Martin-Löf Type Theory, Homotopy Type Theory and Agda

Natalie Ravenhill
Supervisor: Martín Escardó

School of Computer Science
University of Birmingham

Abstract

Martin-Löf Type Theory, Homotopy Type Theory and Agda

Natalie Ravenhill

In this Individual Study module I have learned Martin Löf Type Theory (MLTT), Agda and the basics of Homotopy Type Theory (HoTT). Agda is both a dependently typed programming language and a language in which mathematics, including theorems, definitions and proofs, can be formalized and checked by the computer. In this report I demonstrate my understanding MLTT and HoTT by explaining how to formalise them in Agda. This is done by constructing functions which represent proofs of propositions, the propositions being the types of the functions. This is due to the BHK interpretation of the Curry Howard Correspondence between intuitionistic logic and types. We formalise some types that are used to define MLTT and begin to formalise HoTT, starting with properties of equality and the Eckmann Hilton theorem.

Contents

Chapter 1

Introduction

My interest in the topics I have chosen for my Individual Study began as a result of the Agda Club that ran as an optional part of the Functional Programming module last year. We briefly discussed Homotopy Type Theory (HoTT) as it is built on Martin Löf Type Theory (MLTT), a variant of which is used as the basis for Agda. It was interesting to see that Homotopy Type Theory uses concepts from Topology to help solve problems in Computer Science and vice versa, as I had not previously encountered a link to Topology within Computer Science.

In summary, in this study I have learned Martin Löf Type Theory (MLTT), Agda and the basics of Homotopy Type Theory (HoTT). In this report I demonstrate my understanding MLTT and HoTT by explaining how to formalise them in Agda. This is done by constructing functions which represent proofs of propositions, the propositions being the types of the functions. This is due to the BHK interpretation of the Curry Howard Correspondence between intuitionistic logic and types (Chapter 2.8). We formalise some types that are used to define MLTT and begin to formalise HoTT, starting with properties of equality and the Eckmann Hilton theorem (Chapter 3.2).

Writing proofs in a programming language instead of on paper is advantageous because the proofs in MLTT and HoTT use intuitionistic logic. In this logic, instead of propositions being true or false, we can either construct a proof of it or we can't. Therefore using the Curry Howard Correspondence between proofs and programs we can write functions of a certain type that are proofs of a certain proposition.

We specifically use Agda to formalise HoTT because it is a dependently typed programming language and MLTT is based on dependent types, so it is easy to construct dependent types using Agda. There are also other useful features of Agda, for example, it has an interactive mode, where the user can give a partial proof and the type checker can create "holes" to

be filled in. Sometimes what is needed to complete the proof can be filled in using type inference, or the hole can be refined from a parameter, to give a smaller hole to fill in. This makes writing and checking proofs much easier than writing the whole proof and then running the type checker to ensure its correctness.

Another feature of Agda is that source files can be used to produce LaTeXdocuments using the extension ".lagda". This report was printed from a pdf file that was generated from a ".lagda" file, so if that file were to be loaded into Agda, we can use Agda's typechecker to show that the proofs are correct. This colour version of this file, the coloured text is the Agda code.

1.1 Martin Löf Type Theory

Type theory is a formal system that is an alternative to set theory as a foundation of mathematics. Instead of forming sets, we define types and create elements of those types. We use rules to form the types themselves, and to construct and eliminate elements of those types. This is in contrast to set theory, where sets can be formed of any elements and then rules are applied by using a separate logic system (the most simple being First Order Logic).

The logic of type theory is **Intuitionistic (or Constructive) Logic**. In this logic, for a proposition to be true, we must be able to construct a proof of it and this proof will be a meaningful mathematical object. In type theory we do this by having propositions as types and elements of these types being the proofs of propositions. Therefore unlike in classical logic it is not simply the case that a proposition is either true or false, as we may not know if a proof of it can be constructed. For example, if we have an undecidable proposition, of which there are many in Computer Science (such as the halting problem), then we do not know if we can construct a proof of it or not. This means that the *law of extended middle* in classical logic ($A \vee \neg A$), will not hold constructively. However we can use other proof strategies, so intuitionistic logic is not inferior to classical logic.

The type theory that is the basis of HoTT is Martin Löf (or Dependent / Intuitionistic/ Constructive Type Theory). It is similar to simply typed λ -calculus in the sense that any type can be defined and the same rules for constructing and eliminating elements of those types apply, but now dependent types are included. A dependent type is a type that can depend on elements of another type. For example, the type $x = y$ depends on points x and y of some type (or in some space) X. Other examples of dependent types include vectors of a specified length and trees of a specific height. In these examples, elements of the types will depend on an element of the type of natural numbers. Also as previously described, Martin-Löf Type Theory is built on intuitionistic logic, so propositions can be defined as types and proved by constructing elements of those types.

1.2 Homotopy Type Theory

Homotopy Type Theory (HoTT) is an extension of Martin Löf type theory that uses concepts from homotopy theory as defined in [Univalent Foundations Program, 2013]. Homotopy is a branch of topology that describes when two objects in a topological space can be continuously deformed into each other. In HoTT, when you have an element a of a type A, you equivalently have a point a in a space A. Then in type theory we have a type representing equality, so if two elements a and b are of the same type, we can construct the equality type $a = b$. In HoTT, this does not just represent equality but also a path space. A proof of $a = b$ is exactly the same as defining a path from point a to point b in a space A. There may be more than one proof that $a = b$ in type theory, just as there may be more than one path from a to b. Representing types in this way allows us to think about proofs in Homotopy Theory from a different point of view, but also to define types that were previously impossible to define in type theory, such as higher inductive types (types that are defined inductively on paths and higher paths). This is done using path induction, a form of induction that relies on the ability to think of equality as paths and something we will return to later.

