $B(a)$.

In Lean, the syntax for dependent type is Π (x : A), B a. It can be expressed as a judgement as

$$x : A \vdash B : \mathcal{U}_i$$

The special case when B is a constant family gives rise to ordinary function types.

$$\Pi_{(x:A)}B \equiv A \to B$$

We can give the rules for the type former formally as follows:

- Formation rule

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

- Introduction rule

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

This represents that given an expression $b : B(x)$, we can introduce new elements of the type using λ-abstraction, $\lambda x.b : \Pi_{(x:A)}$ B.

- Elimination rule

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

We can apply the function to a given argument of type A, to obtain an element of $B(a)$.

- Computation rule

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

To compute $(\lambda x.b)(a)$, we replace all occurences of x in b with a.

- Uniqueness principle

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

  It states that a function is definitionally equal to the λ-abstraction - $\lambda x.f(x)$. It implies that a function is determined by its values uniquely.

An example of dependent type is the identity function $\Pi_{(A:\mathcal{U})} A \to A$, which is defined as id $\equiv \lambda(A : U).\lambda(x : A).x$. It is a polymorphic function, which takes type as an argument and acts on its elements.

1.6 Inductive types

Apart from universes and dependent function types, we can form other types using inductive definition. Intuitively, an inductive type is freely generated by a specified list of constructors. Each constructor is a function whose codomain is the inductive type. This includes the case of zero arguments as well.

The property that elements of inductive type are obtained by repeatedly applying constructors is expressed in the elimination rule, which is also called induction principle.

In Lean, the syntax for forming an inductive type is:

```
inductive f : Type
| constructor_1 : ... → f
| constructor_2 : ... → f
...
| constructor_n : ... → f
```

The index of Type to which f belongs is greater than each of the indices of types of arguments (excluding the case where f is made to land in Prop). To define functions on inductive types, Lean provides ways to use recursive functions, pattern matching and writing inductive proofs.

Some examples of inductive types in which constructors don't take arguments, and hence the types are just a collection of constructors are (as defined in Lean syntax) :

- `inductive empty : Type`

- ```
  inductive unit : Type :=
  star : unit
  ```

- ```
  inductive bool : Type :=
  | ff : bool
  | tt : bool
  ```

Following are some important examples of inductive type with constructors taking arguments.

1.6.1 Dependent pair types (or Σ type)

This type represents generalization of cartesian product of types, where elements are in the form of pairs, and the type of second component depends on the first. Product type is a special case of dependent pair type.

In Lean, the inductive definition for dependent pair type and product type respectively is as follows:

```
inductive sigma (A : Type) (B : A → Type)
dpair : Π a : A, B a → sigma B

inductive prod ( A B : Type)
mk : A → B → prod A B
```

The type former rules can be given formally as:

- Formation rule

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Sigma_{(x:A)} B : \mathcal{U}_i} \Sigma - FORM$$

  It says that given $A : \mathcal{U}$ and type family $B : A \to \mathcal{U}$, we can form a type $\Sigma_{(x:A)} B(x) : \mathcal{U}$, also denoted as $\Sigma(x : A), B(x)$. When B is a constant family, it is the special case of the product type $(A \times B)$.

- Introduction rule

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

  It says that given $a : A$ and $b : B(a)$, we can introduce an element (a, b) of the type.

- Elimination rule

$$\frac{\begin{array}{c}\Gamma, z : \Sigma_{(x:A)} B \vdash C : \mathcal{U}_i \\ \Gamma, x : A, y : B \vdash g : C[(x, y)/z] \qquad \Gamma \vdash p : \Sigma_{(x:A)} B\end{array}}{\Gamma \vdash ind_{\Sigma_{(x:A)} B}(z.C, x.y.g, p) : C[p/z]} \quad \Sigma - \text{ELIM}$$

  The syntax $z.C$ and $x.y.g$ imply that these variables are bound.

- Computational rule

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

The last two rules say that given a type family $C : (\Sigma_{(x:A)} B(x)) \to U$ and a function g : $\Pi_{(a:A)} \Pi_{(b:B(a))} C((a,b))$, we can construct a function

$$ind_{\Sigma_{(x:A)} B(x)} : \prod_{(C:(\Sigma_{(x:A)} B(x)) \to U)} (\Pi_{(a:A)} \Pi_{(b:B(a))} C((a,b))) \to \prod_{(p:\Sigma_{(x:A)} B(x))} C(p)$$

which follows the computation rule,

$$ind_{\Sigma_{(x:A)} B(x)}(C, g, (a,b)) :\equiv g(a)(b)$$

When C is a constant and not a type family, the elimination rule is called the recursion principle.

1.6.2 Natural numbers

Natural numbers are an example of inductive type where constructor uses the type being defined as an argument. The elements are defined using and initial element zero, and a successor function.

