National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

Homotopy Type Theory and the Fundamental Group of a Bouquet of Circles

DIPLOMA THESIS

GEORGE N. MAGAFOSSIS

Supervisor : Nikolaos S. Papaspyrou
NTUA Professor

Athens, December 2021

National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

Homotopy Type Theory and the Fundamental Group of a Bouquet of Circles

DIPLOMA THESIS

ΓΙΏΡΓΟΣ Ν. ΜΑΓΚΑΦΏΣΗΣ

Supervisor : Nikolaos S. Papaspyrou
NTUA Professor

Approved by the examining committee on the December 30, 2021.

....	....	....
Nikolaos S. Papaspyrou NTUA Professor	Aris T. Pagourtzis NTUA Professor	Dimitris Fotakis NTUA Associate Professor

Athens, December 2021

....

George N. Magafossis
Electrical and Computer Engineer

Abstract

The purpose of this diploma dissertation is on one hand a brief presentation of the main features of Homotopy Type Theory and on the other hand to prove a theorem of classical Homotopy Theory inside HoTT.

Key words

Homotopy Type Theory, Intentional Type Theory, Functional Programming, Group Theory, Fundamental Group, Free Group.

Acknowledgements

I would like to thank the supervisor professor of this thesis, Nikolaos Papaspyrou, for his inspiring introduction to Type Theory and Programming Languages as well as for his advice to work on Homotopy Type Theory, which has surprised me with its special connection to pure mathematics. I would also like to thank my sister Xantippe Magafossi for correcting the text and for her precious advice and help. Finally, I would like to thank Eleftheria Makrinikola and Dimitra Kastora for their important contribution to the English translation of this thesis.

George N. Magafossis,

Athens, December 30, 2021

This thesis is also available as Technical Report CSD-SW-TR-6-21, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, December 2021.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

Chapter 1

Introduction

1.1 Type Theory

Type Theory is fundamental to both Mathematical Logic as well as to Computer Science. Its creator was Bertrand Russell whose intention was to build a system for the Foundation of Mathematics freed from the inconsistencies of naive Set Theory. In the following decades, it developed significantly, mainly by its correlation with λ-calculus by Alonzo Church. Type Theory found a great field of applications in Programming Languages with the progress of computers. One of the most important milestones was the modification of Church's Type Theory to a more *categorical* theory by Per Martin-Löf, the Dependent Type Theory or Intentional Type Theory.

In recent years, Type Theory has flourished with the discovery of Homotopy Type Theory. The idea of Homotopy Type Theory arose around 2006 with the independent work of Awodey-Warren and Voevodsky, who were inspired by the groupoid interpretation of Type Theory by Hofmann and Streicher, which had preceded. Higher Category Theory, which is closely related to Homotopy Theory is now intensely studied by mathematicians of both fields. The semantic models of Awodey-Warren and Voevodsky use well-known concepts and techniques from Homotopy Theory. In particular, Voevodsky constructed an interpretation of Type Theory which satisfies an extra important property which he called univalence. This had not been proposed before in Type Theory and its addition as an axiom has profound consequences for the theory.

1.2 Purpose - Structure

The purpose of this thesis is double. One the one hand, to describe briefly Homotopy Type Theory, to the extent that this is possible, and on the other hand to prove a well-known result from Algebraic Topology inside HoTT. So, this dissertation has the following structure.

Chapter 2 The basis on which Homotopy Type Theory is built on is Intentional Type Theory. In this chapter we firstly give the basic notions of Type Theory such as types, judgments, contexts etc. Also, we give a long and detailed description of the constructions that are possible inside this theory. We emphasize on Identity Types, for which we provide the most important results and tools.

Chapter 3 The central subject of the first part of this thesis is the content of this chapter: a description of Homotopy Type Theory. We begin with an explanation of the Homotopy Interpretation of the types of ITT and we focus on the basic algebraic characteristics that make this interpretation possible. Then, follows the analysis of the two important new features that Homotopy Type Theory brings: the Univalence Axiom and Higher Inductive Types.

Chapter 4 The version of Homotopy Type Theory on which the proof of the theorem was based is Cubical Type Theory. Cubical Type Theory was developed in order to establish the computational character of HoTT and mainly to give the appropriate conditions in which it was possible to prove

Univalence Axiom as a theorem. We do a brief introduction of the main points of the theory and we present the basic results.

Chapter 5 This chapter covers the second part of this thesis, which is to prove a theorem using Cubical Type Theory. The theorem that we chose gives an identification of types. It states that the fundamental group of a wedge of circles, also known as bouquet, indexed by a type A is identified with the free group generated by the elements of A. The library that was developed using the Cubical Agda proof assistant is freely accessible in the following website: https://github.com/gmagaf/Bouquet-HoTT-Cubical-Agda.

Chapter 6 The last chapter is dedicated to possible extensions of the theorem, fields of research in HoTT and open problems.

Chapter 2

ITT - Intentional Type Theory

Homotopy Type Theory is based on Intentional Type Theory, which was first presented in 1972 by the Swedish mathematician and philosopher Per Martin-Löf. It consists of an analysis and a development of Brower's program for the constructive - intuitionistic mathematics.

A wider framework for the theory that we will develop is intuitionism as a philosophical school of thought on mathematics. It arose in the beginning of 20th century and its founder was the Dutch mathematician J.E.L.Brouwer, who tried to give his own solution to the foundational crisis of mathematics. Unlike other attempts with the same goal, intuitionism questions and rejects parts of classical mathematics, instead of trying to support the common mathematical practice of its time. It emphasizes on the examination of the nature of abstract mathematical constructions and demotes questions around the ontology of such constructions as philosophical concerns of less importance. For Brouwer mathematical reasoning is a human activity and mathematics is a language in which the various concepts are expressed. Concisely, intuitionism states that the only criterion of existence of mathematical objects is whether their conceptual creation is possible through constructive procedures, i.e. algorithms.

As a result, Intentional Type Theory treats mathematical objects as constructions. Every mathematical object is of one type. Better formulated, a mathematical object is always accompanied by its type. It is never just an object but an object of a specific type. So, types classify and categorize the constructions and can be seen as specifications of them, just like the types in programming languages.

A type is defined by describing what should we do to construct an object of that type. Put differently, a type is well defined if we fully understand what it means for an object to be of that type. For example, $\texttt{nat} \to \texttt{nat}$ is a type, not because we know a few numerical functions like the primitive recursive ones, but because we accept that we comprehend the concept of a numerical function *in general*. We should note that this condition is irrelevant from whether we are able to produce all the objects of the type or not.

In this chapter we will describe the way in which we can talk about these constructions inside ITT and how these can acquire a more *logical* interpretation from the idea of "Propositions as Types". The main part of the chapter consists of an enumeration of all basic constructions possible inside ITT as well as an analysis of the behavior or the objects of these types.

2.1 Judgments

Type Theory has only one primitive concept, that of types. Inside Type Theory we can produce *judgments*, things that we know, using inference rules. The most basic judgment in the theory is written in the form $a : A$ and we say that "a is of type A" or more loosely "a is an element of A". Intuitively, this means that the object a is a construction which is described - classified by the type A. So, we should not confuse this with the property of set membership of Set Theory and interpret this as $a \in A$ which is a proposition in First Order Logic. For instance, in Type Theory phrases like "if $a : A$ then it's not true that $b : B$" are not valid and we cannot also "disprove" $a : A$. On the contrary, in Type Theory every object has a type, by virtue of its nature. For example, in Set Theory when we say "let x be a natural number" then we mean "let x be an object and we suppose that $x \in \texttt{nat}$" while in Type Theory the judgment $x : \texttt{nat}$ is an atomic statement and it makes no sense to introduce a variable without mentioning its type.

The second type of judgment of Type Theory has to do with the notion of equality and it is called *judgmental* or *definitional equality*. We write $a \equiv b : A$ or simply $a \equiv b$. The intuition of this judgment is that the two objects are equal by definition. For example, if $f : \texttt{nat} \to \texttt{nat}$ is a function discribed by the formula $f(x) = x^2$, then we have that $f(3) \equiv 3^2$. Again, as before, there is no point in supposing or trying to disprove a judgmental equality. Whether two objects are definitionally equal or not is a matter of expansion of their definitions and it is an algorithmically decidable process.

2.2 Propositions as Types

A *proposition* in the sense of Intuitionistic Logic, instead of Classical Logic, is defined and put in the way that we are allowed to prove it. For instance the proposition:

"The number 42 is not a prime number"

is a proposition which we prove by giving two natural numbers greater than one and a computation that shows that their product equals 42. As every mathematical object is a construction, the same applies for proofs and they are classified as well by types. So, a proof of a proposition is nothing more than a construction of a program of that type. This constructive interpretation of propositions creates a correspondence between Intuitionistic Logic and Type Theory.

To illustrate this with an example we take the case of implication. The intuitionistic interpretation of the connective of implication defines that a proof of the proposition $A \implies B$ is a function that assigns every proof of A to a proof of B. In this way, the identity function is a proof of the proposition $A \implies A$. Similarly to this, we will see that this correspondence extends between more intuitionistic connectives and constructions of Type Theory.

Propositions	Types
$\top$	$\mathbf{1}$
$A \wedge B$	$A \times B$
$A \implies B$	$A \to B$
$\perp$	$\mathbf{0}$
$A \vee B$	$A + B$
$\forall x : A.B$	$\Pi x : A.B$
$\exists x : A.B$	$\Sigma x : A.B$

As a result, in the case of ITT there is no need of introducing any added notion of proposition as we can represent every proposition with a type, the type of all constructive proofs of this proposition. So, from now on, we will not make any distinction between the concepts type - proposition and between the phrases "a is an element of type A" and "a is a proof of proposition A".

In this point we would like emphasize the difference between this form of logic and its proofs and the Classical Logic and formal proofs. In Classical Logic our proofs and theorems are produced by a Hilbert-style deductive system, with logical, non logical axioms and inference rules that act upon the "data", the objects of our world, which are sets. In this case however, the deductive system exists inside the nature of the objects by the rules that define them and their behavior. This is a traditional principle in the way that Type Theory functions: the lack of any kind of outer logic and axioms because something like this opposes to the constructive nature of the system. In ITT, proofs are first class mathematical objects and conceptually identical with the data on which they refer, e.g. a proof of a property of natural numbers (the objects of type `nat`), the Euclidean division theorem, is an object itself of the type

$$\Pi m, n : \texttt{nat}.GT(M, 0) \to \Sigma q, r : \texttt{nat}.\texttt{Id}(mq + r, n) \times GT(m, r)$$

Simply, everything is an object of some type and solely its existence is a proof of the truth of the proposition that corresponds to that type. Moreover, the way the intuitionistic connectives work is defined by inference rules and by their computational properties, while in the case of Classical Logic they are defined through truth tables. As a result, ITT is free from more kinds of universal assumptions besides axioms. The various assumptions that might be needed to prove a proposition can be used in any particular case and thus the overall theory becomes stronger. A special case of this, is the Law of the Excluded Middle. In ITT there is no statement that the Law of Excluded Middle is globally true and it is used only wherever it is needed. On the contrary, in Classical Logic, this law is imposed by the rules of the truth tables of conjunction and negation in the foundations of the system.

2.3 Contexts

The judgments that we derive may sometimes depend on some hypotheses. For example, supposing $n, m : \texttt{nat}$ we can conclude that $n + m : \texttt{nat}$. Typically, a *context* is an ordered list of judgments some of which may depend on previous judgments in the list. We usually symbolize a context with Γ and we separate the hypotheses from the conclusion with the symbol $\vdash$. For example, $\Gamma, n : \texttt{nat}, m : \texttt{nat} \vdash n + m : \texttt{nat}$ or $\Gamma, x : A, y : A, p : \texttt{Id}_A(x, y) \vdash p^{-1} : \texttt{Id}_A(y, x)$. In the second case we see that there exist dependencies between the assumptions. We also notice that the objects x, y, p function exactly like ordinary mathematical variables. The intuitive meaning of a context and of $\vdash$ is that of entailment and so, for our theory to have a meaning, we would like that it satisfy some properties that make a form of entailment to be an entailment. These are the following:

1. Reflexivity

$$\Gamma, x : A, \Gamma' \vdash x : A$$

2. Transitivity

$$\frac{\Gamma, x : A, \Gamma' \vdash N : B \quad \Gamma \vdash M : A}{\Gamma\Gamma' \vdash [M/x]N : B}$$

3. Weakening

$$\frac{\Gamma \vdash M : A}{\Gamma\Gamma' \vdash M : A}$$

4. Contraction

$$\frac{\Gamma, x : A, y : A, \Gamma' \vdash N : B}{\Gamma, z : A, \Gamma' \vdash [x, y/z, z]N : B}$$

5. Exchange

$$\frac{\Gamma, x : A, y : A, \Gamma' \vdash N : B}{\Gamma, y : A, x : A, \Gamma' \vdash N : B}$$

When we have a system, such as Type Theory or Intuitionistic Logic, in which when added, these rules are compatible with the rest of the constructions and rules of the system we say that they are *admissible*. It is proved that the rules that we gave are admissible for the Type Theory that we will develop and the rules that we will introduce. It is worth mentioning that there have been studied systems in which we have forms of entailment where the rules of weakening, contraction or exchange are not admissible, but always, in order to have the meaning of entailment, the rules of reflexivity and transitivity should be admissible.

2.4 Basic Constructions

As mentioned earlier, types classify possible constructions in the framework of our system. So, in order to be able to make useful constructions and describe a wide range of objects we should give ways of creating new types. In Type Theory this is done by rules that describe the behavior of the objects of this type.

For every new type, we shall give a set of rules which define its syntax, the way new objects are created, their use and the computational meaning that we want them to have. So for every type, we have rules of

- Formation, that describe what is necessary for the formation of the new type.
- Introduction, that describe the way in which it is possible to construct objects of that type, using constructors.
- Elimination, which show us how we can use the objects of this type.
- Computation or β-reduction, which show the way the constructors interact with the elimination rules. The idea behind these rules comes from Gentzen's Inversion Principle which states that elimination is inverse of introduction. This, as we will see, is the essence of computation.
- Unicity or η-expansion, which express the uniqueness of the mappings to and from objects of the type, stating that every object is uniquely determined by the application of elimination rules on it and that it can be reconstructed by the application of the constructors on the results. They are in some sense dual to β-reduction since they show us the way a constructor acts upon the result of an elimination. These rules are also due to Gentzen's Unicity Principle.

2.4.1 The Unit Type

The Unit type, which is commonly written as **1**, is a simple type that has only one element, symbolised as $\langle\rangle$. So, since the type **1** is not created by any other types we simply have the judgment

$$\frac{}{\Gamma \vdash \mathbf{1} : \mathtt{type}}\ F-\mathbf{1}$$

as its formation rule. For the creation of an object of the type **1**, the only one that there is, there is no condition needed to be satisfied.

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}}\ I-\mathbf{1}$$

This type has no elimination rules and so no β-reduction rules. There is however an η-expansion rule, as an inverse of the introduction rule, which expresses the uniqueness of the element $\langle\rangle$.

$$\frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash M \equiv \langle\rangle : \mathbf{1}}\ \eta-\mathbf{1}$$

It is obvious that this type expresses the existence of a type with exactly one element by definition. It is corresponded with the proposition that is trivially true by definition, and whose truth doesn't need any assumptions. This is the proposition $\top$.

2.4.2 Products

Given two types A and B it is possible to construct the product $A \times B$. The intuitive meaning of this construction is the combination of the objects of the two types keeping the information of both objects, just like in the products of sets, groups, or topological spaces. The rules that describe the behavior of this type are the following. For the formation it is required that both A and B be types.

$$\frac{\Gamma \vdash A : \mathtt{type} \quad \Gamma \vdash B : \mathtt{type}}{\Gamma \vdash A \times B : \mathtt{type}}\ F-\times$$

For the creation and use of new objects of this type we have:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \ I - \times$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathtt{fst}(M) : A} \ E_1 - \times \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathtt{snd}(M) : B} \ E_2 - \times$$

As for the computational meaning of these constructions we have:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \mathtt{fst}\langle M, N \rangle \equiv M : A} \ \beta_1 - \times \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \mathtt{snd}\langle M, N \rangle \equiv N : B} \ \beta_2 - \times$$

and finally the Gentzen's Unicity Principle of these constructions:

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \langle \mathtt{fst}(M), \mathtt{snd}(M) \rangle \equiv M : A \times B} \ \eta - \times$$

2.4.3 Function Types

Given two types A and B we construct the type $A \to B$ of all functions from A to B. On the contrary to Set Theory, functions are not functional relationships but a primitive notion which is defined by the way the objects-functions behave. So let $f : A \to B$ be a function. Then f may be applied on every element $a : A$ giving a new element $f(a)$ in B which we call value of f on a. In Type Theory it is common to write fa instead of $f(a)$.

In order to construct a function we can introduce it by giving it a name, let f, and define $f : A \to B$ by giving an equation of the form $f(x) = \Phi$, where Φ may contain the variable x and be an element of B if we suppose that $x : A$. An other way to create an object of the type $A \to B$ is by using λ-*abstraction* without giving it a name. If $\Gamma, x : A \vdash \Phi : B$ like above, then we may write $\lambda(x : A)\Phi : A \to B$ and define the same function as before.