1.3 Agda

Agda is a dependently typed programming language and theorem prover, based on Martin Löf type theory, defined in [Norell, 2007]. Its syntax is similar to Haskell, but unlike Haskell, dependent types can be defined. For example, as in [Bove and Dybjer, 2009] we can define vectors as follows:

```
-first define natural numbers:
data Nat : Set where
  zero : Nat
  succ : Nat → Nat

-then we can use this type to inductively define vectors
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n
```

Here Nat and Vec are data declarations, so we do not need to prove them, just provide constructors, so that these types can be constructed in proofs. We define them by specifying their name, then any parameters they may have, (e.g. Vec has a parameter A which is any

Set, but is also a function from $Nat \to Set$, so we have another parameter that is a natural number). Then anything that follows "where", is a constructor, specified similarly to type defintions in Haskell.

The type Nat is an element of a type Set (the type describing a set of objects), because Nat is the Set of all natural numbers. It is defined by pattern matching, so all elements of it are either successors of successors of the element $zero$ or $zero$ itself. Then $succ$ is just a function that takes a natural number and returns its successor.

Vec is a type that depends on two other types. It takes an element of Nat and creates a vector of elements of another set type, A. If the element of Nat is $zero$, it defines an empty vector, otherwise it defines a function which takes a number and a set A, then returns a vector using these two types. The vector will be represented as n elements of the set A, separated by the symbol ::

Modifications to use Agda for Homotopy Type Theory. To define Martin Löf Type Theory, we use a book that has been created as a result of a year of research into Homotopy Type Theory, [Univalent Foundations Program, 2013]. This book is the result of a project that seeks to define a new foundation of mathematics, called univalent foundations, based on HoTT extended with a univalence axiom (stating that identity is equivalent to equivalence).

The type theory defined in the HoTT book is the intensional version of MLTT, defined in [Martin-Löf, 1975]. In intensional type theory, judgemental equality and propositional equality are not the same. Judgemental equality means that two things are definitionally equal and propositional equality needs to be proved. We need this for HoTT because identity types are equivalent to path spaces which are mathematical objects, so to prove identity we can create an element of the relevant path space. This means that identity type is proof relevant (ie. that a proof is also a mathematical object).

Agda is based on a variant of intensional MLTT, so there are not too many modifications that need to be made. However, to ensure that we have a proof relevant interpretation we must remove anything that allows an extensional version of type theory:

The K Axiom The K Axiom, by [Streicher, 1993], states that the only proof of equality is reflexivity (up to judgemental equality). It can be used optionally in Agda, so we do not want to use it, as adding it would make Agda's type system extensional and therefore incompatible with HoTT.

We can disable this in Agda by using the option "–without-K".

Now we have everything we need to define MLTT in Agda.

Chapter 2

Martin Löf Type Theory in Agda

In MLTT we can use different types to represent many things, such as functions, products, natural numbers and booleans. There is a general form of defining types, as specified in [Univalent Foundations Program, 2013]:

1. **Formation rules** are used to create types, sometimes using other types (i.e. if they are dependent)

2. **Introduction rules** are used to construct elements of a given type

3. **Elimination rules** define how elements of a type are used (so they are converted to another type and the current type is removed)

4. **Computation rules** are the rules used to do the elimination

5. A **Uniqueness Principle** is used to show the uniqueness of maps to/from a type. It shows that for maps into a type, every element of a type is uniquely determined by the result of applying an eliminator to it and dually can be reconstructed using introduction rules. Then maps from a type are uniquely determined from data. The uniqueness rule is optional and not used for every type.

Now we can define types using these rules. The definitions themselves follow those of [Univalent Foundations Program, 2013], which we will then formalise in Agda:

2.1 Function Types

Forming the Function Type

A function is a mapping from any type A, to any other type B (or itself). So given any types A and B, the formation rule forms the type $A \to B$. Function types are a primitive concept in MLTT, so are built into Agda and don't need to be redefined.

Constructing elements of function types

Elements of a function $A \to B$ can be constructed in two ways:

1. **Direct Defintion**:

   Given a function $f : A \to B$ and an element of the domain $a : A$, we can apply the function to obtain an element of the codomain, $\Phi : B$:

   ```
   {-# OPTIONS --without-K #-}
   U = Set

   fdef : {A B : U} (f : A → B) (x : A) {Φ : B} → B
   fdef f x {Φ} = Φ
   ```

   This is our first example of a proof. We want to show that f applied to any $x : A$ is the same as an element $\Phi : B$. The type $fdef$ is a proposition that takes a function $f : A \to B$, where given an argument in A, it will return a value of type B. Then we give it an $x : A$ as another parameter then an implicit parameter $\Phi : B$ which is optional to give in the function defintion. Then this proposition correpsonds to a function that returns an element of type B.

   Then we define $fdef$ with f and x, so that $f\ x$ is an element of B (and will be equivalent to an expression $\Phi : B$ as we have assumed $x : A$). The argument to $fdef$ in curly brackets, is an implicit argument in Agda, which means Agda can infer the argument itself, as the function can be defined by any expression of type B (but here we give an element Φ, which is an expression that will be the function (i.e. if $f(x)$ is defintionally equal to Φ, then Φ is the codomain of the function.)

   Also, $U = Set$ means that everything we define will be within the same universe, which is represented by a set. We can also define higher universes, but for now we want to define all of our types in the same universe.