In Lean syntax, the definition for natural numbers is:

```
inductive nat : Type
|zero : nat
|succ : nat → nat
```

The formal rules specifying the type are:

- Formation rule

$$\frac{\Gamma ctx}{\Gamma \vdash N : \mathcal{U}_i} \quad N - \text{FORM}$$

- Introduction rule

$$\frac{\Gamma ctx}{\Gamma \vdash 0 : N} \quad N - INTRO_1$$

$$\frac{\Gamma \vdash n : N}{\Gamma \vdash succ(n) : N} \quad N - INTRO_2$$

- Elimination rule

$$\frac{\begin{array}{c}\Gamma, x : N \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_0 : C[0/x] \\ \Gamma, x : N, y : C \vdash c_s : C[succ(x)/x] \qquad \Gamma \vdash n : N\end{array}}{\Gamma \vdash ind_N(x.C, c_0, x.y.c_s, n) : C[n/x]} \quad N - \text{ELIM}$$

- Computation rule

$$\frac{\begin{array}{c}\Gamma, x : N \vdash C : \mathcal{U}_i \\ \Gamma \vdash c_0 : C[0/x] \qquad \Gamma, x : N, y : C \vdash c_s : C[succ(x)/x]\end{array}}{\Gamma \vdash ind_N(x.C, c_0, x.y.c_s, 0) \equiv c_0 : C[0/x]} \quad N - COMP_1$$

$$\frac{\begin{array}{c}\Gamma, x : N \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c_0 : C[0/x] \\ \Gamma, x : N, y : C \vdash c_s : C[succ(x)/x] \qquad \Gamma n : N\end{array}}{\begin{array}{l}\Gamma \vdash ind_N(x.C, c_0, x.y.c_s, succ(n)) \\ \quad \equiv c_s[n, ind_N(x.C, c_0, x.y.c_s, n)/x, y] : C[succ(n)/x]\end{array}} \quad N - COMP_2$$

The last two rules state that, to use the induction principle on nat onto a type family $C : N \to \mathcal{U}$, we need an element $c_0 : C(0)$ and a function $c_s : \Pi_{(n:N)} C(n) \to C(succ(n))$. Then we can construct a function,

$$ind_N : \prod_{(C:N \to U)} C(0) \to (\Pi_{(n:N)} C(n) \to C(succ(n))) \to \Pi_{(n:N)} C(n)$$

which follows the computation rules,

$$ind_N(C, c_0, c_s, 0) :\equiv c_0$$

$$ind_N(C, c_0, c_s, succ(n)) :\equiv c_s(n, ind_N(C, c_0, c_s, n))$$

In case of natural numbers also, taking a constant C gives recursion principle and the function is said to be defined by primitive recursion.

1.6.3 Identity Type

Identity or equality types correspond to the type of equality of two elements of the same type, $a, b : A$ from the propositions-as-types correspondence. This type, written as $a =_A b$, is different from the judgement of definitional equality.

Identity types are an example of generalization of inductive types, called inductive type family. They are indexed families which, instead of constructing an element of Type, construct a function ... $\to$ Type, where ... are indices. They are represented in Lean as,

```
inductive foo : ... → Type u
|constructor₁ : ... → foo ...
|constructor₂ : ... → foo ...
...
|constructor_n : ... → foo ...
```

The definition of equality type in Lean is

```
inductive eq {α : Sort u} (a : α) : α → Prop
| refl : eq a
```

This definition constructs a type family eq a x which is indexed by x.

The formal type former rules are given as:

- Formation rule

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

  We can form the equality type in the same universe as A.

- Introduction rule

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash refl_a : a =_A a} = \quad -\text{INTRO}$$

It states that $refl_a$ is a constructor for the type $a =_A a$. If a and b are judgementally equal, then $refl_a$ is an element of $a =_A b$ as well. It is important to note that even though $refl$ is the only introduction rule, it is not the only possible element of equality type.

- Elimination rule

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

- Computation rule

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

The last two rules (also called as path induction) state that, to define a function on a type family $C : \Pi_{(x,y:A)}(x =_A y) \to U$, we need a function c : $\Pi_{(x:A)}C(x, x, refl_x)$. We can then define a function,

$$ind_{=_A} : \prod_{(C:\Pi_{(x,y:A)}(x=_A y)\to U)} (\Pi_{(x:A)}C(x, x, refl_x)) \to \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

with the computation rule,

$$ind_{=_A}(C, c, x, x, refl_x)) :\equiv c(x)$$

We consider a special case to understand this interpretation. If C doesn't depend on p, then C associates propositionally equal x, y to $C(x, y)$ and c is a proof of $C(x, x)$. We conclude

that $C(x, y)$ is true. Hence to prove something about propositionally equal x, y, we need to consider the case where $x \equiv y$. In general, if C depends on p, we need to consider the case where x and y are judgementally equal and the proof is $refl$.

An important consequence of the induction principle is the indiscernability of identicals, which is the recursion principle for equality type. It states that type families respect equality.

It is important to note that the induction principle cannot be used on a family $C(p)$ where x and y are fixed. The rule only says that when x and y vary over elements of A, then the triple (x, y, p) can be generated using $(x, x, refl_x)$. Hence we cannot prove that all proofs of equality $(x = x)$ are reflexivity. This non trivial structure of equality is important in homotopy interpretation of type theory.

There is a second way to represent induction principle, known as based path induction, where we don't need to vary both x and y over A and instead can consider the type family $a =_A x$ where a is fixed. We can prove that this is equivalent to path induction.

Chapter 2

Logic and Proof

2.1 Propositions as types

Until now, we have seen how to construct general types and represent mathematical objects using them (for instance natural numbers). Now we will look at the notion of logic and proof in this theory. There exists an external notion of proof in the metatheory where we derive judgements on the basis of some rules. Using the internal notion of proof, we can incorporate logic into the type theory itself, unlike set theory. This comes from the Curry-Howard isomorphism, which gives a correspondence between proofs and programs. It means that if we interpret propositions as types, there is a natural correspondence between logical operations and various type formers, and also between natural deduction rules and type system for λ-calculus.