All these are described by the following rules. To be able to properly define a function type, the domain and the codomain should be well-defined types.

$$\frac{\Gamma \vdash A : \mathtt{type} \quad \Gamma \vdash B : \mathtt{type}}{\Gamma \vdash A \to B : \mathtt{type}} \ F\!- \to$$

While for λ-abstraction and function application on an object of the domain we have:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda(x : A)M : A \to B} \ I\!- \to$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B} \ E\!- \to$$

In order to express the computational behavior of objects-functions we introduce the β-reduction rule, which intuitively says that the computation of a function f on $N : A$ is nothing more than the substitution of all free recurrences of a variable x in f by the term N.

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda(x : A)M)(N) \equiv [N/x]M : B} \ \beta\!- \to$$

Finally, Gentzen's Unicity Principle expresses the fact that λ- abstraction is essentially the only way of creating functions.

$$\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash (\lambda(x : A)M(x)) \equiv M : A \to B} \ \eta\!- \to$$

For constructing functions of many variables we may create functions using as a domain a product of types ($f : A \times B \to C$) or, more commonly in Type Theory, we may use a technique called Currying (named after the mathematician Haskell Curry). According to it, a function with arguments in both A, B to C is a function with domain A and codomain the type of functions from B to C, i.e. the type $A \to (B \to C)$. It is important to note that the connective $\to$ is right commutative and so this is equivalent to $A \to B \to C$. So, a function taking two parameters can be defined using a double λ-abstraction: $\lambda(x : A).\lambda(y : B)\Phi$.

2.4.4 The Empty Type

Also, useful is the existence of the concept of the Empty Type, dual construction to the type **1**, commonly symbolized as **0**. Intuitively, it is a type that does not contain and cannot contain any construction and thus expresses the false proposition. Therefore, we can always build a function from the Empty Type to any type without giving any formula. This is because **0** has no evidence of truth. So there is no rule of introducing objects of this type and there is a rule of elimination to any other type.

$$\frac{}{\Gamma \vdash \mathbf{0} : \mathtt{type}}\ F-\mathbf{0}$$

$$\frac{\Gamma \vdash A : \mathtt{type}}{\Gamma, x : \mathbf{0} \vdash \mathtt{abort}(x) : A}\ E-\mathbf{0}$$

Since there are no items in **0** there are no β-reduction and η-expansion rules too.

This type is often useful when we want to express a negation or prove one. Assuming A is a type, we define the type of its negation as follows:

$$\neg A :\equiv A \to \mathbf{0}$$

Therefore, when we want to prove a negation it is enough to assume that the proposition is valid without the negation and come up with an object of the type **0**. This should be distinguished from the method of proof by contradiction of Classical Logic. There, in order to prove that something is true we assume that its negation is true and try to end up to something invalid. That is, in the language of Type Theory, to prove A we produce a construction of $\neg A \to \mathbf{0} \equiv \neg\neg A$ which is of course equivalent to A, in Classical Logic, but not by definition in Type Theory. In Type Theory a statement for which this is true, that $\neg\neg A \to A$, is called *stable* and there are propositions that are not stable.

2.4.5 Coproducts

Now, as before with the types **1** and **0**, we will define the dual type of products, coproducts. We symbolize the coproduct connective on two types A and B as $A+B$. Coproducts express the union of elements of two types while maintaining the information of the origin of each element. This is crucial mainly in light of Props as Types. If, for example, we want to present an element of $A+B$, then this should be either an element of A or an element of B, within $A+B$ through the appropriate constructor each time $\mathtt{inl} : A \to A+B$ and $\mathtt{inr} : B \to A+B$ respectively. This source information, since it is maintained, is available when creating mappings from coproducts into other types and allows the differentiation between objects coming from A or B. In other words, it is a form of pattern matching on the terms of A or B.
Hence, for the formation of this type we have:

$$\frac{\Gamma \vdash A : \mathtt{type} \quad \Gamma \vdash B : \mathtt{type}}{\Gamma \vdash A+B : \mathtt{type}}\ F-+$$

The introduction and elimination rules are

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{inl}(M) : A+B}\ I_1-+ \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathtt{inr}(M) : A+B}\ I_2-+$$

$$\frac{\Gamma, x : A \vdash P : C \quad \Gamma, y : B \vdash Q : C \quad \Gamma \vdash M : A+B}{\Gamma \vdash \mathtt{case}(x.P, y.Q)(M) : C}\ E-+$$

respectively. The computational content of pattern matching is obviously the following.

$$\frac{\Gamma, x : A \vdash P : C \quad \Gamma, y : B \vdash Q : C \quad \Gamma \vdash M : A}{\Gamma \vdash \mathtt{case}(x.P, y.Q)(\mathtt{inl}(M)) \equiv [M/x]P : C}\ \beta_1-+$$

$$\frac{\Gamma, x : A \vdash P : C \quad \Gamma, y : B \vdash Q : C \quad \Gamma \vdash M : B}{\Gamma \vdash \texttt{case}(x.P, y.Q)(\texttt{inr}(M)) \equiv [M/y]Q : C} \beta_2 - +$$

As for the η-expansion rule, very simply, whatever behaves like a case analyser over a coproduct is in fact a form of pattern matching

$$\frac{\Gamma, z : A + B \vdash P : C}{\Gamma, M : A + B \vdash [M/z]P \equiv \texttt{case}(x.[\texttt{inl}(x)/z]P, y.[\texttt{inr}(y)/z]P)(M) : C} \eta - +$$

Booleans As a special case of the above, we can define the Bool type as $\mathbf{1} + \mathbf{1}$, the type consisting of only two elements with unique information the distinction of one from the other. We can make this more clear by naming them $\texttt{tt} \equiv \texttt{inl}(\langle\rangle)$ and $\texttt{ff} \equiv \texttt{inr}(\langle\rangle)$. Moreover, we can name the pattern matching on them differently so that it is more natural as if ... then ... else, as in most functional programming languages.

$$\frac{\Gamma \vdash P : C \quad \Gamma \vdash Q : C \quad \Gamma \vdash M : A + B}{\Gamma \vdash \texttt{if } M \texttt{ then } P \texttt{ else } Q \equiv \texttt{case}(P, Q)(M) : C} E - \texttt{bool}$$

where P and Q are independent of $\langle\rangle$, which in any case is trivial.

2.4.6 Natural Numbers

It's time to look now how one of the most basic objects of Set Theory and mathematics in general is defined within ITT, the Natural Numbers. The approach is quite similar to that in axiomatic foundation of Peano's arithmetic. Firstly, there is a formation rule that defines the existence of a type nat (F-nat). The elements of this type is an element, 0, and then for each other object of nat we can create another one through a function $\texttt{suc} : \texttt{nat} \to \texttt{nat}$.

$$\frac{}{\Gamma \vdash \texttt{zero} : \texttt{nat}} I_0 - \texttt{nat} \qquad \frac{\Gamma \vdash M : \texttt{nat}}{\Gamma \vdash \texttt{suc}(M) : \texttt{nat}} I_s - \texttt{nat}$$

The way in which we can create a mapping from the nat type to any other type A is by a *recursive definition*. This means that, an object $P : A$ is required as the value of the mapping at zero and an object $n.x.Q : A$ as the value of $\texttt{suc}(n)$ given some $n : A$ and $x : A$ (the value of the mapping on n).

$$\frac{\Gamma \vdash M : \texttt{nat} \quad \Gamma \vdash P : A \quad \Gamma, n : \texttt{nat}, x : A \vdash Q : A}{\Gamma \vdash \texttt{rec}(P, n.x.Q)(M) : A} E - \texttt{nat}$$

The computation of the recursor is the obvious one.

$$\frac{\Gamma \vdash P : A \quad \Gamma, n : \texttt{nat}, x : A \vdash Q : A}{\Gamma \vdash \texttt{rec}(P, n.x.Q)(\texttt{zero}) \equiv P : A} \beta_0 - \texttt{nat}$$

$$\frac{\Gamma \vdash M : \texttt{nat} \quad \Gamma \vdash P : A \quad \Gamma, n : \texttt{nat}, x : A \vdash Q : A}{\Gamma \vdash \texttt{rec}(P, n.x.Q)(\texttt{suc}(M)) \equiv [M, \texttt{rec}(P, n.x.Q)(M)/n, x]Q : A} \beta_{\texttt{suc}} - \texttt{nat}$$

While similar to the η-expansion rule for coproducts we have

$$\frac{\Gamma, z : \texttt{nat} \vdash M : A \quad \Gamma \vdash [\texttt{zero}/z]M \equiv P : A \quad \Gamma, z : \texttt{nat} \vdash [\texttt{suc}(z)/z]M \equiv [z, M/n, x]Q : A}{\Gamma, z : \texttt{nat} \vdash M \equiv \texttt{rec}(P, n.x.Q)(z)} \eta - \texttt{nat}$$

Dependent Types One might wonder if it is possible to use Natural Numbers as presented above for the creation of proofs by induction. For example assuming that we have recursively defined an addition operation in nat, how can we prove that it is commutative? Thinking that the proposition we want to prove is a type (Props as Types) for each $n, m : \texttt{nat}$ there should be a proposition expressing the requested property.

In ITT this is achieved by introducing *dependent types*. A dependent type is a family $x.B$ of types indexed by elements of a type A. A very simple example is the family of types Seq

$$\Gamma, x : \texttt{nat} \vdash \texttt{Seq}(x) : \texttt{type}$$

which for each natural number n defines the type of finite sequences of natural numbers of length exactly n. That is, for each element of type `nat`, a separate type is defined with this parameter-element. It is also possible for the type $x.B$ to be defined without any dependence on the variable x. Then we have the constant family where for every element of the index type we get exactly the same type B.

Another very important example that will be examined later is that of families of Identity types of a type. Assume that A is a type and $a, b : A$. Then a proof of identity of the elements a, b is a construction of type $\texttt{Id}_A(a), b)$, i.e. the type of all "evidence" of identity of a, b. So, in the example above we are looking for an abel term such that $\Gamma, n, m : \texttt{nat} \vdash \mathit{abel} : \texttt{Id}_A(n+m, m+n)$.

Now that we have presented the existence of families of dependent types let's analyze how we can define mappings from natural numbers to such dependent families from `nat`. For each $n : \texttt{nat}$ we want a item in the type $[n/z]C$ where $z.C$ is a family of types indexed by the elements of `nat`.

$$\frac{\Gamma \vdash M : \texttt{nat} \quad \Gamma, z : \texttt{nat} \vdash C : \texttt{type} \quad \Gamma \vdash P : [\texttt{zero}/z]C \quad \Gamma, n : \texttt{nat}, x : [n/z]C \vdash Q : [\texttt{suc}(n)/z]C}{\Gamma \vdash \texttt{ind}[z.C](P, n.z.Q)(M) : [M/z]C} \; E-\texttt{nat}$$

The dependent variants of β and η reduction are similar.

2.4.7 Dependent Functions

In this subsection and in the following two we will present the dependent versions of some of the constructions we saw. Here, we will analyze the type of dependent functions. If we have a family of types $x.B$ dependent on a type A (for each $a : A$ $[a/x]B$ is some type) then a dependent function is a construction of an object $f(a) : [a/x]B$ for each $a : A$. Therefore, the rules for the formation of the type, for λ-abstraction and application are quite similar to the non-dependent case.

$$\frac{\Gamma \vdash A : \texttt{type} \quad \Gamma, x : A \vdash B : \texttt{type}}{\Gamma \vdash \Pi x : A.B : \texttt{type}} \; F-\Pi$$

$$\frac{\Gamma, a : A \vdash M : [a/x]B}{\Gamma \vdash \lambda(x : A)M : \Pi x : A.B} \; I-\Pi$$

$$\frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M(N) : [N/x]B} \; E-\Pi$$

It is important to note the dependence that exists between the type of the result and the argument of the function. This piece of information spreads at the level of types. In other respects, the behavior is quite similar to the case of simple functions. In fact, if B is the constant family, then $\Pi x : A.B \equiv A \to B$. The rules for β and η reduction are identical to the simple case but we list them here for the sake of completeness.

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda(x : A)M)(N) \equiv [N/x]M : [N/x]B} \; \beta-\Pi$$

$$\frac{\Gamma \vdash M : \Pi x : A.B}{\Gamma \vdash (\lambda(x : A)M(x)) \equiv M : \Pi x : A.B} \; \eta-\Pi$$

This type in the light of Props as Types expresses propositions of the form $\forall x : A.\Phi$, where A is some type and Φ a proposition concerning the objects of A.

2.4.8 Dependent Pairs

Pairs, that is, the objects of a product type, can also be generalized to dependent versions. In this case, there is dependence of the type of the second term of the pair on the first term. As before, if $x.B$ is a family of types indexed by a type A, then for a dependent pair of $\langle a, b\rangle$ it is true that $a : A$ and $b : [a/x]B$. Therefore in the form of rules whenever we have a dependent family of types we can form the type of the dependent product (Σ type).

$$\frac{\Gamma \vdash A : \texttt{type} \quad \Gamma, x : A \vdash B : \texttt{type}}{\Gamma \vdash \Sigma x : A.B : \texttt{type}}\ F - \Sigma$$

The syntax for pairs and their projections remains the same with increased meaning.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : [M/x]B}{\Gamma \vdash \langle M, N\rangle : \Sigma x : A.B}\ I - \Sigma$$

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \texttt{fst}(M) : A}\ E_1 - \Sigma \qquad \frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \texttt{snd}(M) : [\texttt{fst}(M)/x]B}\ E_2 - \Sigma$$

By looking at the rule of elimination of the second projection it becomes clear the dependence of the type of term $\texttt{snd}(M)$ on $\texttt{fst}(M)$. With a more "logical interpretation" of these two elimination rules, an object of $\Sigma x : A.B$ type is a construction of an object of the type A along with a proof that this object satisfies the property B. In general, in the intuitionistic context where we work this is called *constructional existence*. That is, in order to prove that there is an object that satisfies a property, we must have an (algorithmic) way of constructing this object and a proof that it satisfies the requested property. For this reason the type $\Sigma x : a.B$ is mapped to the existential quantifier of propositional logic.

We want the computational behavior of these pairs to be similar to the non-dependent case. So, we have:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : [M/x]B}{\Gamma \vdash \texttt{fst}\langle M, N\rangle \equiv M : A}\ \beta_1 - \Sigma \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : [M/x]B}{\Gamma \vdash \texttt{snd}\langle M, N\rangle \equiv N : [M/x]B}\ \beta_2 - \Sigma$$

and finally η-expansion remains the same

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \langle \texttt{fst}(M), \texttt{snd}(N)\rangle \equiv M : \Sigma x : A.B}\ \eta - \Sigma$$

2.4.9 Dependent Coproducts

For coproducts of types we will not change anything else but the way that we can create mappings that start from them. Like in the case of Natural Numbers, we will extend the recursion, case analysis for coproducts, to induction allowing the values of our functions to belong to different types depending on each argument. So, the new elimination rule in coproducts will be the following

$$\frac{\Gamma, z : A + B \vdash C : \texttt{type} \quad \Gamma, x : A \vdash P : [\texttt{inl}(x)/z]C \quad \Gamma, y : B \vdash Q : [\texttt{inr}(y)/z]C \quad \Gamma \vdash M : A + B}{\Gamma \vdash \texttt{case}(x.P, y.Q)(M) : [M/z]C}\ E - +$$

The new rule is much more expressive. If we have a family $z.C$ of dependent types on the terms of a type $A + B$, for each object $a : A$ we can construct an object $P : [\texttt{inl}(a)/z]C$ and respectively for each object $b : B$ an object $Q : [\texttt{inr}(b)/z]C$. Then there is the natural (dependent) function from $A + B$ into the type family $z.C$ defined by pattern matching.

2.4.10 Universes

In this section we will describe a very important concept, the introduction of which aims to increase the expressiveness of the theory. So far, we were using the judgment a : `type` although it is not one of the two main judgments of Type Theory that we presented in the corresponding section. Now this will become more accurate and formal with the introduction of Universe Types. A universe $\mathcal{U}$ is a type whose elements are types. One might wonder if this means that a universe can include itself, that is, if it is true that $\mathcal{U} : \mathcal{U}$. It turns out that if this were the case we could reproduce a version of a Set Theoretic paradox, e.g. the Burali-Forti paradox, and so the theory would be inconsistent making any type, including the **0** to have elements.

To avoid this we define a cumulative hierarchy of universes $\mathcal{U}_i$

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

where by cumulative we mean that every element-type that belongs to the $\mathcal{U}_i$ belongs to $\mathcal{U}_{i+1}$ as well. So, when we write that A is a type, we'll mean it belongs to some universe $\mathcal{U}_i$. Quite often we will avoid mentioning explicitly the level i of the hierarchy and we will simply write $A : \mathcal{U}$. That way it is possible to write even $\mathcal{U} : \mathcal{U}$ where the omitted indices would be $\mathcal{U}_i : \mathcal{U}_{i+1}$. This way of writing is often referred to as *typical ambiguity* and -although convenient- it may cause mistakes. It is customary to call the types within a given universe $\mathcal{U}$ small types.