2. **λ-abstraction**

To define a function without giving it a name, we can also use λ-abstraction. For example if we have $\Phi : B$ which uses an element $x : A$, the lambda abstraction will be $\lambda(x : A).\Phi$ and then we can show that this will have the type $A \to B$. So $\lambda x \to \Phi$ is a λ-abstraction on an element $x : A$ and the expression $\Phi : B$:

```
λ-abs : {A B : U} → A → B → (A → B)
λ-abs x Φ = λ x → Φ
```

For example, for any types A and B and any element $y : B$, we can have a **constant function** $\lambda(x : A).y$ (where no matter what the value of x is, the result will always be y):

```
const : {A B : U} → (y : B) →  (A → B)
const y  = λ x → y
```

(Note: Agda infers the type of x as A, so you don't need to write the types as $x : A$ and use $\to$ in place of . so $\lambda(x : A).\Phi$ will be $\lambda x \to \Phi$ in Agda)

We can also use **implicit lambda abstraction** in Agda. For example, in an expression $\lambda x.\Phi : A \to B$, it is obvious from the type that x must be an element of A. The underscore symbol can be used in place of a variable to denote implicit lambda abstraction, so in this example, $\lambda y \to g\ x\ y$ can be written as $g\ x\ \_$, because $y : B$ is inferred from the type $B \to C$:

```
impλ : {A B C : U} (g : A → B → C) (x : A) (y : B) → B → C
impλ g x _ = λ y → g x y
```

Computation Rule for Function Types

Applying λ-abstraction is equivalent to the computation rule, so for an argument $a : A$, given $\lambda x.\Phi$, a replaces all ocurrences of $x : \Phi$. Therefore $(\lambda x.\Phi)(a)$ is judgementally equal to replacing x with a in Φ:

```
computation-rule : {A B : U} → A → B → (A → B)
computation-rule a Φ = λ-abs a Φ
```

This is equivalent to β reduction, where an eliminator is applied to a constructor (so lambda abstraction is applied to the lambda expression).

Uniqueness Principle for Function Types

The uniqueness principle for function types shows that f is uniquely determined by its values. From any function $f : A \to B$, we can construct a λ-abstraction function $\lambda\ x \to f\ x$: (ie. the eta rule for functions (simplifying a term with a constructor applied to an eliminator)).

```
eta-rule : {A B : U} → (f : A → B) → A → B
eta-rule f = λ x → f x
```

Dependent Functions can then be defined in a similar way:

2.2 Dependent Function Types

Dependent function types are also referred to as Π types. Dependent functions are functions where the codomain can be different depending on what the domain of the function is.

Formation Rule for Π types

Given a type $A : U$ and a dependent type $B : A \to U$, (a type where a value must be obtained by giving a value of A, or equivalently the family of all elements in U given an element of A), we may construct the type of dependent functions: $\Pi_{(x:A)}B(x)$.

```
{-# OPTIONS --without-K #-}

U = Set

Π : {A : U} (B : A → U) → U
Π {A} B = (a : A) → B a
```

If B is a constant family (so B does not depend on A), then this type is the same as the non-dependent function type:

```
Π-const : {A B : U} → U
Π-const {A} {B} = A → B
```

Introduction Rule for Π types

We can create dependent functions using direct defintions or by using λ-abstraction, just as we did for function types. To define a dependent function, $f : \Pi_{(x:A)} \to B\ x$, we need an expression $\Phi : B\ x$ (where $x : A$). Then in Agda we can show that $f\ x$ is equivalent to Φ as we did for function types:

```
Π-fdef : {A : U} {B : A → U} (f : (x : A) → B x ) (x : A) {Φ : B x} → B x
Π-fdef f x {Φ} = Φ
```

Then to define a dependent function using λ-abstraction we can have $\lambda x \to \Phi$ where $\Phi : B(x)$ uses $x : A$.

```
abs : {A : U} {B : A → U} {Φ : (x : A) → B x} → (x : A) → B x
abs {A} {B} {Φ} = λ x → Φ x
```

Computation rule for Π types

The computation rule will be the same as for function types, applying λ-abstraction to the dependent function, so the function will be applied to an $x : A$ and an element in $B\ x$ will be returned :

```
comp : {A : U} {B : A → U} {Φ : (x : A) → B x} → (x : A) → B x
comp {A} {B} {Φ} x = abs {A} {B} {Φ} x
```

Uniqueness Principle for Π types

The uniqueness principle will also be the same as for function types, that given a function f and an argument x, $f \equiv (\lambda x.f(x))$. This can be represented in Agda as:

```
Π-eta : {A : U} {B : A → U} → (f : (x : A) → B x) → (x : A) → B x
Π-eta f = λ x → f x
```

Examples of Dependent Function Types

Polymorphic function types work just like parametric polymorphism in languages such as Haskell. Polymorphic function types take a type as an argument, then the function is defined on elements of that type. The simplest example is the polymorphic identity function, which given any element of any type, will just return itself. This is $id \equiv \lambda(A : U).(x : A).x$ and in Agda it will be:

```
id : (A : U) → A → A
id = λ A x → x
```

This says given a type, A, and an element of that type, x, it will return that element.