Hence, propositions and proofs are placed on the same level as types and their elements, and the typing rules can be applied on them. To prove a proposition, we have to construct an element of that type using introduction or elimination rules. To understand this better, we will look at how various logical operations and rules of natural deduction system correspond to specific type theoretic operations and how there is a natural correspondence between logic and programming. Types can also be interpreted as propositions that they are non empty. In that case, the proof is precisely an element of the type.

The following table gives different points of view for type theoretic operations.

Types	Logic
A	proposition
a:A	proof
B(x)	predicate
b(x):B(x)	conditional proof
0,1	$\bot, \top$
A+B	A$\vee$ B
A$\times$ B	A$\wedge$ B
A$\rightarrow$B	A$\Rightarrow$B
$\Sigma_{(x:A)}$B(x)	$\exists_{x:A}$B(x)
$\Pi_{(x:A)}$B(x)	$\forall_{x:A}$B(x)
Id_A	equality=

Lean has a specific type for propositions, Prop. This means that all types belonging to Prop are viewed as propositions rather than just data. This notion is evident from the following example. Given types A and B, if B : Prop, then Π x : A, B x also belongs to Prop because it is also a proposition depending on A. Furthermore, if p : Prop and t_1, t_2 are proofs of p, then they are treated to be definitionally equal.

2.2 Propositional logic

Natural deduction gives rules for proving propositional formulae which are built using propositions and logical connectives. Here, we will give the rules for various connectives and show the correspondence between rules of introduction and elimination of types, and rules of reasoning about propositions.

We represent the statement that C follows logically from hypotheses A and B as

$$\frac{A \quad B}{C}$$

The introduction and elimination rules characterize any logical connective,i.e., rule to prove

a claim which involves the connective and rule to use a statement containing the connective to derive others.

- **Implication**

  Implication represents if...then statements,i.e., A $\implies$ B means that truth of B can be inferred if the truth of A is known.

  The introduction and elimination rules are:

$$\frac{\begin{array}{c}\overline{A}\\ \vdots\\ B\end{array}}{A \to B} \to I$$

$$\frac{A \to B \qquad A}{B} \to E$$

  The introduction rule states that, if on assuming A, we are able to show B, then we can show A $\to$ B.

  Elimination rule (also known as modus ponens) states that if we have a proof of A $\implies$ B and A, then we get a proof of B.

  These rules correspond to the introduction and elimination rules of functions because getting a proof of B on assuming proof of A is equivalent to having a function A $\to$B and deriving a proof of B from A $\implies$ B and A is the same as function application.

  In Lean, if we have a function type A $\to$B where B : Prop, then A $\to$B : Prop.

- **Conjunction**

This represents the propositions of the form A and B, hence combining two assertions into one. The introduction and elimination rules are:

$$\frac{A \qquad B}{A \wedge B} \wedge I$$

$$\frac{A \wedge B}{A} \wedge E_l$$

$$\frac{A \wedge B}{B} \wedge E_r$$

It states that given proof of A, B, we get proof of A and B. This is the same as introduction rule for product type.

Elimination rule states that given proof of (A and B), we can extract proofs of A, B separately. For product type A $\times$ B, we can construct projection functions and get elements of A and B by elimination rule. Hence conjunction corresponds to product type.

In Lean, 'and' type is defined as

```
inductive and (a b : Prop) : Prop
intro : a → b → and a b
```

This definition is similar to 'prod', except the arguments and the type belong to Prop.

- **Disjunction**

Disjunction represents propositions of the form A or B. The introduction and elimination rules are:

$$\frac{A}{A \vee B} \vee I_l$$

$$\frac{B}{A \vee B} \vee I_r$$

$$\frac{A \vee B \qquad \begin{array}{c}\overline{A}\\ \vdots \\ C\end{array} \qquad \begin{array}{c}\overline{B}\\ \vdots \\ C\end{array}}{C} \vee E$$

This states that A $\vee$ B can be derived separately by both A and B. This is similar to the introduction rule of coproduct type A+B, where elements are constructed by elements of A or B by left injection or right injection.

Elimination rule states that given a proof of A $\vee$ B, and case wise proofs of A $\implies$ C and B $\implies$ C for some C, then we get a proof of C. This is exactly the case analysis elimination rule for coproduct type.

In Lean, we have the type 'or' defined as

```
inductive or (a b : Prop) : Prop
| intro_left : a → or a b
| intro_right : b → or a b
```

It is similar to A+B except 'or' type belongs to Prop.

- **Negation**

  This represents the proposition 'not A' and logically means showing that A leads to contradiction. We will introduce logical terms true ($\top$) and false ($\bot$) here.

  False refers to contradiction or impossibility in logical language. It has no introduction rule and one elimination rule that you can derive anything from contradiction. It corresponds to the empty type.

  True has one introduction rule, that it is true in all cases, and no elimination rule. It corresponds to the unit type.

Now, the introduction and elimination rules for negation, true and false are:

$$\frac{\begin{array}{c}\overline{A}\\ \vdots\\ \bot\end{array}}{\neg A}\neg I$$

$$\frac{\neg A \qquad A}{\bot}\neg E$$

$$\frac{\bot}{A}\bot E$$

$$\frac{}{\top}\top I$$

The introduction rule for negation states that if on assuming A, we get a contradiction, then we get a proof of ¬A. Elimination rule states that if we have A and ¬A, then we have a contradiction. If we think of negation ¬A as A $\to$ **0** (where **0** is the empty type) , then the rules correspond to implication rules. Hence the corresponding type is function type.