Universes of each level are closed under the constructions we have seen so far. This means that the following are true

$$\frac{}{\texttt{nat} : \mathcal{U}} \qquad \frac{}{\mathbf{0} : \mathcal{U}} \qquad \frac{}{\mathbf{1} : \mathcal{U}}$$

$$\frac{A : \mathcal{U} \quad B : \mathcal{U}}{A \times B : \mathcal{U}} \qquad \frac{A : \mathcal{U} \quad B : \mathcal{U}}{A \to B : \mathcal{U}} \qquad \frac{A : \mathcal{U} \quad B : \mathcal{U}}{A + B : \mathcal{U}} \qquad \frac{A : \mathcal{U} \quad B : \mathcal{U}}{\Sigma x : A.B : \mathcal{U}} \qquad \frac{A : \mathcal{U} \quad B : \mathcal{U}}{\Pi x : A.B : \mathcal{U}}$$

With Universes we can see dependent families of types $x.B$ indexed by elements of a type A as functions $B : A \to \mathcal{U}$. To give an example of the expressive strength that the universes and dependent types provide we consider the following:
Since `bool` : $\mathcal{U}_i$ and `nat` : $\mathcal{U}_i$, we can define the dependent type $\lambda x.$`if` x `then nat else bool` : `bool` $\to \mathcal{U}_i$. This allows us to write programs such as

$$\texttt{if } M \texttt{ then } 17 \texttt{ else tt} : \texttt{if } M \texttt{ then nat else bool}$$

which type checks. This is impossible in most programming languages. Finally, we should note that the information in the argument is passed to the type of the result.

2.4.11 Identity Types

We now move on to one of the most essential types, the type of identity of two elements of a type. To begin with, we will give the definition of the type and then we will describe some of its most important properties.

Definition Let A be a type and $a, b : A$. Then a proof of the identity of a and b will of course be a program and therefore will meet the specification of the corresponding type. This type is the type of Identity of a, b as elements of A and is denoted as $\texttt{Id}_A(a), b)$ or $a =_A b$ or when there is no risk of confusion of the type of data simply $a = b$. As we can see it is not just a type but a family of types indexed by two elements of A, for each type A. The corresponding formation rule of this type is therefore the one we described

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \texttt{Id}_A(M, N) : \mathcal{U}} F - \texttt{Id}$$

Care must be taken not to confuse this kind of identity with judgmental equality. In that case, the equality between two terms is at the level of definitions and calculations and the judgment $a \equiv b$ is not a type. In the case of Identity types it is true that when two objects are identified (we say propositional equal to distinguish them) this means that the Identity type is not empty. That is, there is an object $p : \mathtt{Id}_A(a, b)$. As we have seen in the case of other types, this is very important. The $\mathtt{Id}_A(a, b)$ type can include multiple proofs "p" each with different information and have a more complex structure.

Obviously, we would like the relationship of Identity that we define to be at least reflexive. Therefore, the introduction rule should allow us to be able to build programs that are witnesses to Identity of an object with itself.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{refl}_A(M) : \mathtt{Id}_A(M, M)}\ I - \mathtt{Id}$$

Reflexivity ($\mathtt{refl}$) does exactly that and when there is no risk of confusion we will omit the type A or object M. Specifically, when we have $a \equiv b$, we also have $\mathtt{refl} : \mathtt{Id}_A(a, b)$ which type checks since the fact that $a \equiv b$ means that the type $\mathtt{Id}_A(a, b)$ is equal by definition to the type $\mathtt{Id}_A(a, a)$ which is the type of $\mathtt{refl}_A(a)$.

It makes sense, too, when we have two identical objects, that they satisfy the same properties, in a way. This means that identity should be the least reflexive relationship. That is, in any case a relationship is reflexive, then identical objects should satisfy it. This is a very reasonable requirement that one could have of a type of identity of objects. "Since they are identical, then the there should be no difference in any relationship that respects reflexivity." That's exactly what expresses the elimination rule and the operator $\mathtt{J}$.

$$\frac{\Gamma \vdash P : \mathtt{Id}_A(M, N) \quad \Gamma, x, y : A, z : \mathtt{Id}_A(x, y) \vdash C : \mathcal{U} \quad \Gamma, x : A \vdash Q : [x, x, \mathtt{refl}_A(x)/x, y, z]C}{\Gamma \vdash \mathtt{J}[x.y.z.C](x.Q)(P) : [M, N, P/x, y, z]C}\ E - \mathtt{Id}$$

This rule is also called *path induction* because of the homotopy interpretation that we will see later. Also, the $\mathtt{J}$ operator satisfies the following computational rule of β-reduction.

$$\frac{\Gamma \vdash M : A \quad \Gamma, x, y : A, z : \mathtt{Id}_A(x, y) \vdash C : \mathcal{U} \quad \Gamma, x : A \vdash Q : [x, x, \mathtt{refl}_A(x)/x, y, z]C}{\Gamma \vdash \mathtt{J}[x.y.z.C](x.Q)(\mathtt{refl}_A(M)) \equiv [M/x]Q : [M, M, \mathtt{refl}_A(M)/x, y, z]C}\ \beta - \mathtt{Id}$$

Properties Although we avoid proofs of properties of various types, we will present here the most important ones because they will appear next.

Identity is an equivalence relation Identity is reflexive by definition since for every $a : A$ we have $\mathtt{refl} : Id_A(a, a)$. To prove that it is symmetric we need for each $a, b : A$ and $p : \mathtt{Id}_A(a, b)$ to find an object $q : \mathtt{Id}_A(b, a)$. By path induction it is enough for each $x : A$ to find a $q : \mathtt{Id}_A(x, x)$. This is obviously $\mathtt{refl}$. So, we have

$$\mathtt{sym} : \Pi a, b : A.\mathtt{Id}_A(a, b) \to \mathtt{Id}_A(b, a)$$

where $\mathtt{sym} :\equiv \lambda a.\lambda b.\lambda p.\mathtt{J}[x, y, z.\mathtt{Id}_A(y, x)](x.\mathtt{refl}_A(x))(p)$.
To prove that it is transitive we assume that for $a, b, c : A$ we are given $p : \mathtt{Id}_A(a, b)$ and $q : \mathtt{Id}_A(b, c)$ and we search for a $r : \mathtt{Id}_A(a, c)$. If we look at it differently, for given $c : A$ we want for each $x, y : A$ and $z : \mathtt{Id}_A(x, y)$ a function with type $\mathtt{Id}_A(y, c) \to \mathtt{Id}_A(x, c)$. Again, by path induction it is enough to find such a function for $x \equiv y$ and $z \equiv \mathtt{refl}$. This is the identity function. Consequently

$$\mathtt{trans} : \Pi a, b, c : A.\mathtt{Id}_A(a, b) \to \mathtt{Id}_A(b, c) \to \mathtt{Id}_A(a, c)$$

where $\mathtt{trans} :\equiv \lambda a.\lambda b.\lambda c.\lambda p : \mathtt{Id}_A(a, b).\mathtt{J}[x, y, z.\mathtt{Id}_A(y, c) \to \mathtt{Id}_A(x, c)](x.\lambda w.w)(p)$.
Notice that the definitions of $\mathtt{sym}$ and $\mathtt{trans}$ satisfy some definitional equalities, thanks to the β-reduction of $\mathtt{J}$. Specifically

$$\mathtt{sym}(M)(M)(\mathtt{refl}) \equiv \mathtt{refl}$$

$$\mathtt{trans}(M)(M)(N)(\mathtt{refl})(Q) \equiv Q$$

If we had defined these two functions differently, that is, if we had given other evidence of symmetry and transitivity, then these would satisfy some other computational properties. This is typical of constructivel mathematics and is referred to as *Proof Relevance*. Finally, when $a, b, c : A$ are known, we write $\mathtt{sym}(a)(b)(p) \equiv p^{-1}$ and $\mathtt{trans}(a)(b)(c)(p)(q) \equiv p \cdot q$.

Functions respect Identity A very reasonable behavior of the Identity type that we defined would be the functions to respect it (Simple Functionality). That is, equal arguments correspond to equal results. Let $A, B : \mathcal{U}$ types, $f : A \to B$ and $a, b : A$. By path induction, it is enough to show that if $x \equiv a \equiv b$ then we have a proof of identity of $f(a)$ and $f(b)$. That's of course `refl`. So, we define

$$\Gamma, f : A \to B, x, y : A \vdash \mathtt{ap}_f : \mathtt{Id}_A(x, y) \to \mathtt{Id}_B(fx, fy)$$

as $\mathtt{ap}_f :\equiv \lambda p.\mathtt{J}[x, y, z.\mathtt{Id}_B(fx, fy)](x.\mathtt{refl}_B(fx))(p)$. Because of the β-reduction of the operator `J` we have that it respects reflexivity judgmentally: $\mathtt{ap}_f(\mathtt{refl}_A(x)) \equiv \mathtt{refl}_B(fx)$.

Transport Let $x.B$ be a family of types indexed by the objects of A, $a, b, : A$ and $p : \mathtt{Id}_A(a, b)$. Then, we would expect the types $[a/x]B$ and $[b/x]B$ to contain the "same" objects. That is, there should be a function $\mathtt{tr}[x.B](p) : [a/x]B \to [b/x]B$ which sends every object of $[a/x]B$ to the corresponding one in $[b/x]B$. With proof by path induction it is enough to find out what happens in the case where $x \equiv a \equiv b$. Then the most natural correspondence between $[x/x]B$ and $[x/x]B$ is the identity function.

$$\Gamma, a, b : A \vdash \mathtt{tr}[x.B] : \mathtt{Id}_A(a, b) \to [a/x]B \to [b/x]B$$

with $\mathtt{tr}[x.B] :\equiv \lambda p.\mathtt{J}[x, y, z.[x/x]B \to [y/x]B](x.\lambda w.w)(p)$. This process of sending objects of two types over an identity path $p : \mathtt{Id}_A(a, b)$ is called transport and sometimes it is symbolized as $\Gamma, v : [a/x]B \vdash p_*(v) : [b/x]B$. If the path is `refl` then the resulting transport is judgmentally equal to the identity function: $\mathtt{tr}[x.B](\mathtt{refl}) \equiv \mathtt{id}$.

Path over path Let's now look at a generalization of the concept of Simple Functionality and $\mathtt{ap}_f$ operator, for dependent functions. If we have a type family $x.B$ dependent on the elements of a type A, or using the universe, a function $B : A \to \mathcal{U}$, a dependent function $f : \Pi x : A.B$, two elements $a, b : A$ and one proof of identity $p : \mathtt{Id}_A(a, b)$ between them we would like a proof of identity between the elements fa and fb. However, this is not possible because $fa : [a/x]B$, $fb : [b/x]B$ and in general the types $[a/x]B$ and $[b/x]B$ are not equal by definition. So the closest we can get is to prove the identity between $\mathtt{tr}[x.B](p)(fa) : [b/x]B$ and $fb : [b/x]B$ which are elements of the same type. That is, the corresponding element of fa inside $[b/x]B$ by transporting over the path p. This proof is called `dap`, is of type

$$\Gamma, f : \Pi x : A.B, a, b : A \vdash \mathtt{dap}_f : \Pi p : \mathtt{Id}_A(a, b).\mathtt{Id}_{[b/x]B}(p_*(fa), fb)$$

and is proved by path induction $\mathtt{dap}_f :\equiv \lambda p.\mathtt{J}[x, y, z.\mathtt{Id}_{[y/x]B}(z_*(fx), fy)](x.refl(fx))(p)$. Of course, quite identical is the proof if instead of transporting fa we transferred fb. As the type $\mathtt{Id}_{[b/x]B}(p_*(fa), fb)$ is somehow not very symmetric we sometimes prefer a simpler syntax: $fa =_p^{x.B} fb$. It is obvious that if $fa =_p^{x.B} fb$ then $fb =_{p^{-1}}^{x.B} fa$.

Function Extensionality In this paragraph we deal with equality in a particular type, types of functions. When are two functions $f, g : A \to B$ equal? According to classical Set Theory and the definition of functions as their graph, two functions are the same when for every point in their domain they have the same values. This is called *Function Extensionality*. In Type Theory we would write

$$\frac{\Gamma \vdash f, g : A \to B \quad \Gamma, x : A \vdash p : \mathtt{Id}_B(fx, gx)}{\Gamma \vdash \_ : \mathtt{Id}_{A \to B}(f, g)}$$

But in the case of ITT, what we have is against this expectation of our intuition. Martin-Löf has shown that this does not apply in ITT. For example, for the usual inductive definitions of addition and multiplication in the natural numbers it is very easily proved that

$$n : \texttt{nat} \vdash p : \texttt{Id}_{\texttt{nat}}(n + n, 2 * n)$$

but

$$\nexists q : \texttt{Id}_{\texttt{nat}\to\texttt{nat}}(\lambda n.n + n, \lambda n.2 * n)$$

In order to bypass this weakness and be able to do within ITT math that respect Function Extensionality there exist many different variations. One of them is the introduction of an axiom in our theory, the *Axiom of Extensionality*, i.e. the postulation of the existence of a term `funext`, without any constructive definition, which satisfies exactly what we need

$$\texttt{funext} : \Pi f, g : A \to B.((\Pi x : A.\texttt{Id}_B(fx, gx)) \to \texttt{Id}_{A\to B}(f, g))$$

This gives a solution to the issue of function extensionality but creates some other problems. To begin with, adding and assuming axioms is against the nature and tradition of Type Theory. The reason for this is precisely the non-constructive nature of axioms. Unfortunately the problem is not just philosophical. Adding objects to a type whose behavior is not determined by their construction has direct impact on their computational behavior. In the case of `funext` this is immediately felt when we do path induction for a path resulting from its use. For instance, let $C : \Pi x, y : A \to B.\Pi z : \texttt{Id}_{A\to B}(x, y).\mathcal{U}$ be a property and $Q : \Pi x : A \to B.C(x)(x)(\texttt{refl})$. Then, in what way is a term of the form $\texttt{J}[x, y, z.C](x.Q)(\texttt{funext}(f)(g)(H))$ computed? It is a term with a correct type which in ITT is not calculated, i.e. it is stuck. If we are not interested in the computational content of the theory but only in the deductive part then there is no reason not to integrate the Axiom of Extensionality. But if we want to treat Type Theory as a theory of computation then it is logical to question this. As we will see later, with an extension of ITT to a new theory, Cubical Type Theory, `funext` appears as a theorem with a proof- program and thus acquires its computational meaning.

Chapter 3

HoTT - Homotopy Type Theory

Homotopy Type Theory is a new branch of mathematics which combines several fields in a very unexpected way. On the one hand, Homotopy Theory is a branch of Algebraic Topology and Homological Algebra and is directly related to Higher Category Theory. Someone could say that the most natural language for Homotopy Theory is Category Theory, since it can codify and generalize its basic constructions. On the other hand, Type Theory is, as we have seen, a branch of Mathematical Logic and Computer Science. In addition to the impressive connection of these branches, Homotopy Type Theory brings new ideas on the Foundation of Mathematics: the *Axiom of Univalence* and *Higher Inductive Types*. The first axiom allows us to identify isomorphic structures, which is in the daily practice of mathematicians, while higher inductive types give us a more logical description of basic spaces and structures of Homotopy Theory such as circles, tori, cylinders, truncations, etc. in a very abstract way.

Homotopy Type Theory interprets Type Theory homotopically. In Homotopy Type Theory we treat types as spaces, like those in Homotopy Theory, or as higher groupoids, and logical constructions, such as the product $A \times B$, as homotopy universal constructions on these spaces. Homotopy Theory studies the properties of topological spaces and mappings which are invariant under *homotopy equivalence*. A homotopy between two continuous mappings $f, g : A \to B$ is a continuous function $H : A \times [0, 1] \to B$ such that $H(x, 0) = f(x)$ and $H(x, 1) = g(x)$, we symbolize $f \simeq g$. It is somehow a continuous deformation of f in order to get identified with g. Two spaces are called *homotopy equivalent* if there are two continuous mappings $f : A \to B$ and $g : B \to A$ such that $f \circ g \simeq 1_B$ and $g \circ f \simeq 1_A$, and we write $A \simeq B$. Homotopy equivalent spaces have the same algebraic invariants (fundamental group, homology, etc.) and we say that they have the same *Homotopy Type*.

Let's briefly see what this interpretation implies in the most basic concepts. The meaning of $a : A$, a is a term of the type A, except for its usual interpretation will also have a homotopy interpretation. Specifically, we would also think about it as *a is an element of the space A*. Similarly, functions $f : A \to B$ will be treated as continuous functions between the spaces A and B.

These spaces as we have said should be considered purely homotopically and not topologically. In this way we achieve an abstraction and we do not deal with the details of topology, such as these of open sets or convergence. It is a form of abstraction similar to that which occurs in the study of geometry. When we study Euclidean Geometry we make a "synthetic" view of concepts as opposed to Analytic Geometry where we have a "detailed" view with points and lines being sets in a specific way within $\mathbb{R}^3$. Therefore, it would be more accurate to say that we interpret types as ∞-groupoids.

The key new idea is the homotopy interpretation of the logical concept of the identity $a = b$ of two objects $a, b : A$. In the homotopy context an identity object is a *path* $p : a \rightsquigarrow b$ from a to b within the space A. Therefore, two functions can be identified if there is a Homotopy between them, that is, for each $x : A$ a path $p_x : f(x) \rightsquigarrow g(x)$ (`funext`). The type $\mathtt{Id}_A(a, b)$ of the identity between two objects of a type is non other than the path space with edges a and b inside A. Also, the isomorphism of two spaces, that is, functions from both types whose compositions at every point are identities, are precisely the definition we have given of the homotopy equivalence of two spaces.