Another example of a polymorphic function is the swap function, which changes the order of the arguments of a two argument function. The types in the function, A, B and C can be any type, and then given a function $g : (A \to B) \to C$ it will return a function of type $(B \to A) \to C$. We can use g again as it takes two arguments in U and returns a value of C, so the order does not matter. The function type is defined as being left associative in Agda so we do not need to use brackets, but they can be useful for clarity. Also we can use currying to get a two argument function (where the result of a function is then applied to another function) as the function type is only defined on one argument. Then in Agda we write:

```
swap : (A B C : U) → (A → B → C) → (B → A → C)
swap A B C g = λ b a → g a b
```

The type arguments could also be defined implicitly (in curly brackets in Agda) but for clarity we can also define them as explicit arguments. If we wanted to use implicit arguments we can do so as follows:

```
swap' : {A B C : U} → (A → B → C) → (B → A → C)
swap' {A} {B} {C} g b a = g a b
```

The arguments to swap can also be dependent function types, so the other arguments may depend on the first. The second domain may depend on the first one and the codomain may depend on both of them, so given $f : (x : A) \to (y : B\ x) \to C\ x\ y$ and arguments $a : A$ and $b : B$, we write $f\ a\ b : C\ x\ y$:

```
dependent-swap : {A : U} {B : A → U} {C : (x : A) → B x → U}
  {x : A} {y : B x} → (A → B x → C x y) →
  (B x → A → C x y)
dependent-swap {A} {B} {C} {x} {y} f b a = f a b
```

2.3 Universes

A Universe is a type that contains other types. All elements belonging to a universe, U, are called **small types**. Universes are at different levels, so a universe $U_{(n+1)}$ will contain all of the types in a universe below it, U_n. This means universes in MLTT are **cumulative**.

However it also means that elements do not have a unique type as was could have $a : A : U_n : U_{(n+1)} \ldots$ To abbreviate this we can use **typical ambiguity** (as defined in [Univalent Foundations Program, 2013]), meaning that we can read $U_i : U_{(i+1)}$ and so on as $U : U$, assuming that universe levels are assigned consistently.

We cannot define a universe that contains all other types as this would give us a type theoretic version of Russell's Paradox in set theory (that if R is the set of all sets that don't contain themselves then R must be a member of itself, which is impossible). If we have the universe of all universes as this would be the same as saying the set of all sets - every type would then be inhabited, including the empty type. We will see later that by representing propositions as types, the empty type corresponds to false, so this is unsound. Russell's paradox can be directly encoded into type theory by representing sets as trees, as in [Coquand, 1991]. These types are well founded trees, or **W types**. They are specified in Univalent Foundations Program [2013] as $W_{(a:A)}B(a)$, a type with two parameters A (which represents the labels of the nodes of the tree) and $B\ a$ (which defines the arity (how many branches the nodes will have)). To formalize this paradox, we can use the option below in Agda:

```
{-# OPTIONS –type-in-type #-}
```

A formalization in Agda by Thorsten Altnkirch using this is available at `http://www.cs.nott.ac.uk/~txa/g53cfr/l20.html/l20.html`

Using "–type-in-type" disables universe level checking in Agda which means we can show that $Set : Set$. We do not usually use this option as it makes Agda inconsistent.

2.4 Product Types and Dependent Product Types

Formation Rules for Product Types and Σ types

The product type represents what would be the cartesian product in set theory. However in type theory the cartesian product does not have to be defined as particular sets collected into ordered pairs as it would be in set theory. So given two types A and $B : U$, we have the type $A \times B : U$.

The Σ type is the **Dependent Pair Type**. This is the same as a product type except now the type of the second element of each pair depends on that of the first, so given $A : U$ and $B : A \to U$ we can define $\Sigma_{(x:A)}B\ x : U$. Then we can define the product type as a constant version of the dependent pair type. In Agda this is represented as follows:

```
{-# OPTIONS –without-K #-}
```

```
U = Set

-dependent pair type
data Σ {A : U} (B : A → U) : U where
_,_ : (a : A) → (b : B a) → Σ B

-product type
_×_ : U → U → U
A × B = Σ (λ (a : A) → B)
```

There is also another type concerned with product types, the **unit type**. This is the nullary product type, so it is a pair that contains no elements. It is defined in Agda as:

```
data 𝟙 : U where
  ⋆ : 𝟙
```

The only element of $\mathbb{1}$ is $\star$.

Introduction Rules for Product Types and Σ types

Σ takes two arguments, a type A and a type B that is a dependent type on A. The type will be written as $(a\ ,\ b)$ where $a : A$ and $b : B\ a$.

$\times$ takes in two arguments, any two types in U (which may or may not depend on each other). Then the defintion says that for any element $a : A$ there will be a corresponding element in B to form a pair with. So to construct an element of $\times$, given an element $a : A$ and a and $b : B$ we form the type $(a\ ,\ b) : A \times B$

The unit type has only one element, $\star$, so to construct an element of it we just have $\star : \mathbb{1}$

Elimination Rules for Product Types

To eliminate product types we need to define functions out of them. To define a non-dependent function out of $A \times B$, of type $f : A \times B \to C$, where C is an arbitary type, we get the result when f is applied to a pair $(a\ ,\ b)$.

We can do this by introducing the elimination rule, which says that for any function $g : A \to B \to C$ on two arguments, we can define a function $f : A \times B \to C$ by $f((a, b)) \equiv g(a)(b)$. In Agda we write this as:

```
f : {A B C : U} {g : A → B → C} → A × B → C
f {A} {B} {C} {g} (a , b) = g a b
```

In [Univalent Foundations Program, 2013] it is stated that in type theory we assume that a function on $A \times B$ is well defined as soon as we specify its value on tuples and from this we can prove every element of $A \times B$ is a pair (unlike in set theory where f would be defined on a set of pairs that have already been defined). Therefore this is enough to define f on any pair.

An example of using this elimination rule is for getting the original types A and B back from a pair $A \times B$. To do this we define projections:

```
pr₁ : {A B : U} → A × B → A
pr₁ {A} (a , b) = a

pr₂ : {A B : U} → A × B → B
pr₂  (a , b) = b
```

So for pr_1, given a pair $(a\ ,\ b)$ we get a and for pr_2 given the same pair we get b.