2.3 Predicate Logic

We have seen the types which represent propositional logic. To work with a family of propositions or propositions formed by quantification which make a statement about a family of propositions, we need to look at predicate logic. Predicates which take an element of A as an input and give a proposition, correspond to the type family B : A $\to$ U. We will look at quantifiers and equality in this section.

- **Universal quantifier**

The universal quantifier '∀' reads for all. ∀ x, P(x) states that every value of x follows the property P. It has the following introduction and elimination rules.

$$\frac{A(x)}{\forall y A(y)}\ \forall \text{ I}$$

$$\frac{\forall x A(x)}{A(t)}\ \forall \text{ E}$$

It states that if A is true for an arbitrary x, then it is true for all x. Elimination rule states that if A is true for all x, then it is true for some specific t. These rules are the same as function abstraction and application. However, since the proposition is dependent on x, universal quantifier corresponds to dependent function type.

Lean has the dependent type Π x : A, B x. If B(x) has the type Prop, then we use the universal quantifier type ∀ x : A, B x, which also belongs to Prop.

- **Existential quantifier**

The quantifier '∃' reads there exists. It expresses assertions of the form ∃ x, A(x) which means that there exists some x such that A(x) is true. The introduction and elimination rules are:

$$\frac{A(t)}{\exists x A(x)}\ \exists \text{ I}$$

$$\frac{\exists x A(x) \qquad \begin{array}{c}\overline{A(y)}\\ \vdots \\ B\end{array}}{B}\ \exists \text{ E}$$

Introduction rule states that if t is any term for which A(t) is true, then ∃ x, A(x) is true. Hence we need to provide a witness t. Elimination rule states that if we have a proof of ∃ x, A(x) and A(y) → B for an arbitrary y, then we have a proof of B.

We note that in introduction rule for dependent pair type, we need to provide an element of A and $B(a)$, given $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, which is similar to that of $\exists$. In case of elimination rule too, to construct an element of C, we need an element of $\exists x : A, B(x)$ and a function $\Pi x : A, B(x) \to C$. Hence existential quantifier corresponds to dependent pair type.

Similar to other propositional types, Lean has a type Exists (or $\exists$) which belongs to Prop and is defined as:

```
inductive Exists {A : Type}(P : A → Prop) : Prop
intro : ∀ (a : A), P a → Exists P
```

- **Equality**

  In logic, the expression s = t states that "s and t are 'equal' or identical"". This could mean that they are the same object or different descriptions of same object.

  We have the following rules for equality:

$$\frac{}{t = t} \quad \text{refl}$$

$$\frac{s = t}{t = s} \quad \text{symm}$$

$$\frac{r = s \qquad s = t}{r = t} \quad \text{trans}$$

$$\frac{s = t}{r(s) = r(t)} \quad \text{subst}$$

$$\frac{s = t \qquad P(s)}{P(t)} \quad \text{subst}$$

The introduction rule is reflexivity, which states that an object is equal to itself. The elimination rule is substitution; we can substitute equal elements. The other rules can be derived from these two. They show that equality is am equivalence relation.

In type theory, we have the identity type to represent equality. It has similar introduction and elimination rules, except elimination rule is more general and quantifies over different kinds of proofs as well. Its consequence is the substitution rule. Another difference is that type theory differentiates between definitional and propositional equality.

In the standard kernel of Lean, the equality type belongs to Prop. Here, any two proofs of the type are considered judgementally same. However, the Prop is absent in homotopy type theory kernel.

2.4 Classical vs Constructive logic

The logic represented by proposition as types interpretation is different from the usual classical logic of mathematics; it is constructive in nature. Propositions are not just true or false, they are viewed as a collection of their proofs. Hence a valid proof of a proposition is one which has a valid type and can be constructed in a step by step manner, following the computational rules of types, as proofs exist at the same level as elements of types.

For instance, to prove A, it is not possible to assume $\neg A$ and get a contradiction, as no obvious function exists from $\neg(\neg)A$ to A. This is an example of proof by contradiction used in classical logic. Another principle used in classical logic is law of excluded middle. It implies in type theoretic context that for every type A, we have a term of the type $A + (A \to 0)$. Hence it gives a method to prove or disprove any proposition. However, the logic of type theory doesn't computationally derive that every proposition is either true or false. Hence, we can't prove LEM using logic, but adding it as an axiom is consistent with type theory. Yet another difference between classical and type theory logic is the axiom of choice, which is a provable theorem in type theory.

The core parts of standard library of Lean are computationally pure and don't have the type Prop. However, Lean also gives the option to use classical choice axiom (Hilbert operator) to support classical reasoning, whose consequence is LEM. It also provides the axioms propositional and function extensionality.

We have seen the types and rules corresponding to logical operations. Now, we will see how to give complete proofs using them. We give a simple example each from propositional logic and predicate logic.