In this chapter we will try to shed light on the homotopy nature of types by analyzing their ∞-groupoid structure and then describe the two new ideas that extend Intentional Type Theory in Homotopy Type Theory, namely the Univalence Axiom and Higher Inductive Types. As it is not possible in a brief description of the field to include everything that is possibly developed in this context, we will present selective topics based on what is necessary for working on the subject of this thesis.

3.1 Higher Groupoids

We have discussed the logical content of Type Theory and the idea of Props as Types. In this subsection we will present the relationship that exists between Type Theory and Category Theory, and particularly with Homotopy Type Theory.

Categories Category Theory is a branch of mathematics which provides a unifying language for various other disciplines. It tries to create mathematical objects that express common patterns and relationships in several branches at the same time, making it easier to study similarities of these systems. Category Theory deals with mathematical abstraction in general.

A *directed graph* is a set of objects $\mathcal{O}$, a set of arrows or morphisms or mappings $\mathcal{A}$ and two functions

$$\texttt{dom}, \texttt{cod} : \mathcal{A} \to \mathcal{O}$$

In this graph, the set of pairs of arrow that can be composed is the set

$$\mathcal{A} \times_{\mathcal{O}} \mathcal{A} :\equiv \{\langle g, f\rangle | g, f \in \mathcal{A} \text{ and } \texttt{dom}g = \texttt{cod}f\}$$

A category is a graph with two additional functions, the identity arrow for each object

$$\texttt{id} : \mathcal{O} \to \mathcal{A},\ c \mapsto \texttt{id}_c$$

and mappings' composition

$$\circ : \mathcal{A} \times_{\mathcal{O}} \mathcal{A} \to \mathcal{A},\ \langle g, f\rangle \mapsto g \circ f \text{ or } gf$$

for which we have that

$$\texttt{dom}(\texttt{id}_c) = \texttt{cod}(\texttt{id}_c) = c$$

$$\texttt{dom}(gf) = \texttt{dom}(f) \text{ and } \texttt{cod}(gf) = \texttt{cod}(g)$$

It is quite common in Category Theory to avoid the strict formulations of this type and use diagrams with arrows. So, for example, for a mapping $f \in \mathcal{A}$ with $\texttt{dom}f = a \in \mathcal{O}$ and $\texttt{cod}f = b \in \mathcal{O}$ we simply write $a \xrightarrow{f} b$. Also, the symbolism with $\mathcal{O}$ and $\mathcal{A}$ is often abandoned and if $\mathcal{C}$ is a category we just write $a \in \mathcal{C}$ instead of $a \in \mathcal{O}$ and $f \in \mathcal{C}$ instead of $f \in \mathcal{A}$. We also symbolize the set of arrows between two objects $a, b \in \mathcal{C}$ with $\texttt{Hom}(a, b)$. The two operations satisfy two additional axioms.

Commutativity If $a, b, c, d \in \mathcal{C}$ and $a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{h} d$ then $h \circ (g \circ f) = (h \circ g) \circ f$.

Unit laws If $a, b, c \in \mathcal{C}$ and $a \xrightarrow{f} b \xrightarrow{g} c$ then $\texttt{id}_b \circ f = f$ and $g \circ \texttt{id}_b = g$

Groupoids In the context of categories it is very easy to represent the structure of a group in a very simple and natural way. We can imagine the group like a category $\mathcal{C}$ with a single object $a \in \mathcal{C}$. We correspond the elements of the group to the mappings from a to a in a way that respects the operation of the group (which is interpreted as morphism composition) and the identity element of the group (which is interpreted as $\texttt{id}_a$). A *groupoid* is a generalization of this idea into one category with more than one objects.

More specifically, a groupoid is a category such that for each arrow $f : a \to b$ there is another arrow $f^{-1} : b \to a$ for which the *Laws of Groupoids* are satisfied:

$$f \circ f^{-1} = \texttt{id}_b \text{ and } f^{-1} \circ f = \texttt{id}_a$$

Category theory is capable of giving semantics and of being a model for Type Theory as various constructions have their corresponding ones to Category Theory (products-coproducts, initial-final objects, etc.). In fact, different versions of Type Theory have been matched with different kinds of Categories. In the case of ITT it is very easy to match types with groupoids. If we consider each type as category then the arrows-mappings between the objects are the objects of the Identity type between the objects of the type, the paths from one to the other. Indeed for each object we have an Identity path that begins and ends at itself, `refl`, for every two paths $p : \mathtt{Id}_A(a, b)$ and $q : \mathtt{Id}_A(b, c)$ (which can be joined together) exists a path $p \cdot q : \mathtt{Id}_A(a, c)$. It is proved by path induction that these satisfy transitivity and unit laws. Thus, each type does indeed acquire a categorical interpretation.

However, we should reconsider the above correspondence and notice something really impressive that arises because of the constructive nature of Type Theory and the treatment of proofs as first-class objects. When we say that there is a proof by path induction of groupoid properties for paths in a type, let $\mathtt{refl} \cdot p = p$ (left law of the unit), this means that there exist a construction α that satisfies the following

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash a, b : A \quad \Gamma \vdash p : \mathtt{Id}_A(a, b)}{\Gamma \vdash \alpha : \mathtt{Id}_{\mathtt{Id}_A(a,b)}(\mathtt{refl}_A(a) \cdot p, p)}$$

We observe that the Identity type can be defined inductively for higher and higher dimensions of ”paths” as in Homotopy Theory. In this case we have a path between two paths: $\mathtt{refl} \cdot p$ and p. This of course is none other than a surface in the type/space A. That is, the type of Identity does not just induce an one-dimensional structure in type A, the structure of a groupoid, but a structure of higher dimensions, that of an ∞-*groupoid*.

An ∞-*category* is a generalization of the concept of category. As in any category there are 1-arrows between objects, in a 2-category we have 1-arrows between objects and 2-arrow between 1-arrows, in a ∞-category we can have k-arrows between $(k - 1)$-arrows, for each $k = 1, 2, \ldots$. An ∞-groupoid is therefore an ∞- category which satisfies the laws of groupoids in every dimension.

Finally, we should mention that in this categorical approach to types the functions are functors. Indeed, we have seen that it is true that

$$\mathtt{ap}_f(\mathtt{refl}_A(x)) = \mathtt{refl}_B(fx)$$

and it turns out that these identities also apply:

$$\mathtt{ap}_f(p^{-1}) = \mathtt{ap}_f(p)^{-1}$$

$$\mathtt{ap}_f(p \cdot q) = \mathtt{ap}_f(p) \cdot \mathtt{ap}_f(q)$$

As before though, the fact that these identities are true means that there is a construction-path of the appropriate Identity type. So, to emphasize this, it is often said that functions respect the groupoid structure *up to higher dimensional homotopy*.

3.2 Univalence Axiom

In this section we present how HoTT deals with identity between types instead of identity of objects within a type. First, we give the definition of type equivalence used in HoTT, the *Homotopy Equivalence*.

Homotopy Equivalence First, we introduce a notation for the type of *homotopies* between two functions $f, g : A \to B$. We set $f \simeq g :\equiv \Pi(a : A).fa =_B ga$. It is easy to prove that this is an equivalence relation, which satisfies the very important property of *naturality*: if $H : f \simeq g$ then the homotopy H respects the paths between elements of the A, is *dependently functorial on the elements* $x : A$. This means that the following diagram commutes.

$$\begin{tikzcd}
f(a) \arrow[r, Rightarrow, "H(a)"] \arrow[d, Rightarrow, "\mathsf{ap}_f(p)"] & g(a) \arrow[d, Rightarrow, "\mathsf{ap}_g(p)"] \\
f(a') \arrow[r, Rightarrow, "H(a')"] & g(a')
\end{tikzcd}$$

Now, to define the equivalence of two types $A, B : \mathcal{U}$ we define the type $A \simeq B$ consisting of the function $f : A \to B$ along with the following data:

- A function $g : B \to A$
- A homotopy $g \circ f \simeq \mathtt{id}_A$, i.e. a proof of $\alpha : \Pi(a : A).g(f(a)) =_A a$
- A function $h : B \to A$
- A homotopy $f \circ h \simeq \mathtt{id}_B$, i.e. a proof of $\beta : \Pi(b : B).f(h(b)) =_B b$

So, the type of the equivalences of the types A and B is defined as $A \simeq B :\equiv \Sigma(f : A \to B).\mathsf{isEquiv}(f)$ where

$$\mathsf{isEquiv}(f) :\equiv (\Sigma(g : B \to A).g \circ f \simeq \mathtt{id}_A) \times (\Sigma(h : B \to A).f \circ h \simeq \mathtt{id}_B)$$

From the way that equivalence was defined we realize that if two types are equivalent then they have the same structure of paths in all dimensions.

Univalence Axiom If we have two types $A, B : \mathcal{U}$ then we can define the type $\mathtt{Id}_{\mathcal{U}}(A, B)$ of all paths in the universe between A, B. It is very natural to define a function

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

which gives a function $f : A \to B$ that carries over an identity path in $\mathcal{U}$ elements of A to B. We would like that `idtoeqv` be an equivalence. This is exactly what *Univalence Axiom* gives us, which is due to Vladimir Voevodsky. More specifically, for each $A, B : \mathcal{U}$ we have that

$$(A \simeq B) \simeq (A =_{\mathcal{U}} B)$$

Based on this axiom we can *identify equivalent types*. We can therefore analyze this equivalence in the parts that make it up:

- An introduction function for the type $A =_{\mathcal{U}} B$:

$$\mathtt{ua} : (A \simeq B) \to (A =_{\mathcal{U}} B)$$

- An elimination function:

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

- A computation rule:

$$\mathtt{tr}[X.X](\mathsf{ua}(f))(x) = fx$$

- A unicity rule $p : A =_{\mathcal{U}} B$:

$$p = \mathsf{ua}(\mathtt{tr}[X.X](p))$$

3.3 Inductive Types

In this section we will look at the last part of ITT's constructions, inductive types. Inductive types take the idea of types that support recursion and induction, just like natural numbers, in a completely general context. Intuitively, to define an inductive type X it is required to give a finite number of constructors, functions with values in X, and objects of X may be considered freely generated by these constructors. The constructors can have multiple arguments (or none, i.e. just indicate the existence of an element of X) and from the type X itself.

Many of the types we have seen can be defined in an inductive way. For example type `bool` can be given with two constructors with no arguments

- $\mathtt{tt} : \mathtt{bool}$
- $\mathtt{ff} : \mathtt{bool}$

Two examples where constructors take arguments are the coproduct of two types A, B

- $\mathtt{inl} : A \to A + B$
- $\mathtt{inr} : B \to A + B$

and the product

- $\langle -, - \rangle : A \to B \to A \times B$

Whereas, an example where the constructor takes arguments from the defined type is the Natural Numbers when defined as an inductive type.

- $\mathtt{zero} : \mathtt{nat}$
- $\mathtt{suc} : \mathtt{nat} \to \mathtt{nat}$

The way we construct functions $f : X \to P$, where X is an inductive type, is a recursion principle. For each constructor and for any of its arguments we give an object of P, allowing us to call f recursively on the constructor's arguments. For example, to define a function from `bool` to any type P it suffices to give two items $e_{\mathtt{tt}}, e_{\mathtt{ff}} : P$ and so we have defined a function $f = \mathtt{rec}_{\mathtt{bool}}(e_{\mathtt{tt}}, e_{\mathtt{ff}}) : \mathtt{bool} \to P$ which additionally satisfies the following equalities:

$$\mathtt{rec}_{\mathtt{bool}}(e_{\mathtt{tt}}, e_{\mathtt{ff}})(\mathtt{tt}) \equiv e_{\mathtt{tt}} : P \qquad \mathtt{rec}_{\mathtt{bool}}(e_{\mathtt{tt}}, e_{\mathtt{ff}})(\mathtt{ff}) \equiv e_{\mathtt{ff}} : P$$

In the case where we have a constructor with arguments, we need to take them into account and give an element for each case of argument, but also for every result of f when it takes as an argument one of the arguments of the constructor. If, for example, we define inductively the type of linked lists of objects from a type A as follows:

- $\mathtt{nil} : \mathtt{list}(A)$
- $\mathtt{cons} : A \to \mathtt{list}(A) \to \mathtt{list}(A)$

then to define a function $f : \mathtt{list}(A) \to P$ we need to find $e_{\mathtt{nil}} : P$ and $e_{\mathtt{cons}} : A \to \mathtt{list}(A) \to P \to P$. That is, we should determine the result of the function for the empty list `nil` and for the list $\mathtt{cons}(\mathtt{x}, \mathtt{xs})$ if we know the result of $f(\mathtt{xs})$. In this case we have that $f(\mathtt{nil}) \equiv e_{\mathtt{nil}}$ and $f(\mathtt{cons}(\mathtt{x}, \mathtt{xs})) \equiv e_{\mathtt{cons}}(\mathtt{x}, \mathtt{xs}, f(\mathtt{xs}))$.

In a similar way we can define dependent functions by induction. To define a function $f : \Pi x : X.B$ where $B : X \to \mathcal{U}$ a family of types and X an inductive type we give for each constructor, for each of its arguments and for each possible value of f on the constructor's arguments a value in $B[x]$. If we consider that we have defined the natural numbers inductively and we have a family $E : \mathtt{nat} \to \mathcal{U}$ then to define a function $f : \Pi n : \mathtt{nat}.E$ it is enough to have an element $e_{\mathtt{zero}} : E(\mathtt{zero})$

for the constructor `zero` and an element $e_{\mathtt{suc}} : \Pi n : \mathtt{nat}.E(n) \to E(\mathtt{suc}(n))$ for the constructor `suc`. Then f satisfies the following equalities:

$$f(\mathtt{zero}) \equiv e_{\mathtt{zero}} : E(\mathtt{zero}) \qquad f(\mathtt{suc}(n)) \equiv e_{\mathtt{suc}}(n, f(n)) : E(\mathtt{suc}(n))$$

The presentation we made about inductive types is more descriptive rather than formal and complete unlike the previous types of ITT. For example, the standard presentation of W-types, proofs of uniqueness, categorical semantics such as F-algebras, coinductive types, etc are missing. The reason for this is that this would be long and also the subject of inductive types is not directly related to the topic of work. Nevertheless, a brief reference was intended to make a smooth connection with the topic of HoTT's higher inductive types and for completeness. For a more complete analysis we refer to the 5th chapter of [Univ13].

3.4 Higher Inductive Types

In this section we will present the second addition of Homotopy Type Theory compared to other theories, Higher Inductive Types. Higher Inductive Types are a general form of definition of new types based on some constructors. However, unlike the simple inductive types, in their definitions we can add constructions of paths of any dimension. That is, in addition to the points of space we can impose the existence of relations of identity between them which in the relevance proof environment of Type Theory are specific objects, paths. One might think of these types as free algebraic structures for the definition of which we give the generators who produce the elements and also the relationships that these satisfy.

Typically, to define such types we use the terminology 0-cells for the objects of the space, 1-cells for the paths between objects, 2-cells for the paths between 1-cells (the surfaces of the space), etc. Therefore, to define such a type we list constructors of cells of all dimensions. For example, to define the higher inductive type of the circle $\mathbb{S}^1$ we must give a point of the circle, `base`, as 0-cell and a path which goes from `base` to `base`, as a 1-cell. So, we write

- $\mathtt{base} : \mathbb{S}^1$
- $\mathtt{loop} : \mathtt{base} =_{\mathbb{S}^1} \mathtt{base}$

With this definition we also placed a new object in the type $\mathtt{Id}_{\mathbb{S}^1}(\mathtt{base}, \mathtt{base})$ which is not necessarily equal to any other. In fact, it turns out that the `loop` path is not equal with the path $\mathtt{refl}(\mathtt{base})$ and therefore the type $\mathbb{S}^1$ is not equal to the type **1**. Moreover, a significant difference from simple inductives types comes from the inherent groupoid structure that all types have. When we explicitly create a `loop` path through the definition, we also implicitly create various other paths, such as $\mathtt{loop} \cdot \mathtt{loop}$, $\mathtt{loop} \cdot \mathtt{loop} \cdot \mathtt{loop}$, $\mathtt{loop}^{-1}$,... which are all different from each other.

Also note that as the constructors of points and paths of the type are functions, and therefore as we have seen functors, they respect the structure of their domain in each dimension. If a type B is inductively defined by a constructor $A \to B$, then paths between elements of the A imply the existence of paths between the data created by the specific constructor in B. The same also applies to higher dimensions. If we have a path constructor $A \to x =_B y$ then the paths in A are transported through the constructor on paths in $x =_B y$ and therefore on 2-paths in B.

It's really impressive how simply can spaces be defined with these tools. In the case of the definition of the circle we have omitted all geometric and topological properties that it has with its usual definitions as a subspace of $\mathbb{R}^2$ or $\mathbb{C}$ and we have only focused on its homotopy or categorical characteristics. This abstraction, allows us to make it easier to study these objects focusing only on their *universal properties*, which are expressed through the recursion and induction principles of these types.

Recursion and Induction We presented how it is possible to define inductive types with a higher groupoid structure, that is, we gave the rules for the introduction of their data. Now, we must also give rules of elimination, or how it is possible to use objects of these types. These rules as in the case of simple inductive types are the known recursion and induction shcemas.