Instead of using the principle of function definition every time we want to define a function we can use it once in a universal case and apply the resulting function in all other cases. This is done using a **recursor**as defined in [Univalent Foundations Program, 2013]. The recursor for product types is a function of type $rec_{A\times B} : \Pi_{(C:U)}(A \to B \to C) \to A \times B \to C$, defined as $rec_{A\times B}(C, g, (a, b)) \equiv g(a)(b)$. C is the return type of the function, g is the function we are applying and $(a\ ,\ b)$ the pair we are applying the function to We can write this in Agda as:

```
×-rec : {A  B : U}→ (C : U) → (A → B → C) → A × B → C
×-rec  C g (a , b)= g a b
```

So now we can redefine the projection functions using the recursor.

```
pr₁' : {A B : U} → A × B → A
pr₁' {A} {B} = ×-rec A (λ a b → a)

pr₂' : {A B : U} → A × B → B
pr₂' {A} {B} = ×-rec B (λ a b → b)
```

For pr_1, C is replaced by A (the return type), and the function being applied is $(\lambda a\ b \to a)$. Then pr_2 is defined similiarly.

Although this is called a recursor, it does not actually use recursion. This is also called the recursion principle for cartesian products, meaning we can define a function $f : A \times B \to C$ by giving its value on pairs.

The recursor can also be defined using the projections:

```
×-rec' : {A  B : U}→ (C : U) → (A → B → C) → A × B → C
×-rec' C g (a , b) = g (pr₁ (a , b)) (pr₂ (a , b))
```

We also have a recursor for the unit type, of type $rec_{\mathbb{1}} : \Pi_{(C:U)}\ C \to \mathbb{1} \to C$ defined by $rec_{\mathbb{1}}(C, c, \star) \equiv c$. The C is again the return type and c the function being used. $\star$ is the 3rd argument as there is only one element the recursor can be used on. We define it in Agda as:

```
𝟙-rec : (C : U) → C → 𝟙 → C
𝟙-rec  C c ⋆ = c
```

To define **dependent functions** over the product type using the recursor, it has to be generalised, so given $C : A \times B \to U$, we can define a function $f : (x : A \times B) \to C\ x$ by providing a function $g : (x : A)\ (y : B)\ C\ x\ y$

```
f' : {A B : U} {C : A × B → U} {g : (x : A) → (y : B) → C (x , y)}
   → (x : A × B) → C x
f' {A} {B} {C} {g} (x , y) = g x y
```

We can use this to prove the propositional uniqueness principle for product types which we will do in the relevant section. Defining dependent functions in this way means that we can prove a property for all elements of a product just by proving it for the tuples, in a similar way to how use proof by induction on natural numbers. So using this principle once in the universal case, the function we get is called **induction** for product types: given $A, B : U$ we have $ind_{(A\times B)} : \Pi_{(C:A\times B\to U)}(\Pi_{(x:A)}\Pi_{(y:B)}\ C((x\ ,\ y))) \to \Pi_{(x:A\times B)}\ C\ x$. The return type C is a family of product types, and the function g is a function from any $x : A$ and $y : B$ to a return type depending on these, $C\ (x\ ,\ y)$:

```
×-ind : {A B : U} (C : A × B → U) → ((x : A) → (y : B) → C (x , y))
   → (x : A × B) → C x
×-ind C g (a , b)= g a b
```

A dependent function can be obtained from the induction principle of the cartesian product, so the recursor is just a special case of induction where the family C is constant:

```
×-rec'' : {A  B : U}→ (C : U) → (A → B → C) → A × B → C
×-rec'' {A} {B} C g (a , b) = ×-ind  {A} {B} (λ _ → C) g (a , b)
```

Induction is often referred to as a dependent eliminator (and recursion is therefore the non-dependent eliminator).

Elimination Rules for Σ types

The constructions on Σ-types are generalisations of the ones for product types, where dependent functions replace non-dependent ones. For example, for the recursion principle out of a Σ-type $f : \Sigma_{(x:A)} B(x) \to C$ we provide a function $g : \Pi_{(x:A)} B(x) \to C$. Then define f by $f((a\ ,\ b)) \equiv g(a)(b)$ Then in Agda we represent this as:

```
F : {A C : U} {B : A → U} {g : (x : A) → B x → C} → Σ (λ (x : A) → B x) → C
F {A} {C} {B} {g} (a , b) = g a b
```

We can define projections using this eliminator, as we did for product types. To define projections out of a Σ type, the first one can be defined straight out of the type:

```
Σpr₁ : {A : U} {B : A → U} → Σ (\ (x : A) → B x) → A
Σpr₁ (a , b) = a
```

The second projection is less simple. As the type of the second component of the pair depends on the first, Σpr_2 must be a dependent function, whose type involves the first dependent function, as we need the value of it to form the type:

```
Σpr₂ : {A : U} {B : A → U} → (p : Σ \ (x : A) → B x) → B (Σpr₁ p)
Σpr₂ (a , b) = b
```

Therefore we need to use the induction principle for Σ types, which says to construct a dependent function out of a Σ type into the return type $C : (\Sigma_{(x:A)} B(x)) \to U$ we need a function $g : \Pi_{(a:A)} \Pi_{(b:Ba)} \to C((a\ ,\ b))$ we can define a function $f : \Pi_{(p:\Sigma_{(x:A)} B(x))}\ C(p)$. This function will be $f((a\ ,\ b)) \equiv g(a)(b)$. We represent this in Agda as:

```
Σf' : {A : U} {B : A → U} {C : (Σ \ (x : A) → B x) → U}
   {g : (a : A) → (b : B a) → C (a ,  b)} → (p : Σ ( \ (x : A) → B x)) → C p
Σf' {A} {B} {C} {g} (a , b) = g a b
```