Consider a proposition (A and B) $\implies$ (B and A). In natural deduction style, it can be proved as follows.

$$\dfrac{\dfrac{\dfrac{}{A \wedge B}}{B} \qquad \dfrac{\dfrac{}{A \wedge B}}{A}}{B \wedge A}$$

The Lean version is:

```
variables A B : Prop
theorem commutative : (A ∧ B) → (B ∧ A) :=
λ h : (A ∧ B), and.intro (and.right h) (and.left h)
```

To formalize the proof in type theory, we first need to find the valid type of the proposition. In this case, the proposition is (A $\wedge$ B) $\rightarrow$ (B $\wedge$ A). 'commutative' is the name given to the proof and the command theorem is used to give a definition of 'commutative' as the relevant proof object.

Now we construct the object as follows. Since it is a function type between two propositions, we assume an object of the domain and specify a valid object of codomain to which it is mapped. The object of (B $\wedge$ A) is obtained by applying introduction and elimination rules of and type.

Now consider a proposition involving universal quantifier, $\forall$x, A(x) and B(x) $\implies$ $\forall$x, A(x). We can prove it in natural deduction style as,

$$\dfrac{\dfrac{\dfrac{\forall x, A(x) \wedge B(x)}{A(y) \wedge B(y)}}{A(y)}}{\forall y, A(y)}$$

The Lean version is

```
variable U : Type
variables A B : U → Prop
example : ∀x, A x ∧ B x → ∀x, A x :=
assume h : ∀x, A x ∧ B x,
assume y,
show A y, from and.left (h y)
```

'example' command is used to define a proof object without giving name to it. Here, we use the 'assume' and 'show' commands which are an alternate syntax for λ-abstraction. 'show' command is used to check that the object defined in from has the correct type.

Now, to define an element of the proposition, we use λ-abstraction twice.We assume a proof of (∀x, A x ∧ B x), take an arbitrary y, and introduce an object of (∀x, A x) using function application and '∧' elimination rules.

Lean provides alternative syntax for many commands which are proof friendly and look informal. Lean also allows us to introduce goals in long proofs. It also provides features like tactics, which are instructions that describe how to build a proof. They make proofs shorter and easier to write.

Chapter 3

Example - Formalization of graphs

As a part of getting familiarized with type theory, my goal in this project was to work on a concrete problem of formalizing graphs. In particular, the aim was to 1.) write an algorithm to check if a finite graph is connected and prove the correctness of the algorithm 2.) show that such an algorithm is not possible in a general case.

Graphs as mathematical objects, are a collection of vertices with a relation on them which gives edges. We can form different types of graphs based on whether the edges are directed, whether there are multiple edges between two vertices and whether there are loops on a single vertex. In a general case, we can define a graph to be a triplet (V,E,ϕ) where V and E are sets of vertices and edges respectively, and ϕ is a function mapping each edge to a pair of vertices. Path is an ordered collection of edges, where the end point of an edge is the starting point of next edge.

Here, we define an undirected graph, allowing multiple edges and loops. I have used a syntactic variation of inductive type, called structure to represent a graph. To define path, I have used inductive family, which given a graph and a starting point, takes a vertex as input and gives a path ending with the vertex. This is followed by some simple objects of these types.

```
universes u v
```

```
structure graph :=
(vertex : Type u)
(edge : Type v)
(ϕ1 : (edge → vertex))
(ϕ2 : (edge → vertex))

#check graph
#print graph

/-graph with single point and loop-/
inductive One:Type
|one:One
def a (x:One) : One := One.one
def graph0 : graph :=
{vertex := One, edge := One,ϕ1 := a, ϕ2 := a}
#print graph0

/-graph with 2 points and an edge-/
inductive Two:Type
|one:Two
|two:Two
def b1 (x:One) : Two := Two.one
def b2 (x:One) : Two := Two.two
def graph1 : graph :=
{vertex := Two, edge := One, ϕ1 := b1, ϕ2 := b2}
#print graph1

inductive path (g:graph.{u v})(start:g.vertex):
(g.vertex) → Type (max u v)
|fix{} : path start
|addedge (add:g.edge)(last:g.vertex)(p:path last)(pr:last = g.ϕ1 add):
path (g.ϕ2 add)

#check path
```

```
#check path.addedge

/-path with single point-/
def path0 : path graph0 One.one One.one:= path.fix
/-path with one edge-/
def path1a : path graph1 Two.one Two.one:= path.fix
def path1b : path graph1 Two.one Two.two :=
path.addedge One.one Two.one path1a rfl
#check path1b
#print path1b
```

Now, to give an algorithm to check connectedness, we need an extra condition for the graph to be finite. In general case, such as for infinite graphs, such an algorithm is not possible. We can only have a half algorithm, which either gives one of yes/no output or keeps going on forever. However in the finite case, the program halts after finite number of steps, so it is possible to have a concrete yes/no for each input. We call such problems undecidable.

To define a finite graph, I have used finite sets. I have defined vertices and edges to be separate sets and added a proposition that edge is subset of product of vertex sets. I have defined a neighbor_of_set function, which takes a set of vertices and gives a set containing all their immediate neighbors, including the input set as well. Then, we can introduce a function which takes a set of vertices and gives its connected component by recursively using neighbor_of_set function, and based on whether the connected component is equal to the vertex set, it decides the connectedness of graph.

```
import data.finset

variable β : Type

structure finitegraph (β : Type):=
(fvertex: finset β)
(fedge : finset (β × β))
(is_sub: fedge ⊆ (finset.product fvertex fvertex))
```

```
def neighbor_of_set (g:finitegraph nat) (s:finset nat) (p:s ⊆ g.fvertex)
: finset nat :=
finset.filter (λ v, (∃ (w : nat) (h : w ∈ s),
(v,w) ∈ g.fedge ∨ (w,v) ∈ g.fedge))
g.fvertex ∪ s

#check neighbor_of_set
#print neighbor_of_set
```

We can further define other operations on graphs and develop the theory. We can also prove that the connectivity problem is non-computable by comparing it to a previously known and proved undecidable problem. This code and its updates can be viewed on the github repository **https://github.com/enharsha/Graphs-in-Lean**

Chapter 4

Homotopy Type Theory

Homotopy type theory, as an extension of type theory, introduces an idea of regarding types as spaces. It explores the deep connections between homotopy theory and type theory.