To correctly define functions on inductive types with no higher dimensions we gave in detail for each constructor and for any possible arguments the result we would like the function to have in the codomain. This is exactly what we will do now, but taking into account the path constructors. We should determine its behavior on 0-cells, and then determine its behavior over 1-cells so that they be consistent with the previous points and so on for higher dimensions. We should somehow identify the image of our function with the elements of the codomain with similar structure with that of the higher inductive type. More specifically, after we give the values of the function $f : A \to B$ we want to define, on the 0-cells of A according to the constructors and what we have seen, then we should determine how f carries the 1-cells between the points which have already been defined, that is, to give the value of $\mathtt{ap}_f$, and $\mathtt{dap}_f$ for the case of a dependent functions, for these points. Then we continue in all dimensions for all path constructors.

For example, suppose the type A which is defined as follows:

- $a_{-2}, a_{-1} : A$
- $a : \mathtt{nat} \to A$
- $p_{-1} : a(\mathtt{zero}) =_A a_{-1}$
- $p_{-2} : a(\mathtt{zero}) =_A a_{-2}$
- $p : \Pi n : \mathtt{nat}.a(\mathtt{zero}) =_A a(n)$

Then to define a function recursively from A we must provide the following information

- $f(a_{-2}) : B$
  $f(a_{-1}) : B$
  two points in B for the first two constructors
- $n : \mathtt{nat} \vdash f(a(n)) : B$
  one point for each natural number
- $\mathtt{ap}_f(p_{-2}) : f(a(\mathtt{zero})) =_B f(a_{-2})$
  $\mathtt{ap}_f(p_{-1}) : f(a(\mathtt{zero})) =_B f(a_{-1})$
  the way f carries the paths p_{-2}, p_{-1} with respect to their ends
- $n : \mathtt{nat} \vdash \mathtt{ap}_f(p(n)) : f(a(\mathtt{zero})) =_B f(a(n))$
  the way f carries the paths of the constructor p

In case of induction we operate in a similar way. In other words, if $B : A \to \mathcal{U}$ is a family of types, then since the objects that we will define as values of f on the 0-cells of A will belong to different types. we need the paths which will be carried by f (1-cells, 2-cells,...) be paths over paths (dependent case). In the above example we would have exactly the same way of defining f by changing paths to paths over paths.

- $f(a_{-2}) : B(a_{-2})$
  $f(a_{-1}) : B(a_{-1})$
- $f(a(\mathtt{zero})) : B(a(\mathtt{zero}))$
  $n : \mathtt{nat}, b : B(a(n)) \vdash f(a(\mathtt{suc}(n))) : B(a(\mathtt{suc}(n)))$
- $\mathtt{dap}_f(p_{-2}) : f(a(\mathtt{zero})) =^{x.B}_{p_{-2}} f(a_{-2})$
  $\mathtt{dap}_f(p_{-1}) : f(a(\mathtt{zero})) =^{x.B}_{p_{-1}} f(a_{-1})$

- $n : \mathtt{nat} \vdash \mathtt{dap}_f(p(n)) : f(a(\mathtt{zero})) =^{x.B}_{p(n)} f(a(n))$

In the following subsections we give examples of definition of higher inductive types which are important for the next parts of this thesis but also for understanding the definition of these types and the concepts of recursion and induction.

3.4.1 The Interval

The interval, which we symbolize as $\mathbb{I}$, is one of the most simple higher inductive types. It is defined as follows:

- $0, 1 : \mathbb{I}$
- $\mathtt{seg} : 0 =_{\mathbb{I}} 1$

The interval is an abstraction of the topological interval $[0, 1]$ subset of $\mathbb{R}$. For the recursion, it is enough to provide the following "data" of the codomain A, two points a_0, a_0 and a path between them.

$$\frac{a_0 : A \quad a_1 : A \quad q : a_0 =_A a_1}{z : \mathbb{I} \vdash \mathtt{rec}[A](a_0, a_1, q)(z) : A}$$

Then the defined function satisfies the following properties:

$$\mathtt{rec}[A](a_0, a_1, q)(0) \equiv a_0$$
$$\mathtt{rec}[A](a_0, a_1, q)(1) \equiv a_1$$
$$\mathtt{ap}_{\mathtt{rec}[A](a_0,a_1,q)}(\mathtt{seg}) = q$$

Similarly, the case of induction, in the dependent context.

$$\frac{z : \mathbb{I} \vdash A : \mathcal{U} \quad a_0 : A(0) \quad a_1 : A(1) \quad q : a_0 =^{z.A}_{\mathtt{seg}} a_1}{z : \mathbb{I} \vdash \mathtt{ind}[z.A](a_0, a_1, q)(z) : A(z)}$$

The above data is sufficient to define a function that satisfies, as before, the following:

$$\mathtt{ind}[z.A](a_0, a_1, q)(0) \equiv a_0$$
$$\mathtt{ind}[z.A](a_0, a_1, q)(1) \equiv a_1$$
$$\mathtt{dap}_{\mathtt{ind}[z.A](a_0,a_1,q)}(\mathtt{seg}) = q$$

It turns out that this space has a very important property. The following type equivalence

$$\mathbb{I} \to A \simeq \Sigma x, y : A.\mathtt{Id}_A(x, y)$$

That is, the type of functions from interval to any type A is equivalent to the type of all paths existing in A with any ends. This, is intuitively obvious if we consider a topological interpretation of paths as continuous functions from the closed interval $[0, 1]$ in the space A.

3.4.2 Circles and Spheres

The Circle At this point we will look in more detail at the type of circle. Circle is defined as we have seen by the presentation of a point (0-cell) and a path starting and ending at this point (1-cell).

- $\mathtt{base} : \mathbb{S}^1$
- $\mathtt{loop} : \mathtt{base} =_{\mathbb{S}^1} \mathtt{base}$

The rules for recursion and induction are as expected. To define a function recursively it is enough to give an element and a loop starting and ending with that element.

$$\frac{a : A \quad \ell : a =_A a}{z : \mathbb{S}^1 \vdash \texttt{rec}[A](a, \ell)(z) : A}$$

The resulting function satisfies the following:

$$\texttt{rec}[A](a, \ell)(\texttt{base}) \equiv a$$
$$\texttt{ap}_{\texttt{rec}[A](a,\ell)}(\texttt{loop}) = \ell$$

While, for the induction

$$\frac{z : \mathbb{S}^1 \vdash A : \mathcal{U} \quad a : A(\texttt{base}) \quad \ell : a =^{z.A}_{\texttt{loop}} a}{z : \mathbb{S}^1 \vdash \texttt{ind}[z.A](a, \ell)(z) : A(z)}$$

with

$$\texttt{ind}[z.A](a, \ell)(\texttt{base}) \equiv a$$
$$\texttt{dap}_{\texttt{ind}[z.A](a,\ell)}(\texttt{loop}) = \ell$$

Like the interval type, the circle type satisfies a similar equivalence. For each type A

$$\mathbb{S}^1 \to A \simeq \Sigma x : A.\texttt{Id}_A(x, x)$$

That is, as before the type of all possible paths with the same beginning and end is equivalent to the type of functions from the circle in space A. As before the similarity to the way they are defined in topology loops in spaces is obvious.

Spheres To make clear how we can define types with a higher dimensional structure than one we give the examples of spheres of higher dimensions than the circle.

To define $\mathbb{S}^2$ we give a point on the sphere ($\texttt{base}$) and a 2-dimensional path constructor, with ends the reflexive path $\texttt{refl}(\texttt{base})$

- $\texttt{base} : \mathbb{S}^2$
- $\texttt{surf} : \texttt{refl}(\texttt{base}) =_{\texttt{base}=\texttt{base}} \texttt{refl}(\texttt{base})$

To define recursively functions from the two-dimensional sphere to any type A we need to define where the function carries the point $\texttt{base}$ and where the function transfers the surface $\texttt{surf}$, that is, define a surface in A which is the value of function $\texttt{ap}^2_f(\texttt{surf})$. The function $\texttt{ap}^2$ is the proof that functions respect two-dimensional paths and is easily defined similar to $\texttt{ap}$ as follows:

$$\Gamma, f : A \to B, x, y : A, p, q : x =_A y, s : p =_{x=y} q \vdash \texttt{ap}^2_f(s) : \texttt{ap}_f(p) =_{f(x)=f(y)} \texttt{ap}_f(q)$$

with

$$\texttt{ap}^2_f(s) \equiv \texttt{J}[p', q', s'.\texttt{Id}_{\texttt{Id}_B(f(x),f(y))}(\texttt{ap}_f(p'), \texttt{ap}_f(q'))](z.\texttt{refl}(\texttt{ap}_f(z)))(s)$$

So, the definition of a function $f : \mathbb{S}^2 \to B$ requires a point $b : B$ and a surface $s : \texttt{refl}(b) = \texttt{refl}(b)$. Then, we have the following rule and equations:

$$\frac{b : B \quad s : \texttt{refl}(b) =_{b=_B b} \texttt{refl}(b)}{\Gamma, z : \mathbb{S}^2 \vdash \texttt{rec}[B](b, s)(z) : B}$$

$$\texttt{rec}[B](b, s)(\texttt{base}) \equiv b$$
$$\texttt{ap}^2_{\texttt{rec}[B](b,s)}(\texttt{surf}) = s$$

Induction is done in a similar way.

3.4.3 Truncations

In this subsection we introduce a way to control the higher dimensional structure of a type using the concept of truncations. The way in which we will present them and we will need them is as higher inductive types.

Homotopy Types One way in which the structure of certain spaces is described in classical homotopy theory is by categorizing them into Homotopy Types. Homotopy Types show us the way in which the elements of a space are connected to each other by homotopies of higher dimension, i.e. higher-dimensional paths. In HoTT these classes of types are called h-types or n-types. There is a cumulative hierarchy of structure complexity in types which starts from -2 and increases indefinitely to more and more complex structures at higher dimensions. The reason why the hierarchy starts at -2 and not 0 is to keep up with similar concepts from other branches of mathematics.

The basis of the hierarchy are -2-types or contractible types. This is a case of subsingleton, i.e. types which are contractible have at least one element x_0 and for every other element there is a path from x_0 to the element. These are fully connected spaces which have at least one element. Under the prism of Props As Types these types express the true propositions of which truth evidences are of no interest to us since they are all equal to each other. In HoTT this is encoded in the following definition:

$$\texttt{isContr}(A) :\equiv \Sigma a : A.\Pi x : A.a =_A x$$

At the second level we have the -1-types or h-Props. In this case, if there are elements of the type then these are all identical with each other. That is,

$$\texttt{isProp}(A) :\equiv \Pi x, y : A.x =_A y$$

This turns out to be equivalent to the definition

$$\texttt{isProp}(A) \simeq \Pi x, y : A.\texttt{isContr}(x =_A y)$$

To these types correspond the propositions of the intuitionistic logic that we saw at the beginning, Props as Types. They are types which are either empty, false propositions, or they are true and that alone is whatever information they provide, all their elements are equal.

The next level in the hierarchy are 0-types or hSets. Types which have no structure higher than 0 dimension and thus one could say that they behave like classical sets. The formal definition states that two paths between two objects of the type, if they exist, are also equal

$$\texttt{isSet}(A) :\equiv \Pi x, y : A.\Pi p, q : x =_A y.p =_{x=_A y} q$$

Again the equivalent definition is as before

$$\texttt{isSet}(a) \simeq \Pi x, y : A.\texttt{isProp}(x =_A y)$$

i.e., the paths between any two objects is a type of exactly one less path structure complexity in the hierarchy.

Continuing inductively we can define n-types with exactly this method: a type is n-type if and only if the type of paths between any two of its elements is (n-1)-type or else

$$\texttt{is-suc}(n)\texttt{-type}(A) :\equiv \Pi x, y : A.\texttt{is-}n\texttt{-type}(x =_A y), n \geq -2$$

Truncations Truncations are ways of controlling the complexity of the structure of a type by cutting the most complex path information available from one dimension and above. In this way a new type is created which is at most n-type, n depending on the dimension of the truncation, and with trivial structure in higher dimensions. One way of defining truncations is as higher inductive types. Here, we present two types of truncation: the propositional truncation or -1-truncation which removes any information-data contained in a type and degenerates it to the level of hProp and the set truncation or 0-truncation, which removes the information about the different paths of all dimensions greater than or equal to 1 by identifying them.

Let's look at the definition of propositional truncation in the context of higher inductive types. The logic of the definition is to create an object in the truncation for each object of the original type and the creation of identity paths between all new objects. The truncated new type is denoted by $\|A\|$ and we have:

- $|-| : A \to \|A\|$
- $\texttt{squash} : \Pi x, y : \|A\| . x =_A y$

The idea is to substitute any information about the structure of A to information about the "mere inhabitance" of the type, i.e. it removes the proof relevance content of the type and keeps whether it contains any elements or not, ignoring what they are and what properties they have. The recursion and induction principles are relatively simple. In the case of recursion, to define a function with a truncated type as its domain, the codomain must be an hProp.

$$\frac{\Gamma \vdash P : \texttt{isProp}(B) \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \texttt{rec}[B](x.N, P) : \|A\| \to B}$$

Then, the defined function satisfies the identities

$$\texttt{rec}[B](x.N, P)(|M|) \equiv [M/x]N : B$$
$$\texttt{ap}_{\texttt{rec}[B](x.N,P)}(\texttt{squash}(|M_1|, |M_2|)) = P([M_1/x]N)([M_2/x]N)$$

Similarly, for the induction and definition of a dependent function $f : \Pi x : \|A\| .B$ there must exist for each element $x : A$ an element $b : B(|x|)$ and there must be dependent paths over over the paths $\texttt{squash}(x, y)$.

$$\frac{\Gamma, z : \|A\| \vdash B : \mathcal{U} \quad \Gamma, x : A \vdash N : B(|x|) \quad \Gamma, x, y : \|A\|, u : B(x), v : B(y) \vdash q : u =^{z.B}_{\texttt{squash}(x,y)} v}{\Gamma \vdash \texttt{ind}[z.B](x.N, x, y, u, v.q) : \Pi x : \|A\| .B}$$

The corresponding identities are satisfied.

The set truncation of a type is defined in exactly the same way as the propositional truncation except that it does not identify all elements of a type, 0-cells, but all paths of the type, 1-cells. The type that is produced is denoted by $\|A\|_0$.

- $|-|_0 : A \to \|A\|_0$
- $\texttt{squash}_0 : \Pi x, y : \|A\|_0 . \Pi p, q : x =_{\|A\|_0} y . p =_{x =_{\|A\|_0} y} q$

To define a recursive function $f : \|A\|_0 \to B$ the type B must be hSet.

$$\frac{\Gamma \vdash P : \texttt{isSet}(B) \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \texttt{rec}[B](x.N, P) : \|A\|_0 \to B}$$

Then we have the following identities:

$$\texttt{rec}[B](x.N, P)(|M|_0) \equiv [M/x]N : B$$
$$\texttt{ap}_{\texttt{rec}[B](x.N,P)}(\texttt{squash}_0(|M_1|_0, |M_2|_0)) = P([M_1/x]N)([M_2/x]N)$$

Finally, for the induction we follow the same logic as in the case of propositional truncation

$$\frac{\begin{array}{c}\Gamma, z : \|A\|_0 \vdash B : \mathcal{U} \\ \Gamma, x : A \vdash N : B(|x|_0) \\ \Gamma, x, y : \|A\|_0, p, q : x =_{\|A\|_0} y, u : B(x), v : B(y), r : u =^{z.B}_{p} v, s : u =^{z.B}_{q} v \vdash sq : r =^{w.u =^{z.B}_{w} v}_{\texttt{squash}_0(x,y,p,q)} s\end{array}}{\Gamma \vdash \texttt{ind}[z.B](x.N, x, y, p, q, u, v, r, s.sq) : \Pi x : \|A\|_0 .B}$$

The function again satisfies the obvious equations.

Chapter 4

Cubical Type Theory - Cubical Agda

As we saw in the formal presentation of Homotopy Type Theory, for each type A and two objects $x, y : A$, possibly the same, we can create the type $\texttt{Id}_A(x, y)$ of all proofs of identification of objects x, y. This approach has some characteristics. First the definition of the type $\texttt{Id}_A$ is generic in terms of the type A. This means that the rules of the type, do not differ accordingly to the type A. Type $\texttt{Id}_A$, appears to be defined as the type of free constructions obtained by applying the `J` operator on the `refl` object. Also, the `J` operator can produce a wide variety of proofs, e.g. `tr`, `ap`, ..., by virtue of the definition of $\texttt{Id}_A$ and itself, when given any type of identity proof while giving a way of computation only for the case of `refl`. So, we can do the following β-reductions:

$$(\texttt{refl})^{-1} \equiv \texttt{refl}\ \texttt{refl} \cdot \texttt{refl} \equiv \texttt{refl}\ \texttt{tr}[x.C](\texttt{refl}) \equiv \texttt{id}\ \texttt{ap}_f(\texttt{refl}) \equiv \texttt{refl} \ldots.$$

which are all based on the `J` β-reduction rule.

But in order to enrich our theory with the Univalence Axiom and Higher Inductive Types, we had to add a variety of new proofs of equality like `loop`, `seg`,... in $\texttt{Id}_A$ after its definition. This creates the following problem: ”`J` has no idea what to do with these arguments”. More formally, there is no computation rule for other objects of the type $\texttt{Id}_A$ except `refl`. The result is the creation of expressions with correct types (that type check) but not normalizable to values. So we have *stuck terms*.