Then we can redefine Σpr_2 using this induction principle, replacing $C(a\ ,\ b)$ with $B\ (\Sigma pr_1 p)$ as $C(a\ ,\ b) \equiv B\ (\Sigma pr_1 p)$. So applying $C\ p \equiv B\ (\Sigma pr_1 p)$ will get Σpr_2 (as above). To prove this is correct, note that $B\ (\Sigma pr_1 (a, b)) \equiv B\ a$ using the definition for Σpr_1 and $B\ a$

```
Σpr₂' : {A : U} {B : A → U} {g : (a : A) → (b : B a) →
   B(Σpr₁ {A} {B} (a , b))} → (p : Σ ( \ (x : A) → B x)) → B (Σpr₁ p)
Σpr₂' {A} {B} {g} (a , b) = g a b
```

What we have defined is just the recursion and induction principles for Σ-types. The recursor takes a type A and a type B that depends on A, to be the two components of the pair, then C is the return type that the function g is applied to. The induction is exactly the same, except now the type of C is a dependent type on pairs. They are represented in Agda as:

```
Σ-rec : {A : U} {B : A → U} (C : U) → ((x : A) → B x → C)
  → (Σ \ (x : A) → B x) → C
Σ-rec C g (a , b) = g a b

Σ-ind : {A : U} {B : A → U} (C : (Σ \ (x : A) → B x) → U)
      → ((a : A) → (b : B a) → C (a , b)) → ((p : Σ \ (x : A) → B x) → C p)
Σ-ind C g (a , b) = g a b

rec-claim : {A : U} {B : A → U} (C : U) → ((x : A) → B x → C)
          → (Σ \ (x : A) → B x) → C
rec-claim {A} {B} C g (a , b) = F {A} {C} {B} {g} (a , b)

ind-claim : {A : U} {B : A → U} (C : (Σ \ (x : A) → B x) → U)
            → ((a : A) → (b : B a) → C (a , b)) → ((p : Σ \ (x : A) → B x) → C p)
ind-claim {A} {B} C g (a , b) = Σf' {A} {B} {C} {g} (a , b)
```

So the principle F corresponds to recursion on Σ types and $\Sigma f'$ for dependent functions corresponds to induction.

As with product types before, the recursor is a special case of induction where the return type C is constant:

```
Σ-rec' : {A : U} {B : A → U} (C : U)
  → ((x : A) → B x → C) → (Σ \ (x : A) → B x) → C
Σ-rec' {A} {B} C = Σ-ind (λ _ → C)
```

Computation Rule for Product Types and Σ types

As we are now using induction principles to use elements of the product and Σ types, we do not need to specify additional computation rules.

Uniqueness Principle for Product Types and Σ types

Every element of a product type $A \times B$ is a pair. This can be proved by defining a dependent function on product types, as we previously mentioned.

```
-identity type
data _≡_ {A : U} : A → A → U where
  refl : {a : A} → a ≡ a

uppt : {A B : U} → (x : A × B) → ((pr₁ x , pr₂ x) ≡ x)
uppt (a , b) = refl
```

We use a new type, the identity type, to prove this. We represent this type as $\equiv$ to distunguish it from the $=$ used in function defintions. We need this type in order to represent equality. So *uppt* says that for all $x : A \times B$, x is equal to a pair composed of $pr_1\ x$ (which will return the first element) and $pr_2\ x$ (which will return the second element). This reduces to reflexivity (that $x \equiv x$). The identity type has one element, refl, representing reflexivity, so we can use this in the proof. We will consider the identity type in more detail later.

The propostional uniqueness principle for the unit type states that the only element is $\star$. We can also prove this using the identity type, as x can only be $\star$, so we have $\star \equiv \star$, which reduces to the reflexivity element, refl:

```
upun : (x : 𝟙) → x ≡ ⋆
upun ⋆ = refl
```

It can also be defined using induction.

```
𝟙-ind : (C : 𝟙 → U) → C ⋆ → (x : 𝟙) → C x
𝟙-ind C c ⋆ = c

upun' : (x : 𝟙) → x ≡ ⋆
upun' = 𝟙-ind (λ x → x ≡ ⋆) refl
```

As before we can see that any element, x, can only be defined as $\star$, so $x \equiv \star$ reduces to $\star \equiv \star$ which will be the element refl, representing reflexivity.

For now we do not propose a uniqueness principle for Σ types as we need path induction to prove it.

Examples of Σ-types

We can prove a type theoretic version of the axiom of choice in set theory, which says that the cartesian product of a collection of non-empty sets is non-empty.

The type theoretic version states that given $R : A \times B \to U$, which models a relation on pairs, if for all $x : A$ there is a $y : B$ such that $R(x\ ,\ y)$ then there is a function

$f : A \to B$ such that for all $x : A$ we have $R(x, f(x))$. Then ac will be $ac : (\Pi_{(x:A)}\Sigma_{(y:B)}R(x\ , y)) \to (\Sigma_{(f:A\to B)}\Pi_{(x:A)}R(x\ , f(x))$.