In classical homotopy theory, we associate to each $x_0 \in$ X a fundamental group $\Pi_1(X, x_0)$. This can be generalized to fundamental groupoid $\Pi_1(X)$ of the space X. A groupoid is a category where all morphisms are isomorphisms. A category has objects, morphisms between them, and laws of composition, associativity, and identity. The points of space are considered as objects and homotopy classes of paths between them as morphisms. On generalizing $\Pi_1(X)$ vertically, we get $\Pi_\infty(X)$ or weak ∞-groupoid where we have morphisms between morphisms (2-morphisms, 3-morphisms and so on). We take points as objects and actual paths as morphisms and groupoid laws on paths of nth level are satisfied upto (n+1)th level.

Now, type theory is related to homotopy theory in the sense that the induction rule for identity types gives an ∞-groupoid structure to every type. The elements $a, b : A$ are points of space and $p : a =_A b$ is a path between them. The higher structure comes from the fact that since $a =_A b$ is a type, we have equality between elements of them; $p =_{x=_A y} q$, $r =_{p=_{x=_A y} q} s$ and so on. We will see in detail that they follow the laws of groupoid structure. In the upcoming sections, we will give a very brief overview of homotopy type theory as an extension to type theory,i.e., addition of univalence axiom, and the concepts leading upto it, without going into details of many proofs. We will also see how we can do classical logic in homotopy type theory.

In homotopy type theory, notion of spaces is synthetic,i.e., points and paths are basic notions and logical deductions are made on them. Classical homotopy theory is based on analytic geometry where cartesian coordinates are used to represent points and lines.

The following table compares the connections between types, spaces and groupoids which we will see in detail.

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

4.1 Types as higher groupoids

We will prove symmetry and transitivity of paths and coherence laws on operations on higher dimensions as well.

Lemma 4.1.1. *For every type A and every $x, y : A$ there is a function*

$$(x = y) \to (y = x)$$

denoted $p \mapsto p^{-1}$ such that $refl_x^{-1} \equiv refl_x$ for each $x : A$. We call p^{-1} the inverse of p.

Proof. Formally, we need to construct an element of $(y = x)$. To use the induction principle, we define ,

$$D : \Pi_{(x,y:A)}(x = y) \to \mathcal{U}, D(x, y, p) \equiv (y = x)$$

$$d \equiv \lambda x.refl_x : \Pi_{(x:A)} D(x, x, refl_x)$$

Hence we obtain $ind_A(D, d, x, y, p) : (y = x)$ and we get $refl_x^{-1} \equiv refl_x$ from the computation rule. □

We can also prove this informally using the elimination rule that it is sufficient to prove the principle when $y \equiv x$ and p is $refl_x$. This is an informal way of implicitly assuming the type family and producing an object of $D(x, x, refl_x)$. We will use this way of proving henceforth. Now, when $y \equiv x, (x = y)$ and $(y = x)$ become $(x = x)$. Hence we define $refl_x^{-1} \equiv refl_x$. Rest follows from computation rules.

Lemma 4.1.2. *For every type A and every x, y, $z : A$ there is a function*

$$(x = y) \to (y = z) \to (x = z)$$

written $p \mapsto q \mapsto p \centerdot q$ such that $refl_x \centerdot refl_x \equiv refl_x$ for any $x : A$. We call $p \centerdot q$ the concatenation or composite of p and q.

Proof. We need to construct an element of $(x = z)$ given $x, y, z : A, p : (x = y)$ and $q : (y = z)$. When $y \equiv x$ and p is $refl_x$, then $q : (x = z)$. By induction on q, assume that $z \equiv x$ and q is $refl_x$. Then $(x = z)$ becomes $(x = x)$ and we get $refl_x : (x = x)$ as an element of $(x = z)$. Hence proved. □

We now prove coherence laws on operations at higher dimensions.

Lemma 4.1.3. *Suppose $A : U$, that x, y, z,$w : A$ and that $p : x = y$ and $q : y = z$ and $r : z = w$. We have the following:*

1. $p = p \centerdot refl_y$ *and* $p = refl_x \centerdot p$
2. $p^{-1} \centerdot p = refl_y$ *and* $p \centerdot p^{-1} = refl_x$.
3. $p^{(-1)^{(-1)}} = p$
4. $p \centerdot (q \centerdot r) = (p \centerdot q) \centerdot r$

Proof. 1. Assume y is x and p is $refl_x$. Then we have $refl_x \centerdot refl_x \equiv refl_x$ from transitivity law. Hence we get a proof in the case of reflexive paths.

2. Assume p is $refl_x$. From symmetry and transitivity, we have $p^{-1} \centerdot p \equiv refl_x^{-1} \centerdot refl_x \equiv refl_x$. Hence proved.