Cubical Type Theory is an extension of Homotopy Type Theory which aims to fully recover the computational content of the theory without reducing its expressive power. The way this is done is through an attempt to transfer the concept of path from the level of type objects to the level of judgments. To do this, it creates a new interpretation of paths as functions from the unit interval $\mathbb{I}$ to the type A. This interval has nothing to do with the interval we defined as a Higher Inductive Type. In the case of Cubical Type Theory $\mathbb{I}$ is not a type but an extension of context and has its own algebraic structure. So, in Cubical Type Theory the paths from $x : A$ to $y : A$ are ”continuous” functions from $\mathbb{I}$ to A with value x for $\texttt{i0} : \mathbb{I}$ and y for $\texttt{i1} : \mathbb{I}$, just like in topology. This means that if we have $\Gamma, i : \mathbb{I} \vdash t : A$, i.e. for each point in the interval we can find an element of type A, then we can create a path in the type A with the values of $t : A$ at the ends of the interval. In this way, the concept of the path is transferred to a more general frame, at the level of contexts, and then internalized by the creation of the path type, `Path`. The very same process is done in the case of entailment and logical implication. The entailment ”$\Gamma, x : A \vdash y : B$” expresses a mode of dependence between the judgments and this notion is internalized in the system by introducing the type of implication ”$\Gamma \vdash \lambda x.y : A \to B$”. Cubical Type Theory transfers the treatment of paths and equality to a more primitive level so that their definition will not be generic with respect to the type A but that each type is responsible for the correct definition of its paths. Then, follows the internalization of the concept of path by the creation of the type `Path`. In this context, type $\texttt{Id}_A$ appears as an inductive type, generic to A. Thus, there is a distinction between the judgmental framework, the paths of `Path` type and the identity type $\texttt{Id}_A$ which will be adopted from this point on.

There are several variants of Cubical Type Theory. Here, we will present the one used to develop Cubical Agda which is the proof assistant used to create the library of this thesis. We will first give the new extensions of the syntax and then we will show how our framework acquires computational content. It is worth noting here that this work is not mandatory for someone who is not interested in the computational side of Type Theory. One can create proofs and objects as in previous chapters but which will not "run" as programs. However, if we are interested in the computational content of the theory, Cubical Type Theory offers a solution to the problem.

4.1 Path Types

To be able to describe the new interpretation of the paths we extend the syntax of our language. We assume that we have a discrete infinite set of directions , $j, k, \dots$. We define $\mathbb{I}$ to be the freely generated De Morgan algebra with generators this set of directions. This means that the elements of $\mathbb{I}$ can be derived from the following grammar

$$r, s ::= \mathtt{i0} | \mathtt{i1} | i | \sim r | r \vee s | r \wedge s$$

and satisfy the following equations:

$$\begin{gathered} \sim \mathtt{i0} = \mathtt{i1} \\ \sim \mathtt{i1} = \mathtt{i0} \\ \sim (r \vee s) = (\sim r) \wedge (\sim s) \\ \sim (r \wedge s) = (\sim r) \vee (\sim s) \end{gathered}$$

We extend the concept of context to include expressions of the form $\Gamma, i : \mathbb{I}$, and with the rule

$$\frac{\Gamma \vdash}{\Gamma, i : \mathbb{I} \vdash}$$

In this way we can express the concept of an object defined using directional variables. Thus, when we have a judgment of the form $\Gamma, i : \mathbb{I} \vdash M : A$ the way in which we imagine the result is a continuous line of objects in A starting at $M(\mathtt{i0}) : A$ and ending at $M(\mathtt{i1}) : A$, i.e. a 1-cube. When there is a dependence of two variables $\Gamma, i : \mathbb{I}, j : \mathbb{I} \vdash M : A$ then a square is generated, i.e. a 2-cube, as in the figure

$$\begin{tikzcd}
M(\mathtt{i0}/i)(\mathtt{i1}/j) \arrow[r, dash, "M(\mathtt{i1}/j)"] \arrow[d, dash, "M(\mathtt{i0}/i)"] & M(\mathtt{i1}/i)(\mathtt{i1}/j) \arrow[d, dash, "M(\mathtt{i1}/i)"] \\
M(\mathtt{i0}/i)(\mathtt{i0}/j) \arrow[r, dash, "M(\mathtt{i0}/j)"] & M(\mathtt{i1}/i)(\mathtt{i0}/j)
\end{tikzcd}$$

In the case where we have 3 directions then the result is a cube, while when we have n dimensions the result is an n-cube. This is why it is called Cubical Type Theory.

In this way we managed to describe, using variables $i, j, k, \dots : \mathbb{I}$ for the various dimensions, points paths, squares, cubes, hypercubes, etc. in the level of judgments of our theory. Now we define a type that internalizes this concept in the system of types. This is the type $\mathtt{Path}\ A\ t_0\ t_1$, which for a given type A and two objects t_0, t_1 of this type, categorizes all paths in A starting at t_0 and ending at t_1.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash t_0 : A \quad \Gamma \vdash t_1 : A}{\Gamma \vdash \mathtt{Path}\ A\ t_0\ t_1}$$

To create an object of this type, a path from t_0 to t_1 we need to have for each $i : \mathbb{I}$ an object $t : A$ such that $t(\mathtt{i0}) \equiv t_0$ and $t(\mathtt{i1}) \equiv t_1$. The similarity with the topological definition of paths is obvious. To denote the new path we adopt the Cubical Agda way and we use λ-abstraction.

$$\frac{\Gamma \vdash A \quad \Gamma, i : \mathbb{I} \vdash \lambda t : A}{\Gamma \vdash \lambda i.t : \mathtt{Path}\ A\ t(\mathtt{i0})\ t(\mathtt{i1})}$$

Of course, since we now have a way of expressing various points of $\mathbb{I}$ it makes sense to be able to manipulate the points of the path to which this point corresponds. This is similar to the concept of function application and justifies the choice of the operator λ. The result will of course belong to the type A. This is what following rule tells us.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash p : \texttt{Path}\ A\ t_0\ t_1 \quad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash p(r) : A}$$

Let's see how we can calculate points on a path for each point of $\mathbb{I}$. If we create a path from a judgment of the form $\Gamma, i : \mathbb{I} \vdash t : A$ and apply to it an element $r : \mathbb{I}$ we get the corresponding $[r/i]t$. That is,

$$\frac{\Gamma \vdash A \quad \Gamma, i : \mathbb{I} \vdash t : A \quad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash \lambda i.t(r) \equiv [r/i]t : A}$$

Also, it is necessary to have judgmental equality between the initial end of the path and its value for $i = \texttt{i0}$ and respectively for final end and its value for $i = \texttt{i1}$.

$$\frac{\Gamma \vdash p : \texttt{Path}\ A\ t_0\ t_1}{\Gamma \vdash p(\texttt{i0}) \equiv t_0} \qquad \frac{\Gamma \vdash p : \texttt{Path}\ A\ t_0\ t_1}{\Gamma \vdash p(\texttt{i1}) \equiv t_1}$$

Finally, if we have two paths that agree at each point then the two paths should be equal.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash p, q : \texttt{Path}\ A\ t_0\ t_1 \quad \Gamma, i : \mathbb{I} \vdash p(i) \equiv q(i) : A}{\Gamma \vdash p \equiv q : \texttt{Path}\ A\ t_0\ t_1}$$

With this more direct way of handling paths we can very easily do a lot of things that either weren't so direct or were impossible in the previous framework. For example `refl` is defined naturally as a constant path

$$\frac{\Gamma \vdash A \quad \Gamma \vdash a : A}{\Gamma \vdash \texttt{refl} :\equiv \lambda i.a : \texttt{Path}\ A\ a\ a}$$

for which it is of course true that $\Gamma, r : \mathbb{I} \vdash \lambda i.a(r) \equiv a : A$. Moreover, to define the inverse path of a path p, very simply we apply the operator $\sim i$ to the argument of the path.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash a, b : A \quad p : \texttt{Path}\ A\ a\ b}{\Gamma \vdash \lambda i.p(\sim i) : \texttt{Path}\ A\ b\ a}$$

Furthermore, paths between two points in the domain of a function get mapped to paths in the codomain in a very simple way:

$$\frac{\Gamma \vdash A, B \quad \Gamma \vdash f : A \to B \quad \Gamma \vdash a, b : A \quad \Gamma \vdash p : \texttt{Path}\ A\ a\ b}{\Gamma \vdash \lambda i.f(p(i)) : \texttt{Path}\ B\ f(a)\ f(b)}$$

Finally, the most impressive is the proof of the theorem of function extensionality already with these simple first extensions. Let A, B be types, $f, g : A \to B$ two functions from A to B and $H : \Pi a : A.\texttt{Path}\ B\ f(a)\ g(a)$ a homotopy between f and g. Then we can make the following steps:

$$\begin{gathered}
\Gamma \vdash H : \Pi a : A.\texttt{Path}\ B\ f(a)\ g(a) \\
\Gamma, x : A \vdash H(x) : \texttt{Path}\ B\ f(x)\ g(x) \\
\Gamma, x : A, r : A \vdash H(x)(r) : B \\
\Gamma, r : A, x : A \vdash H(x)(r) : B \\
\Gamma, r : A \vdash \lambda a.H(a)(r) : A \to B \\
\Gamma \vdash \lambda i.\lambda a.H(a)(i) : \texttt{Path}\ (A \to B)\ f\ g
\end{gathered}$$

To check that the ends of the path we constructed are indeed the ones we wrote down, we simply calculate the value for $i \equiv \texttt{i0}$

$$(\lambda i.\lambda a.H(a)(r))(\texttt{i0}) \equiv \lambda a.H(a)(\texttt{i0}) \equiv \lambda a.f(a) \equiv f$$

and similar for the other end. What made this proof possible is the interchange of the terms $x : A, r : A$ in the context, which would not be possible before the expression of paths to this level.

4.2 Composition and Coercion

We have not yet mentioned a very important operation between paths, paths' concatenation, and we have not discussed about paths in the type of universe, that is, paths between types. In order to be able to support the operations of path concatenation and transport between equal types, Cubical Type Theory requires all types to provide a way to perform these operations, *in their definition*. In other words, to recover the computational content of paths such as `ua`, `loop`, etc., types themselves, must provide the way to compute their operations. It turns out that in order to have the paths that we want, that behave in a normal way, i.e. closed under our operations, we need them to satisfy some conditions called Kan conditions. So, any type to be properly defined in Cubical Type Theory must provide operators that witness the satisfaction of the Kan conditions in addition to the various rules we have seen, formation rules, introduction, elimination, β and η reduction.

4.2.1 Composition

More specifically in Cubical Agda we will introduce two operators. The first one is called *homogeneous composition*. Homogeneous composition begins with a cube that is missing a face and gives the composition of the existing faces to create the face that is missing so that the cube is complete. For two dimensions i.e. we have the following picture

$$\begin{tikzcd}
\bullet \arrow[r, dashed] & \bullet \\
\bullet \arrow[u, "\texttt{face}"] \arrow[r] & \bullet \arrow[u, "\texttt{face}"]
\end{tikzcd}$$

This idea extends to any dimension. It is called homogeneous because during our transport from the lower face to the to the newly created one we stay inside the same type A. In Cubical Agda this operator is denoted by `hcomp`. Unfortunately `hcomp` would also be called the operator of heterogeneous composition, therefore we have to be careful with notation. The `hcomp` operator should always return the upper face even when the cube is missing some of the side faces like in these cases:

$$\begin{tikzcd}
\bullet \arrow[r, dashed] & \bullet & & \bullet \arrow[r, dashed] & \bullet \\
\bullet \arrow[u, dashed, "\texttt{face}"] \arrow[r] & \bullet \arrow[u, "\texttt{face}"] & & \bullet \arrow[u, dashed, "\texttt{face}"] \arrow[r] & \bullet \arrow[u, dashed, "\texttt{face}"]
\end{tikzcd}$$

and of course in every dimension.

4.2.2 Coercion

The second operator that we will define to complete Kan conditions is the coercion operator. The concept comes from the ideas we saw in the previous chapter. According to those, when we have an identity p between two types $A, B : \mathcal{U}$ in the universe then we could construct an object-transferring function from one type to the other, in the following way:

$$f :\equiv \texttt{tr}[x.x](p) : A \to B$$

This is a very reasonable situation that we want to maintain in Cubical Type Theory. After all, since the two types are equal their elements should be matched. As this operation is similar to type coercion in programming languages we adopt the same terminology.

In the case of Cubical Type Theory we can have paths between types just like between objects in all dimensions. So, coercion is the transfer of objects from one of the faces to exactly the opposite one. If for example we have the following equality square between types the Coercion operator allows us to transfer objects from the lower face to the upper face.

$$\begin{tikzcd}
A(i/\texttt{i0})(j/\texttt{i1}) \arrow[r, dashed] & A(i/\texttt{i1})(j/\texttt{i1}) \\
A(i/\texttt{i0})(j/\texttt{i0}) \arrow[r] \arrow[u] & A(i/\texttt{i1})(j/\texttt{i0}) \arrow[u]
\end{tikzcd}$$

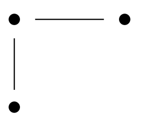
It is important to note the difference between coercion and homogeneous composition. In homogeneous composition we do not leave the original type A while when using the coercion operator we move to another type equal to the original one.

4.2.3 Partial Elements

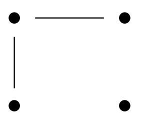
To be able to describe the function of these operators we need to add to our syntax a way of describing n-cubes which are missing one or more faces. This is done using dimensional equations. For example, in the three-dimensional cube equation $(i = \texttt{i0})$ describes all elements of the cube free with respect to dimensions j, k but which have $\texttt{i0}$ as the value of the dimension variable i. So, this is the face which is in the dimension $(i = \texttt{i0})$. In order to be able to describe subpolyhedra of a larger variety we allow two operations between these equations, $\wedge$ and $\vee$, that satisfy the laws of commutativity and $(i = \texttt{i0}) \wedge (i = \texttt{i1}) = 0_{\mathbb{F}}$, where $0_{\mathbb{F}}$ and $1_{\mathbb{F}}$ denote the equations that no element satisfies and the equation that all elements of the cube satisfy. This creates the algebraic structure of a distributive lattice which gives us the complete description we want for the different parts of the cube. Some examples:

$(i = \texttt{i0}) \wedge (i = \texttt{i1})$: two points without any link

$(i = \texttt{i0}) \vee (j = \texttt{i1})$: the sum of two faces



$(i = \texttt{i0}) \vee (j = \texttt{i1}) \vee ((i = \texttt{i1}) \wedge (j = \texttt{i1}))$: the sum of two faces and a vertex not belonging to them



Furthermore, it is possible to introduce these constraints at the level of the context. If we have an object ϕ of the lattice of partial elements then it is possible to introduce it into the context's syntax in the following way

$$\Gamma, \phi \vdash$$

This gives us greater expressiveness as we can prove judgments only in the subpolyhedra defined by the constraint ϕ. More precisely, if we have an object $M : A$ and write $\phi \vdash M \equiv N : A$ then this means that the two objects are equal only at the points of the cube satisfying constraint ϕ and not necessarily everywhere. This is also written as

$$M : A \text{ and } \phi \vdash M \equiv N : A \implies M : A[\phi \mapsto N]$$

For more details on the definition of the above we refer to to [Cohe16].

4.2.4 Syntax and Use

We now turn to the syntax and use of the two operators. For the case of homogeneous composition we need the following as we have seen:

- $A : \mathcal{U}$, the type in which the composition will take place
- $M : A$, the "bottom" face of the cube which we must always have. M in our case is n-dimensional if it depends on n dimensional variables of the context
- $[\phi_1 \mapsto u_1, \phi_2 \mapsto u_2, ...]$, a finite collection of side faces of our hypercube in a cluster that is called a *system*. Here each ϕ_i corresponds to a dimensional equation and each u_i corresponds to an object of A which is judgmentally equal to the hypercube at the points satisfying the equation ϕ_i. At the intersections of the equations ϕ_{i_1}, ϕ_{i_2} u_{i_1}, u_{i_2} must also be judgmentally equal, as well as at the intersections of the side faces with M.

If we have all of the above, then the homogeneous composition operator is defined in the following syntax

$$\texttt{hcomp}^k A[\phi_1 \mapsto u_1, \phi_2 \mapsto u_2, ...]M : A$$

where k is the dimension along which the composition is performed.

For the case of the Coercion operator we need a path between two types in $\mathcal{U}$, i.e. a type with a dependence on some dimension variable $\Gamma, j : \mathbb{I} \vdash A$ or in the form of a path in the universe $A : \texttt{Path}\ universe\ A(\texttt{i0})\ A(\texttt{i1})$. Our goal is to create a function of type $A(\texttt{i0}) \to A(\texttt{i1})$. The implementation which is chosen in Cubical Agda generalizes the transfer operator by giving the option of setting restrictions on how the transfer is performed in dimensions other than $i : \mathbb{I}$. More precisely for the points satisfying the constraints the resulting function should be be the identity, and hence type A constant at these points. The way in which the restrictions are represented is not with an element of the lattice of faces of the cube but with an element of $\mathbb{I}$. It turns out that the points satisfying a set of dimension equations can be represented by a point of $\mathbb{I}$ wherever it equals $\texttt{i1}$. For example, the restrictions $(i = \texttt{i0}) \vee (j = \texttt{i1}) \vee ((i = \texttt{i1}) \wedge (j = \texttt{i1}))$ seen above are equivalently described by the points for which it holds that $(\sim i) \vee j \vee (i \wedge j) : \mathbb{I}$ equals $\texttt{i1}$. Then if:

- $\Gamma, j : \mathbb{I} \vdash A$ is the path on which we are interested in transferring objects from $A(\texttt{i0})$ to $A(\texttt{i1})$ along the dimension j.
- $r : \mathbb{I}$ is a point in the interval that describes the points where the constraints will apply, i.e. where "$r = \texttt{i1}$"
- $M : A(\texttt{i0})$ a point of the type at the beginning of the path

we have the transfer operator

$$\texttt{transp}^j A\ r\ M : A(\texttt{i1})$$

To understand a little better the concept of restrictions $r : \mathbb{I}$ let's see some examples.