Then ac says if we have a function g assigning to every $x : A$ a dependent pair $(y\ , r)$ where $y : B$ and $r : R(x\ , y)$, then we can have a function $f : A \to B$ and a dependent function assigning to every $x : A$ a witness that $R(x\ , f\ x)$. This means we can split the proof into the two components. So in Agda we have:

```
ac : {A B : U} {R : A × B → U} →  ((x : A) → Σ \(y : B) → R (x , y))
   →  (Σ \(f : A → B) → (x : A) → R (x , f x))
ac g = (λ x → Σpr₁ (g x)) , (λ x → Σpr₂ (g x))
```

We can verify that this is well typed, as if $g : (x : A) \to \Sigma\ (y : B) \to R(x\ , y)$ we have $\lambda x.\Sigma pr_1\ (g(x))$ of type $A \to B$ and $\lambda x.pr_2\ (g(x))$ of type $\Pi_{(x:A)}R(x\ , \Sigma pr_1\ (g(x)))$:

```
a : {A B : U} {R : A × B → U} {g : (x : A) → Σ \(y : B) → R (x , y)}
   → A → B
a {A} {B} {R} {g} = λ x → Σpr₁ (g x)

b : {A B : U} {R : A × B → U} {g : (x : A) → Σ \(y : B) → R (x , y)}
   → (x : A) → R (x , Σpr₁ (g x))
b {A} {B} {R} {g} = λ x → Σpr₂ (g x)
```

2.5 Coproduct Types

Formation and Introduction of Coproduct Types

A coproduct type is the same as the disjoint union in set theory. This means that it combines all the elements of two types, so given types A and B, $A + B : U$ is the type of the disjoint union of A and B.

To construct elements of $+$, we use injections. For the type on the left hand side we use a left injection. Given an element $a : A$ we get an element $inl : A + B$. Then for the type on the right hand, $b : B$ we have the right injection defined as $inr : A + B$. We can then define this in Agda as follows:

```
{-# OPTIONS –without-K #-}

open import Product

data _+_ (A B : U) : U where
```

```
inl : A → A + B
inr : B → A + B
```

There is also a nullary type for coproducts as there was for products. This is called the empty type. As it is empty there are no constructors on it. This is done in Agda by having an empty data defintion:

```
data 𝕆 : U where
```

Elimination Rules for Coproduct Types

As with the product type, we now use recursors and induction for the elimination. First we define the recursor, so that we can use non-dependent functions out of coproducts:

To construct a function $f : A + B \to C$, we use two functions, $g_0 : A \to C$, for the left type and $g_1 : B \to$ C, for the right type. Then the recursor will use $g_0\ a$ or $g_1\ b$, depending on if it is given an $inl\ a$ or $inr\ b$. The return type C can be any type. Therefore it will be the case that $f(inl\ a) \equiv g_0(a)$ and $f(inl\ b) \equiv g_1(b)$. We can write this in Agda as:

```
+-rec : {A B C : U} → (A → C) → (B → C) → A + B → C
+-rec g₀ g₁ (inl a) = g₀ a
+-rec g₀ g₁ (inr b) = g₁ b
```

For the empty type we can always use the recursor, because as it has no elements anything can be constructed. This means we will have a function $f : \mathbb{0} \to C$ as defining an element of $\mathbb{0}$ is the same as defining a contradiction, so we can get the return type from a contradition by the law **ex falso quodlibet** in logic (that from a false conclusion you can derive anything). In Agda we write this using () to indicate that there is no case to match, as the type is empty:

```
𝕆-rec : {C : U} → 𝕆 → C
𝕆-rec ()
```

Then as before we can construct a dependent function using induction for coproduct types. Then to define a function $f : \Pi_{(x:A+B)}\ C\ (x)$, then the return will now be a family $C : A + B \to U.$ and g_0 and g_1 will be dependent functions, depending on an element $a : A$ and $b : B$ respectively. We define this in Agda as:

```
+-ind : {A B : U} {C : A + B → U} → ((a : A) → C (inl a))
  → ((b : B) → C (inr b)) → ((x : A + B) → C x)
```

```
+-ind g₀ g₁ (inl a) = g₀ a
+-ind g₀ g₁ (inr b) = g₁ b
```

Then when C is a constant family we can see that the recursor is just that case of induction:

```
+-rec' : {A B C : U} → (A → C) → (B → C) → A + B → C
+-rec' = +-ind
```

We can also define induction on the empty type, to define dependent functions out of it :

```
𝟘-ind : {C : 𝟘 → U} (z : 𝟘) → C z
𝟘-ind ()
```

2.6 Boolean Types

Formation and Construction Rule for Booleans

Boolean types are types that are either true or false. They don't depend on any other types so can be formed by defining 2 elements, 0_2 and 1_2.

```
{-# OPTIONS –without-K #-}

-imports used so we don't have to redefine types we want to use for examples
open import Product
open import Coproduct

-type of booleans
data 𝟚 : U where
  0₂ : 𝟚
  1₂ : 𝟚
```

Booleans can also be defined using a combination of the coproduct type and unit type, where the left injection can represent true and the right injection false (or vice versa). Then each type only needs one element as we only want to represent one thing in each injection. In Agda we can write:

```
bool : U
bool = 𝟙 + 𝟙
```

Induction and Recursor for Booleans

To define a function $f : 2 \to C$, we need to have a return value for each element $c_0, c_1 : C$ and the elements themselves. Then $f(0_2) \equiv c_0$ and $f(1_2) \equiv c_1$ define the function f.