3. Assume p is $refl_x$. Then $p^{-1^{-1}} \equiv refl_x^{-1^{-1}} \equiv refl_x$. Hence proved.

4. Assume p, q, r are $refl_x$. Then,

$$\begin{aligned} p \centerdot (q \centerdot r) &\equiv refl_x \centerdot (refl_x \centerdot refl_x) \\ &\equiv refl_x \\ &\equiv (refl_x \centerdot refl_x) \centerdot refl_x \\ &\equiv (p \centerdot q) \centerdot r \end{aligned}$$

□

4.2 Function extensionality and Univalence axiom

We have seen that types behave like spaces or groupoids. The functions between spaces also behave functorially,ie, they respect equality (or paths).

Lemma 4.2.1. *Suppose that $f : A \to B$ is a function. Then for any $x, y : A$ there is an operation*

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

Moreover for each $x : A$, we have $ap_f\ (refl_x) \equiv refl_{f(x)}$.

Proof. Assume p is $refl_x$. Then we can define $ap_f(p)$ to be $refl_{f(x)} : f(x) \to f(x)$. By induction principle, we can generalize it to when x and y are not definitionally equal. □

The function *ap* is consistent with the functor rules.

As functions behave like functors, dependent types can be interpreted with the help of fibrations. Given a type family $P : A \to \mathcal{U}$, it can be thought of as a fibration with A as base space, $P(x)$ as fiber over x and $\Sigma_{(x:A)}P(x)$ as total space. Given $p : (x = y)$ in A and $u : P(x)$, p can be lifted to a path in the total space starting at u. Using this, we can give a corresponding formulation of **Lemma 3.2.1** for dependent functions. We prove these properties below.

Lemma 4.2.2 (Transport)**.** *Suppose $P : A \to \mathcal{U}$ and $p : (x = y)$. Then there is a function $p_* : P(x) \to P(y)$.*

Proof. Assume p is $refl_x$. We can define p_* or $refl_*$ as the identity function from $P(x)$ to $P(x)$. Hence proved. □

Lemma 4.2.3 (Path lifting property)**.** *Let P be a type family over A and assume we have $u : P(x)$ for some $x : A$. Then for any $p : (x = y)$,*

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

in $\Sigma_{x:A}\ P(x)$.

Proof. In the case $y \equiv x$ and p is $refl_x$, we get $(y, refl_*(u))$ which is definitionally equal to (x, u) as $refl_*$ is the identity function. Hence we get reflexivity as a proof of $(x, u) = (x, u)$. □

Lemma 4.2.4 (Dependent map)**.** *Suppose $f : \Pi_{(x:A)}\ P(x)$, then we have a map,*

$$apd_f : \Pi_{(p:x=y)}\ (p_*(f(x)) =_{P(y)} f(y)).$$

Proof. Assume p is $refl_x$. Substituting y with x and p with $refl_x$, we get the the type $(\ refl_*(f(x)) = f(x))$. Since $refl_*$ is the identity function, $refl_{f(x)}$ is the proof of this type. □

We have seen the identifications between elements of a given type. To identify functions, we need the condition that their values on equal inputs are equal.

Definition 4.2.1. *Given f, g : $\Pi_{(x:A)}$ P(x), we define a homotopy from f to g as*

$$(f \sim g) :\equiv \Pi_{x:A}\ (f(x) = g(x))$$

This type can be viewed as the type of homotopies (continuous paths) or natural isomorphisms.Homotopy is an equivalence relation. However, it is not the same as $(f = g)$.

Types can be identified by the idea of equivalence,i.e., two types are equivalent if there are functions between them whose composites are homotopic to identity function. This corresponds to homotopy equivalence in topology and equivalence of groupoids in category theory.

We define a function isequiv(f).

$$\text{isequiv(f)} :\equiv (\Sigma_{g:B\to A}(f \circ g \sim id_B)) \times (\Sigma_{h:B\to A}(h \circ f \sim id_A))$$

Definition 4.2.2. *An equivalence between types A and B ($A \simeq B$) is defined as a function $f : A \to B$ along with a proof of isequiv(f).*

$$A \simeq b :\equiv \Sigma_{f:A\to B}\ isequiv(f)$$

Type equivalence is an equivalence relation on $\mathcal{U}$. We can also prove that if $e_1, e_2 :$ isequiv(f), then $e_1 = e_2$. With the help of equivalence, we can characterize the equality types of many types, based on the data used to construct them. For instance, we can show for cartesian product that there is an equivalence between equality of ordered pairs and equality of product of their components.

In the case of dependent function types, given a type family $B : A \to \mathcal{U}$ and $f, g : \Pi_{x:A}, B(x)$, we can expect that equality type $(f = g)$ is equivalent to homotopy type $(f \sim g)$. By induction principle, we can construct a function

$$\text{happly} : (f = g) \to \Pi_{x:A}(f(x) =_{B(x)} g(x))$$

The type theory we have seen till now is insufficient to prove the equivalence. Hence we introduce it as an axiom.

Axiom 4.2.1 (Function extensionality)**.** *For type A, type family $B : A \to \mathcal{U}$ and f, g : $\Pi_{x:A}$, B(x), the function happly is an equivalence.*

Similarly, for any two types A and B, we expect $A =_{\mathcal{U}} B$ to be equivalent to $A \simeq B$. By path induction we can construct,

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

Since the type theory is not enough to guarantee equivalence of idtoeqv, we introduce this as an axiom as well.