- If $r \equiv \texttt{i0} : \mathbb{I}$ then the restriction $\texttt{i0} = \texttt{i1}$ is not satisfied for any point on the cube and therefore there is no effective constraint. So, we can define

  $$\texttt{transport} :\equiv \texttt{transp}^i\ A\ \texttt{i0} : [\texttt{i0}/i]A \to [\texttt{i1}/i]A$$

  the function of transferring objects between two equal types.

- If $r \equiv \texttt{i1} : \mathbb{I}$ then the restriction $\texttt{i1} = \texttt{i1}$ is satisfied for any point of the cube and so the resulting function should be the identity everywhere and the path of A should be constant

  $$\texttt{transp}^i\ A\ \texttt{i1} \equiv \texttt{id} : [\texttt{i0}/i]A \to [\texttt{i0}/i]A$$

- If we have the following square

$$\begin{tikzcd}
A(i/\texttt{i0})(j/\texttt{i1}) \arrow[r] & A(i/\texttt{i1})(j/\texttt{i1}) \\
A(i/\texttt{i0})(j/\texttt{i0}) \arrow[r] \arrow[u] & A(i/\texttt{i1})(j/\texttt{i0}) \arrow[u]
\end{tikzcd}$$

  and we want to transfer along dimension j while keeping let's say the elements unchanged when transferred in the left face then the appropriate constraint is "$i = \texttt{i0} \Leftrightarrow \sim i = \texttt{i1}$". This imposes a restriction on the square A so that the left-hand side is the constant path and the types $A(i/\texttt{i0})(j/\texttt{i0})$ and $A(i/\texttt{i0})(j/\texttt{i1})$ be judgmentally equal.

$$\Gamma, i : \mathbb{I} \vdash \texttt{transp}^j\ A\ (\sim i) : [\texttt{i0}/j]A \to [\texttt{i1}/j]A$$

Now that we have seen the syntax of these operators, we need to see how we should use them. As we said in the introduction of the chapter, each type should be responsible for the way in which these operations are performed. In other words, we should review the definitions for all the types we have presented adding the requested functions.

4.2.5 Case Study - Dependent Functions

As an example we will show how this is done for the type of Dependent Functions. Let us consider a cube in some function type $\Pi x : A.B$, $M : \Pi x : A.B$ και $[\phi_1 \mapsto u_1, \phi_2 \mapsto u_2, ...]$ a system of faces, that satisfy the above restrictions. Then

$$\texttt{hcomp}^i(\Pi x : A.B)[\phi_1 \mapsto u_1, \phi_2 \mapsto u_2, ...]M :\equiv \lambda x.\texttt{hcomp}^i B[\phi_1 \mapsto u_1(x), \phi_2 \mapsto u_2(x), ...]M(x) : \Pi x : A.B$$

That is, we define the operator of the composition of paths of functions taking into account only the input-output behavior of the faces- functions. The result should of course be a function within the type $\Pi x : A.B$, since during homogeneous composition the type does not change.

For Coercion again we focus on the behavior of the function as an input-output system. The transferred function takes arguments to the type $[\texttt{i1}/i]A$ but the function we have as input belongs to the beginning type $M : [\texttt{i0}/i](\Pi x : A.B)$. So, to make the computation we transfer the arguments to the type $u : [\texttt{i0}/i]A$, do the computation with M and transfer the result $M(u)$ back to $[\texttt{i1}/i]B(u)$.

$$\texttt{transp}^i\ (\Pi x : A.B)\ r :\equiv \lambda x.\texttt{transp}^i\ [u/x]B\ r\ M(u) : [\texttt{i1}/i](\Pi x : A.B), \text{where } u :\equiv \texttt{transp}^{\sim i}\ A\ r\ x : [\texttt{i0}/i]A$$

Agda defines all these operators of types to be fully computable for each path instance. Internally it pattern matches in the possible type and applies these new operations in an appropriate way in each case. Here, we will not show how they are implemented for other types except for the universe case. We refer to [Cohe16] and [Vezz19] for more details on the implementation.

4.2.6 Universes in Cubical Type Theory

In order to be able to correctly define the operations of homogeneous composition and coercion in the case of the universe, with the ultimate goal of obtaining an implementation of the Univalence Axiom with computational meaning, we introduce a new kind of types, the Glue Types. Glue Types internalize the essence of the Univalence Axiom by allowing us to construct cubes which instead of paths in certain faces have equivalences. The idea is analogous to the way `hcomp` works in which we compose paths but in the case of glue types we compose equivalences of types.

Glue Types The Glue type is also a type, which means that we have to define it just like all the others. Forming Glue type requires that we have a type $A : \mathcal{U}$ which is equivalent to some type $\phi \vdash T : \mathcal{U}$ at some points of the hypercube with some equivalences $\phi \vdash E : T \simeq A$, as in the figure for instance

$$\begin{tikzcd} T \arrow[d, "E"] & T \arrow[d, "E"] \\ A(i/\mathtt{i0}) \arrow[r] & A(i/\mathtt{i1}) \end{tikzcd}$$

Then we can define the Glue type using the following formation rule

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma, \phi \vdash T : \mathcal{U} \quad \Gamma, \phi \vdash E : T \simeq A}{\Gamma \vdash \mathtt{Glue}[\phi \mapsto (T, E)]A : \mathcal{U}[\phi \mapsto T]} \; F - \mathtt{Glue}$$

We note that at the points satisfying ϕ the Glue type is judgmentally equal to T.

Then, the Glue type has a rule for introducing its elements with the constructor `glue`, which for each partial element $\phi \vdash t : T$ of T and any $a : A$ equal to $t : T$ under the equivalence at the points of the constraint ϕ gives an element of the type $\mathtt{Glue}[\phi \mapsto (T, E)]A$.

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash a : A[\phi \mapsto \mathtt{fst}(E)(t)]}{\Gamma \vdash \mathtt{glue}[\phi \mapsto t]a : (\mathtt{Glue}[\phi \mapsto (T, E)]A)[\phi \mapsto t]} \; I - \mathtt{Glue}$$

Again, we see that the element $\mathtt{glue}[\phi \mapsto t]a$ is judgmentally equal to t wherever ϕ holds which does not cause type error since at these points the types $\mathtt{Glue}[\phi \mapsto (T, E)]$ and T are identical.

The elimination rule defines a function `unglue` which sends objects $u : \mathtt{Glue}[\phi \mapsto (T, E)]$ back to A making sure that the elements of T, i.e. those subject to the constraints of ϕ, are sent to their equivalents in A.

$$\frac{\Gamma \vdash u : \mathtt{Glue}[\phi \mapsto (T, E)]}{\Gamma \vdash \mathtt{unglue}[\phi \mapsto E]u : A[\phi \mapsto \mathtt{fst}(E)(u)]} \; E - \mathtt{Glue}$$

These two functions satisfy the following β-reduction rules which are the most obvious ones

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash a : A[\phi \mapsto \mathtt{fst}(E)(t)]}{\Gamma \vdash \mathtt{unglue}(\mathtt{glue}[\phi \mapsto t]a) \equiv a : A[\phi \mapsto \mathtt{fst}(E)(t)]} \; \beta - \mathtt{Glue}$$

$$\frac{\Gamma \vdash u : \mathtt{Glue}[\phi \mapsto (T, E)]}{\Gamma \vdash \mathtt{glue}(\mathtt{unglue}[\phi \mapsto E]u) \equiv u : \mathtt{Glue}[\phi \mapsto (T, E)]} \; \beta - \mathtt{Glue}$$

The last part of the definition of the Glue type is the definition of homogeneous composition and coercion operators. These two points are the most complex and technical ones of Cubical Type Theory. Also, these parts are not necessary for the definition of the Univalence Axiom which is ultimately our goal but also for the intuitive understanding of Glue types. For these reasons we omit them and refer to [Cohe16] for their definition.

Computations in The Universe Let's see how we can use Glue types to define the homogeneous composition and coercion operators in the universe $\mathcal{U}$ and construct the Univalence Axiom, which will now be a theorem in Cubical Type Theory with fully computational behavior.

For the case of homogeneous composition we assume that we have a type $A : \mathcal{U}$, a system of faces $[\phi_1 \mapsto u_1, \phi_2 \mapsto u_2, ...]$ and that we want to compose along dimension i. Then we can convert these faces from paths to equivalences in the following way

$$E_n :\equiv \mathtt{transport}^i \; [\sim i/i]u_n$$

reversing the direction of transport since the direction of equivalences in Glue definition is from T to A. Then, we set

$$\mathtt{hcomp}^i \; \mathcal{U} \; [\phi_1 \mapsto u_1, \phi_2 \mapsto u_2, ...] \; A :\equiv \mathtt{Glue}[\phi_1 \mapsto (u_1(\mathtt{i1}), E_1), \phi_2 \mapsto (u_2(\mathtt{i1}), E_2), ...]A$$

We give a picture as a visual example

$$\begin{tikzcd} u_1(\texttt{i1}) & u_2(\texttt{i1}) & \text{convert to} & u_1(\texttt{i1}) \arrow[r, "Glue"] \arrow[d, "E_1"] & u_2(\texttt{i1}) \arrow[d, "E_1"] \\ A(j/\texttt{i0}) \arrow[u, "u_1"] \arrow[r] & A(j/\texttt{i1}) \arrow[u, "u_2"'] & & A(j/\texttt{i0}) \arrow[r] & A(j/\texttt{i1}) \end{tikzcd}$$

Coercion is done much easily without using Glue types, very simply by returning the same type. That is, we don't do any change in the type.

$$\texttt{transp}^i\, \mathcal{U}\ r\ A :\equiv A$$

We are now ready to prove the Univalence Axiom. We assume that we have two types $A, B : \mathcal{U}$ and a proof of of their equivalence $e : A \simeq B$. Then, we can construct the following partial family of types in the universe with equivalences instead of paths as in the definition of the Glue type, where $\texttt{idEquiv}(B)$ is is the reflexive equivalence

$$\begin{tikzcd} A \arrow[d, "e"] & A \arrow[r, "Glue"] \arrow[d, "e"] & B \arrow[d, "\texttt{idEquiv}(B)"] \\ B & B \arrow[r] & B \end{tikzcd}$$

We finally define

$$\texttt{ua}(e)(i) :\equiv \texttt{Glue}[(i = \texttt{i0}) \mapsto (e, A), (i = \texttt{i1}) \mapsto (\texttt{idEquiv}(B), B)]B$$

and we have that

$$\texttt{transp}^i\ \texttt{ua}(e)(i)\ \texttt{i1}\ x \equiv \texttt{id}(\texttt{transp}^i\ B\ \texttt{i1}\ (\texttt{fst}(e)(x)))$$

That is, the transfer over the type $\texttt{ua}(e)$ is a transfer of an equivalent point over the type $\Gamma, i : \mathbb{I} \vdash B$. If the type B is constant with respect to $i : \mathbb{I}$ then we have the following equality

$$\texttt{transport}\ \texttt{ua}(e)\ x \equiv \texttt{fst}(e)(x)$$

Chapter 5

The Fundamental Group of a Bouquet of Circles

In this chapter, we will present the experimental part of this thesis. The aim is to prove a known result from algebraic topology in Cubical Agda. The objective theorem is the following:

The fundamental group of a Bouquet of circles indexed by a type A
is equal to the free group generated by the elements of A

To explain this better and to describe the most important parts of the library that has been developed we will analyze it in the following points:

- Freely generated group
- Freely generated groupoid
- Fundamental group
- Proof

These will quote the code from the library and will respect the naming conventions used there. Where necessary it will be presented the whole form of each individual proof. The library is freely accessible here: https://github.com/gmagaf/Bouquet-HoTT-Cubical-Agda.

5.1 FreeGroup of A

The free group generated by the elements of A is an algebraic object. Given a set S we can always construct words based on the elements of this set as letters. Thus, by the operation of concatenating elements $\mathtt{m}$ and the empty word that is symbolized with $\mathtt{e}$ we can construct a monoid (the freely generated monoid from the elements of S). We only need a mapping that sends the elements of S in the monoid, in words with only one letter: the element itself, which we call η. In such way we have the following construction:

- $\eta : S \to \mathbb{G}$ The embedding mapping from S into $\mathbb{G}$
- $\mathtt{m} : \mathbb{G} \to \mathbb{G} \to \mathbb{G}$ The operation of concatenating words
- $\mathtt{e} \in \mathbb{G}$ The identity element of the $\mathbb{G}$

These of course, for $\mathbb{G}$ to be truly monoid, must satisfy the associativity law and the laws of identity element.

$$\forall x, y, z \in \mathbb{G}.\mathtt{m}(x)(\mathtt{m}(y)(z)) = \mathtt{m}(\mathtt{m}(x)(y))(z)$$
$$\forall x \in \mathbb{G}.x = \mathtt{m}(x)(\mathtt{e})$$
$$\forall x \in \mathbb{G}.x = \mathtt{m}(\mathtt{e})(x)$$

Extending the idea from monoids to groups, we need for each element its inverse. That is, one more function inv : $\mathbb{G} \to \mathbb{G}$ for which the laws of inverses apply:

$$\forall x \in \mathbb{G}.\texttt{m}(x)(\texttt{inv}(x)) = \texttt{e}$$
$$\forall x \in \mathbb{G}.\texttt{m}(\texttt{inv}(x))(x) = \texttt{e}$$

The way we transport the above construction to the context of type theory is through a higher inductive type. Each one of the functions we described (η, m, e, inv) is also an object constructor in the new type. The equalities that we required these functions to satisfy form the path constructors of dimension one of the inductive definition. Finally, for the result to be a group and especially a set we want a 2-dimension constructor that makes the final type h-Set. In Cubical Agda the definition is the following. Note: the type of the path between two points in the same type is also symbolized as $x \equiv y$ in Cubical Agda.

```
data FreeGroup (A : Type ℓ) : Type ℓ where
  η     : A → FreeGroup A
  m     : FreeGroup A → FreeGroup A → FreeGroup A
  e     : FreeGroup A
  inv   : FreeGroup A → FreeGroup A
  assoc : ∀ x y z → m x (m y z) ≡ m (m x y) z
  idr   : ∀ x → x ≡ m x e
  idl   : ∀ x → x ≡  m e x
  invr  : ∀ x → m x (inv x) ≡ e
  invl  : ∀ x → m (inv x) x ≡ e
  trunc : ∀ x y → ∀ (p q : x ≡ y) → p ≡ q
```

It's truly impressive the way that we defined an object by describing the properties that we want it to satisfy. This way of object definition is very common in Category theory, through the description of a universal property for the defined object.

It turns out that for the definition of a homomorphism with type freeGroup $A \to G$ it is sufficient to determine the behavior of the function on the elements of A, or more correctly on the elements of the form $\eta(a)$: freeGroup A for $a : A$. This mapping is an equivalence of types with the inverse function being the constraint of the homomorphism to the elements $\eta(a)$: freeGroup A. From Univalence Axiom we get that the two types are equal:

$$(A \to G) = \texttt{GroupHom}\ (\texttt{freeGroup}\ A)\ G$$

Finally, for the proof of a property (h-Prop) for every element of the freely generated group it is sufficient to have the following 4 proofs:

- Proof of the property for each element $\eta(a)$, when $a : A$
- Proof of the property for the element $\texttt{m}(g_1)(g_2)$ given the proof of the properties for g_1, g_2 : freeGroup A
- Proof that the element e satisfies the property
- Proof of the property for the element $\texttt{inv}(g)$ given the proof of the property for g : freeGroup A

The type of this induction function in our implementation is given as:

```
elimProp : {B : FreeGroup A → Type ℓ'}
        → ((x : FreeGroup A) → isProp (B x))
```

```
       → ((a : A) → B (η a))
       → ((g1 g2 : FreeGroup A) → B g1 → B g2 → B (m g1 g2))
       → (B e)
       → ((g : FreeGroup A) → B g → B (inv g))
       → (x : FreeGroup A)
       → B x
```

We note that it is not necessary to concern ourselves with how higher-dimensional objects are carried by induction. The reason that this data alone is sufficient to prove the property for each element of the free group, is that when a property is a type with truncated paths (h-Prop) all proofs are automatically equal with each other, and it is simply sufficient that there is at least one.