Axiom 4.2.2 (Univalence)**.** *For any $A, B : \mathcal{U}$, idtoeqv is an equivalence.*

This axiom is an important addition to type theory by Voevodsky which is motivated by the homotopy interpretation of types. Logically, it means that isomorphic things can be identified. Homotopy type theory, with the addition of univalence, formalizes this idea, which mathematicians have been using informally. Homotopy theory also motivates new ways of representing spaces which are inductively defined, such as CW complexes, using higher inductive types. These ideas provide strong tools for constructive mathematics.

4.3 Sets and Logic

The logic of homotopy type theory is constructive,i.e., we construct an element of the relevant type to prove a proposition. This is not the case with classical logic where we have law of double negation and law of excluded middle, which we assume as axioms in type theory. However, with the addition of univalence axiom, we can prove that these laws do not necessarily hold true for all types. To incorporate classical logic in homotopy type theory, we use a modified idea of propositions as types.

We give the following definitions:

Definition 4.3.1. *A type is a set if for all $x, y : A$ and all $p, q : (x = y)$, we have $p = q$. The proposition isSet(A) can be defined as*

$$isSet(A) :\equiv \Pi_{x,y:A}\ \Pi_{p,q:x=y}\ (p = q)$$

Examples of types which are sets are unit type, empty type and type of natural numbers.

Definition 4.3.2. *A type A is a 1-type if for all x, y : A and p, q : x = y and r, s : p = q, we have r = s.*

Definition 4.3.3. *A type P is a mere proposition if for all x, y : P, we have x = y. It can be represented as*

$$isProp(P) :\equiv \Pi_{x,y:P}\ (x = y)$$

This definition of mere proposition equips the idea of propositions in conventional logic that they don't contain any more information than being true or false. They are true if inhabited and false if $(A \to 0)$ is inhabited. The definition of sets encaptures the idea that its elements are abstract points and contain no higher homotopical information.

Sets can be thought of as 0-types and propositions as (-1)-types, by extrapolating the idea of 1-type. We can prove that every mere proposition is a set and further that isProp(A) and isSet(A) are mere propositions.

We can now state law of excluded middle and law of double negation as:

$$\text{law of excluded middle, LEM} :\equiv \Pi_{A:\mathcal{U}}(\text{isProp}(A) \to (A + \neg\ A))$$

$$\text{law of double negation- } \Pi_{A:\mathcal{U}}\ (\text{isProp}(A) \to (\neg\neg A \to A))$$

These laws are consistent because they are applicable on mere propositions and can be added as axioms. Any type for which LEM holds true is called decidable.

We can also add impredicativity to homotopy type theory by adding the propositional resizing axiom. We have natural maps $\text{Prop}_{\mathcal{U}_i} \to \text{Prop}_{\mathcal{U}_{i+1}}$ because of cumulative property

of type universes. We can make this into an equivalence by adding an axiom. We can then define a type $\Omega :\equiv \mathrm{Prop}_{\mathcal{U}0}$ containing all mere propositions, where $\mathcal{U}_0$ is the smallest universe. This is possible because any mere proposition in $\mathcal{U}_{i+1}$ can be resized to U_i as a consequence of LEM.

We have seen that logical operations correspond to type formers. Hence we need to ensure that they preserve mere propositions. It is true in general for all connectives and quantifiers, except coproduct (or) and dependent pair types (Σ). Given mere propositions A and B, we cannot guarantee that $A+B$ is a mere proposition (for instance, $\mathbf{2}$, which is equal to $\mathbf{1}+\mathbf{1}$ is not a mere proposition). Same follows for Σ-type. We then define a type former propositional truncation, which takes any type and gives a mere proposition by forgetting all the information about its inhabitance apart from its existence. We can then redefine the logical notations, given mere propositions P and Q, to represent traditional logic.

$$\begin{aligned}
\top &:\equiv 1 \\
\bot &:\equiv 0 \\
P \wedge Q &:\equiv P \times Q \\
P \Rightarrow Q &:\equiv P \to Q \\
P \Leftrightarrow Q &:\equiv P = Q \\
\neg P &:\equiv P \to 0 \\
P \vee Q &:\equiv \| \ P + Q \ \| \\
\forall (x : A).P(x) &:\equiv \prod_{x:A} P(x) \\
\exists (x : A).P(x) &:\equiv \| \sum_{x:A} P(x) \ \|
\end{aligned}$$

In Lean, we have seen that there exists a pure homotopy type theory kernel which doesn't contain Prop or classical axioms or mere propositions. However, there is a kernel based on impredicative CoC, which captures these notions.

Bibliography

[1] The Univalent Foundations Program, Homotopy Type Theory : Univalent Foundations of Mathematics, Institute for Advanced Study(Princeton), 2013. Available online- http://homotopytypetheory.org/book/.

[2] Jeremy Avigad, Leonardo de Moura and Soonho Kong, Theorem Proving in Lean, version d0dd6d0, 2017. Available online- https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf

[3] Jeremy Avigad, Robert Y. Lewis and Floris van Doorn, Logic and Proof. Available online- https://leanprover.github.io/logic_and_proof/.

[4] Lawrence Dunn III, An overview of Homotopy Type Theory and the Univalent Foundations of Mathematics, Honors thesis, Florida State University, 2014

[5] Nino Guallart, An overview of type theories, arXiv : 1411.1029v2[math.LO], 2014