5.2 FreeGroupoid of A

In order to make our proof as general as possible and to remove additional assumptions, we introduce a new type based on the type A similar to the free group removing any restriction on the truncation level. In other words, in the inductive definition, there is no `trunc` constructor which would force the structure to be an h-Set.

```
data FreeGroupoid (A : Type ℓ) : Type ℓ where
  η     : A → FreeGroupoid A
  m     : FreeGroupoid A → FreeGroupoid A → FreeGroupoid A
  e     : FreeGroupoid A
  inv   : FreeGroupoid A → FreeGroupoid A
  assoc : ∀ x y z → m x (m y z) ≡ m (m x y) z
  idr   : ∀ x → x ≡ m x e
  idl   : ∀ x → x ≡  m e x
  invr  : ∀ x → m x (inv x) ≡ e
  invl  : ∀ x → m (inv x) x ≡ e
```

If we restore these restrictions taking the set truncation of `freeGroupoid` A then the result is homotopically equivalent to the free group of A. Therefore, whenever we wish to add this assumption, it is enough to simply assume that we are now working in the free group of A. This is provided to us through the Univalence Axiom and the next proposition of our library.

```
freeGroupTruncIdempotent : FreeGroup A ≡ ∥ FreeGroupoid A ∥₂
```

In Cubical Agda hierarchy of homotopy types has been chosen to start from 0 and not from -2 and therefore the set truncation is considered level 2.

For proving the theorem, we will use a common technique called encode-decode. The first step in this technique is the creation of homotopy equivalencies of `freeGroupoid` A with itself for each g : `freeGroupoid` A. The functions that we will use (and we call automorphisms of `freeGroupoid` A) are defined as follows

```
automorhpism : ∀ (a : FreeGroupoid A) → FreeGroupoid A → FreeGroupoid A
automorphism a g = m g a
```

The reader familiar with Group Theory will recognize that this is none other than the action of the "group" `freeGroupoid` A on itself, but reversing the action from $\mathtt{m}(a)(g)$ to $\mathtt{m}(g)(a)$. This has as a consequence that these automorphisms behave well to the various operations of `freeGroupoid` A.

We prove that the action of the product is the composition of the actions of the two factors (in the opposite order) and that the action of the identity is the identity function.

The next step is to prove that these automorphisms are indeed homotopy equivalences for every g : `freeGroupoid` A. And in this step, we have that the overall structure respects the operations that we have defined. More specifically, we have that the equivalence of the product is equal to the composition of the equivalences that are created from the two factors, the equivalence of the identity element is the identity equivalence and the equivalence of the inverse of an element is the inverse of the equivalence of the element.

The last step is to use the Univalence Axiom to create a path from `freeGroupoid` A to `freeGroupoid` A in the universe, from the equivalences, for every g : `freeGroupoid` A. And in this step, the structure of a group is preserved and transferred to the group of the paths with beginning and end the type `freeGroupoid` A in the universe. So, we have proved the following:

```
multPathsInUNaturality : ∀ (g1 g2 : FreeGroupoid A) →
    pathsInU (m g1 g2) ≡ (pathsInU g1) • (pathsInU g2)
idPathsInUNaturality : pathsInU {A = A} e ≡ refl
invPathsInUNaturality : ∀ (g : FreeGroupoid A) →
    pathsInU (inv g) ≡ sym (pathsInU g)
```

where pathsInU is the function corresponding to each element g : `freeGroupoid` A the path in $\mathcal{U}$ by the equivalences.

5.3 FundamentalGroup

Just like in Algebraic Topology, if we have a Type and one of its points it is possible to create the Type of the loops which begin and end to that exact point. This, of course, is the well-known type `Path` A `base base` which we symbolize in the proof as ΩA assuming that the base is known. However, in the case of the Homotopy Type theory, the type ΩA is not a group but a higher groupoid. To make it a group we take the set truncation and thus define the Fundamental group of A, denoted by π_1. It turns out that π_1 is indeed a group with the operation induced from the groupoid ΩA.

```
1π₁ : π₁ {base = base}
1π₁ = ∣ refl ∣₂

invπ₁ : π₁ {base = base} → π₁
invπ₁ = map sym

•-π₁ : π₁ {base = base} → π₁ → π₁
•-π₁ = rec2 π₁IsSet (λ p q → ∣ p • q ∣₂)
```

Similar to the case of `freeGroup` and `freeGroupoid`, we will remove from our assumptions all restrictions for the h-level of the fundamental group, and we will focus initially only on the groupoid ΩA.

5.4 Proof

First, we define in our theory the type of Bouquet of circles indexed by a type A, or ”wedge of A circles”, by the following definition as a higher inductive type.

```
data Bouquet (A : Type ℓ) : Type ℓ where
  base : Bouquet A
  loop : A → base ≡ base
```

We note the similarity that exists between this definition and the definition of the type of the circle. In this case, however, there is one more factor affecting the construction, the path structure of A. As `loop` is a function it transfers paths of A to paths in `Bouquet`A in all dimensions. This, potentially complicates the path structure of `Bouquet`A.

We then define the functions that will form the basis of the homotopy equivalence. Ignoring the truncation restrictions, we define the looping function from `freeGroupoid` A to Ω`Bouquet` with a very natural way respecting the algebraic structure of the two types.

```
looping : FreeGroupoid A → ΩBouquet
looping (η a)              = loop a
looping (m g1 g2)          = looping g1 • looping g2
looping e                  = refl
looping (inv g)            = sym (looping g)
looping (assoc g1 g2 g3 i) = pathAssoc (looping g1) (looping g2) (looping g3) i
looping (idr g i)          = rUnit (looping g) i
looping (idl g i)          = lUnit (looping g) i
looping (invr g i)         = rCancel (looping g) i
looping (invl g i)         = lCancel (looping g) i
```

For the winding function with opposite direction, from Ω`Bouquet` to `freeGroupoid` A we will utilize the work we did with the automorphisms of `freeGroupoid` A. We construct a family of types dependent on `Bouquet`A, a function `Bouquet`$A \to \mathcal{U}$. According to the recursion rule for this inductive type in order to define correctly a function with `Bouquet`A as domain, we must give an object $K : \mathcal{U}$ for `base` : `Bouquet`A and for every path $a : A \vdash$ `loop`(a) : `Path Bouquet`A `base base` a path with type `Path`$\,\mathcal{U}\ K\ K$.

We define the code family to be the family that sends `base` to the type `freeGroupoid` A and each path $a : A \vdash$ `loop`(a) : `Path Bouquet`A `base base` to the path resulting from the automorphism of $\eta(a)$. We define the function winding to send every path of Ω`Bouquet` to the object of `freeGroupoid` A to which is transferred the identity element e over this path in the code type family.

```
code : {A : Type ℓ} → (Bouquet A) → Type ℓ
code {A = A} base = (FreeGroupoid A)
code (loop a i)   = pathsInU (η a) i

winding : ΩBouquet → FreeGroupoid A
winding l = subst code l e
```

To complete the proof we need two more constructions, the proof that the looping function is left inverse with type

```
left-homotopy : ∀ (l : ΩBouquet {A = A}) → looping (winding l) ≡ l
```

and a proof that it is the right inverse

```
right-homotopy : ∀ (g : FreeGroupoid A) → winding (looping g) ≡ g
```

To be able to prove the first by path induction, we need at least one of the two ends of the path l : ΩBouquet not to be "constant". So we make an abstraction in our conclusion and we assume that we have x : Bouquet$A$ ⊢ l : Path Bouquet$A$ base x and thus the path in $\mathcal{U}$ has type Path $\mathcal{U}$ (freeGroup A) code(x). Furthermore, we adjust our functions. Winding is generalized to encode with type

```
encode : (x : Bouquet A) → base ≡ x → code x
```

and looping is generalized to decode.

```
decode : {A : Type ℓ}(x : Bouquet A) → code x → base ≡ x
```

Now, the requested proof is of type

```
decodeEncode : (x : Bouquet A) → (p : base ≡ x) → decode x (encode x p) ≡ p
```

The proof of Decode-Encode is done by path induction on p : base $\equiv x$ and applying in the proof base wherever x we get the result required

```
left-homotopy : ∀ (l : ΩBouquet {A = A}) → looping (winding l) ≡ l
left-homotopy l = decodeEncode base l
```

To prove that the looping function is right inverse of winding we need to reset the restrictions on the path structure of the two spaces and restrict our conclusion to them as well. In particular, we didn't prove the required right-homotopy but the propositional truncation of it.

```
truncatedRight-homotopy : (g : FreeGroupoid A) → ‖ winding (looping g) ≡ g ‖
```

Intuitively, we proved that it is indeed true that looping is right inverse of winding but by degenerating the conclusion into a space with a very simpler path structure. This alone is sufficient to obtain the equality of the set truncations of the general spaces base $\equiv x$ and code(x).

```
TruncatedFamiliesEquiv : (x : Bouquet A) → ‖ base ≡ x ‖₂ ≃ ‖ code x ‖₂
```

and from it using the Univalence Axiom and applying base to x : Bouquet$A$ we have

```
π₁Bouquet≡‖FreeGroupoid‖₂ : π₁Bouquet ≡ ‖ FreeGroupoid A ‖₂
```

However, we have already shown that the set truncation of freeGroupoid A is the free group of A and we finally get the result

```
π₁Bouquet≡FreeGroup : {A : Type ℓ} → π₁Bouquet ≡ FreeGroup A
```

Chapter 6

Future Work

In this chapter, we list some of the open problems and search fields regarding the theorem of the thesis but also of Homotopy Type Theory in general.

While it is clear what is the fundamental group of a Bouquet of circles, already from algebraic topology, when we set constraints on the higher path structure of the type, it isn't known what happens when we set no such constraint. More precisely, what is the loop space of $\texttt{Bouquet}A$, meaning $\Omega(\texttt{Bouquet}A, \texttt{base})$, and what is its behavior in every dimension? In fact, we don't even know if the type $\Omega(\texttt{Bouquet}A, \texttt{base})$ is h-Set or not. Such questions had not arisen in the study of these spaces within Set Theory where classic algebraic topology is done but only through the prism of the interpretation of spaces as higher categories and Homotopy Type Theory.

Likewise, many problems have their roots in algebraic topology and are studied in Homotopy Type Theory. One of the most common is the calculation of the fundamental groups of various dimensions of the sphere $\mathbb{S}^n$ for various n. These kinds of calculations are in the same category as the previous one. So, one wonders if there is a more straightforward way of calculating the path space of a space that doesn't use the method encode-decode and that is quite general. Also, one could study the structures of surfaces that are not orientable, such as the projective plane, Klein's bottle or Möbius strip. How can the concept of orientation of a surface, or even the concept of a surface itself, be expressed in the context of Homotopy Type Theory?

Moreover, we saw that Cubical Type Theory gives a computational meaning to our theory. However, there are still many interesting points of the metatheory of HoTT. Cubical Type Theory itself is an important research field at the moment as well as other theories with similar logic. Additionally, even if we have given an intuitive presentation of inductive types, there is a syntactically formal way of defining them. It is unknown whether there is a corresponding rigorous definition of higher inductive types. Another issue is whether it is possible to replicate HoTT within itself! That is, how can we study the syntax of Homotopy Type Theory within HoTT. This issue is often named ”type theory eating itself” or ”type theory in type theory”.

Another very important field of research is the field of semantics. A major issue in this area is the Initiality Conjecture. We assume that there is a way of describing all ”languages” of a particular kind, a signature, and a way of describing all “models” of this language in one category, with the morphisms between them being ”interpretations” of one language within another. The conjecture says that there is a model, the model of the syntactic terms of the language, which is an initial object in the category of models. The conjecture has been proven for languages such as calculus of constructions and λ-calculus but not for more complex languages such as Dependent Type Theory or Homotopy Type Theory. In this way, it is possible to study several languages together and answer questions about normalization or completeness theorems.

Finally, crucial for everyday mathematical practice is writing rich libraries with what is already known in all branches of mathematics with the help of proof assistants. Indicatively, we mention the libraries UniMath and HoTT in Coq and HoTT-Agda and Cubical Agda in Agda and Cubical Agda respectively.

Bibliography

[Baue16] A. Bauer, “Five stages of accepting constructive mathematics”, *Bulletin of the American Mathematical Society*, vol. 54, pp. 481–498, 2016.

[Cohe16] Cyril Cohen, Thierry Coquand, Simon Huber and Anders Mörtberg, “Cubical Type Theory: a constructive interpretation of the univalence axiom”, 2016.

[Fral03] J.B. Fraleigh and V.J. Katz, *A First Course in Abstract Algebra*, Addison-Wesley series in mathematics, Addison-Wesley, 2003.

[Harp12] Robert Harper, “Extensionality, Intensionality, and Brouwer’s Dictum”, http://existentialtype.wordpress.com/2012/08/11/extensionality-intensionality-and-brouwers-dictum/, August 2012.

[Hatc02] A. Hatcher, Cambridge University Press and Cornell University. Department of Mathematics, *Algebraic Topology*, Algebraic Topology, Cambridge University Press, 2002.

[Krau19] Nicolai Kraus and Jakob von Raumer, “Path Spaces of Higher Inductive Types in Homotopy Type Theory”, 2019.

[Lica13] Daniel R. Licata and Michael Shulman, “Calculating the Fundamental Group of the Circle in Homotopy Type Theory”, 2013.

[MacL71] Saunders MacLane, *Categories for the Working Mathematician*, Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[Mart72] Per Martin-Löf, “An intuitionistic theory of types”, 1972.

[Munk00] J.R. Munkres, *Topology*, Featured Titles for Topology, Prentice Hall, Incorporated, 2000.

[Univ13] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[Vezz19] Andrea Vezzosi, Anders Mörtberg and Andreas Abel, “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types”, *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, jul 2019.

[Av09] Α. Διονύσιος Αναπολιτάνος, *Εισαγωγή στη Φιλοσοφία των Μαθηματικών*, Εκδόσεις Νεφέλη, 2009.

Appendix A

Symbols

Γ : Type Context
$\Gamma\vdash$: Entailment in the context
$\equiv$: Judgmental Equality
$:\equiv$: Definition
$::=$: Definition of grammar
`id` : Identity map
`type` : Type
$\circ$: Function Composition
$\mathbf{1}$: Unit Type
$\langle\rangle$: Trivial element of the Unit Type
$\mathbf{0}$: Empty Type
`abort` : Recursion in the Empty Type
`fst` : First projection of pair
`snd` : Second projection of pair
$\langle a, b\rangle$: Pair
$A \to B$: Function with domain A and codomain B
`inl` : First coprojection of coproduct
`inr` : Second coprojection of coproduct
`case` : Elimination in coproducts
`bool` : Bool Type
`tt` : True
`ff` : False
`if` M `then` N_1 `else` N_2 : Elimination in Bool
`rec` : Recursion
`ind` : Induction
`zero` : Zero in Natural Numbers
`suc` : The successor function of Natural Numbers
$\mathbb{N}$, `nat` : Natural Numbers Type
`Seq` : The type of all finite sequences of Natural Numbers
`list`(A) : Type of linked lists of elements from type A
`nil` : Empty list
x `cons` xs : List constructor cons
`Id` : Identity Type
`refl` : Reflexive path
`J` : Path induction
`sym` : Inverse path operation
`trans` : Path concatenation
`ap` : Function application on path
`tr` : Transport
`dap` : Dependent function application on path
$\cdot$: Path concatenation

$\mathbb{I}$: Interval Type
$\mathbb{S}^1$: Circle Type
`base` : Base point of the circle type
`loop` : Non trivial path of the circle type beginning and ending at `base`
$\mathcal{U}$: Universe of Types
`ua` : Univalence Axiom
`ua-equiv` : Equivalence of types proven by identity of `ua`
`funext` : Function Extensionality
`happly` : Homotopy between identified functions
$\simeq$: Equivalence of Types - Homotopy of functions
isEquiv : Type of proofs that a function is an equivalence of types
`idtoeqv` : Equivalence of equal types
`idEquiv` : Identity equivalence
$\circ_{\simeq}$: Composition of equivalences
`equiv-inv` : Inverse equivalence
`equiv-compose` : Composition of equivalences
$\|A\|$: Propositional Truncation
$|x|$: Propositional Truncation projection
`squash` : Propositional Truncation identity constructor
$\|A\|_0$: Set Truncation
$|x|_0$: Set Truncation projection
`squash`$_0$: Set Truncation identity constructor
`Path` $A\ x\ y$: Type of paths in A between $x, y : A$
$\sim x$: Involution in $\mathbb{I}$
$x \vee y$: Join in $\mathbb{I}$
$x \wedge y$: Meet in $\mathbb{I}$
`i0` : Minimum element of $\mathbb{I}$
`i1` : Maximum element of $\mathbb{I}$
`hcomp` : Homogeneous composition
`transp` : Transport - Coercion
`transport` : Transport
$0_{\mathbb{F}}$: Minimum partial ellements
$1_{\mathbb{F}}$: Maximum partial ellements
`Glue` : Glue Type
`glue` : Glue constructor
`unglue` : Glue elimination function
$\mathcal{O}$: Set of objects of a category
$\mathcal{A}$: Set of morphisms of a category
$\mathcal{C}$: Category
`dom`f : Domain of f
`cod`f : Codomain of f
`Hom`(a, b) : Set of morphisms with domain a and codomain b
`freeGroup`A : Freely generated group by the elements of A
`freeGroupoid`A : Freely generated groupoid by the elements of A
`m` : Multiplication function
`e` : Identity element
`inv` : Inverse element
`assoc` : Proof of associativity
`idr` : Proof of right law of identity
`idl` : Proof of left law of identity
`invr` : Proof of right law of inverse element

`invl` : Proof of left law of inverse element
`trunc` : Proof of identity for group's Set Truncation
`Bouquet`A : Bouquet of circles indexed by the elements of A