Then the recursor for booleans will be $rec_2 : \Pi_{(C:U)}\ \mathrm{C} \to \mathrm{C} \to 2 \to \mathrm{C}$ defined as $rec_2(C, c_0, c_1, 0_2) \equiv c_0$ on the element 0_2 and $rec_2(C, c_0, c_1, 1_2) \equiv c_1$ on the element 1_2. We can write this in Agda as:

```
-recursor for booleans
2-rec : {C : U} → C → C → 2 → C
2-rec c₀ c₁ 0₂ = c₀
2-rec c₀ c₁ 1₂ = c₁
```

Then to define a dependent function out of a boolean type, $f : \Pi_{(x:2)}\ C\ (x)$, we now need the return types c_0 and c_1 to be dependent types, so $c_0 : C(0_2)$ and $c_1 : C(1_2)$. This makes the induction principle $ind_2 : \Pi_{(C:2\to U)}\ C(0_2) \to C(1_2) \to \Pi_{(x:2)}\ C\ (x)$. We can represent this in Agda as:

```
-induction for booleans
2-ind : {C : 2 → U} → C 0₂ → C 1₂ → ((x : 2) → C x)
2-ind c₀ c₁ 0₂ = c₀
2-ind c₀ c₁ 1₂ = c₁
```

Uniqueness Principle for Boolean Types

We can use the induction principle and identity type to prove that every element of the Boolean type is either 0_2 or 1_2. The proof is a function that uses the induction principle to assign a proof that for each element x is equal to either 0_2 or 1_2, so both of these cases reduce to the reflexivity element, refl. The return type C is the coproduct stating x is 0_2 or 1_2. c_0 uses inl, so we want to prove $x \equiv 0_2$, so when the induction is used on 0_2 this reduces to refl. c_1 uses inr, so we want to prove $x \equiv 1_2$, so when the induction is used on 1_2 this reduces to refl. Therefore the type reduced to refl in every case. We write this in Agda as follows:

```
upbt : (x : 2) → ((x ≡ 0₂) + (x ≡ 1₂))
upbt 0₂ = 2-ind {λ x → ((x ≡ 0₂) + (x ≡ 1₂))} (inl refl) (inr refl) 0₂
upbt 1₂ = 2-ind {λ x → (x ≡ 0₂) + (x ≡ 1₂)} (inl refl) (inr refl) 1₂
```

2.7 Natural Numbers

Formation and Construction Rules

The natural numbers are defined using zero and successor functions, so the number 0 is defined as $zero : \mathbb{N}$. Then the successor function is a unary function of type $\mathbb{N} \to \mathbb{N}$ that defines the successor of the number given to it. For example the number 4 would be defined as $succ\ (succ\ (succ\ (succ\ zero))) : \mathbb{N}$. We define this in Agda as:

```
{-# OPTIONS –without-K #-}

U = Set

data ℕ : U where
  zero : ℕ
  succ : ℕ → ℕ

four : ℕ
four = succ (succ (succ (succ zero)))
```

Induction and Recursion

We can use recursion and induction to elimate the type of natural numbers as before, but now its meaning is more obvious, because it is very similar to defining recursive functions and proof by induction on natural numbers in the way that we are used to.

To define a non-dependent function out of a natural number, $f : \mathbb{N} \to C$, we give a function for $zero$, which will include the base case: $c_0 : C$ and an inductive step function $c_s : \mathbb{N} \to C \to C$. Then we have:

```
f : {C : U} → C → (ℕ → C → C)  → ℕ → C
f c₀ cs zero = c₀
f c₀ cs (succ n) = cs n (f c₀ cs n)
```

We pass c_0 and cs as parameters so they do not have to be proved separately. Then applying a function to $zero$ returns c_0 and applying a function on $(succ\ n)$ applies the inductive step function on every $n-1$ until we get to the base case. This is exactly how we will define the recursor, so we can just replace f with $\mathbb{N}$-rec:

```
ℕ-rec : {C : U} → C → (ℕ → C → C) → ℕ → C
```

```
ℕ-rec {C} c₀ cs zero = c₀
ℕ-rec {C} c₀ cs (succ n) = cs n (ℕ-rec {C} c₀ cs n)
```

We can define a function to double numbers by recursion. If the number given is 0, we simply return 0. Otherwise for each $(n-1)$ we add 2 to the result until we get to the base case, which will add 0 to the entire result. We write this in Agda as:

```
double : ℕ → ℕ
double zero = zero
double (succ n) = succ (succ (double n))
```

Then using the recursor we can redefine it as:

```
double' : ℕ → ℕ
double' = ℕ-rec zero (λ n y → succ (succ y))
```

Here the parameters to the recursor are $c_0 = zero$, as double $zero$ is just $zero$ and cs is a function that takes a parameter n, which is the next $(n-1)$ value the recursor will use and y, the number passed to the function double.

We can also define non-dependent functions using currying (where a function with multiple arguments is converted to a function with just one argument). This can be done in type theory by letting the return type C be a function type, for example C could be $C : \mathbb{N} \to \mathbb{N}$. An example is this is defining the function add. Here for the base case, adding 0 to n is just n. In the inductive step, adding $(succ\ m)$ to n (i.e. $m+1+n$) is the same as the successor of m and n (i.e. $(m+n)+1$). We write this in Agda as:

```
add : ℕ → ℕ → ℕ
add zero n = n
add (succ m) n = succ (add m n)
```

Then we can also write this using the recursor for natural numbers:

```
add' : ℕ → ℕ → ℕ
add' = ℕ-rec (λ n → n) (λ n g m → succ (g m))
```

Here c_0 is now a function of type $\mathbb{N} \to \mathbb{N}$. In this case, c_0 is just the identity function, because adding 0 to n does nothing. Then cs is now of type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$. and is a function that takes a parameter n, the next $(n-1)$ value then the two arguments to add, g and m.

Then we can define dependent functions using induction. $f : \Pi_{(n:\mathbb{N})}\ C(n)$, will be constructed by giving a function for 0, which will be the base case: $c_0 : C\ (0)$ and an inductive step

function $c_s : \Pi_{(n:\mathbb{N})} C(n) \to C(succ\ n)$. Then we have:

```
f' : {C : ℕ → U} → C zero → ((n : ℕ ) → C n → C (succ n))
   → ((n : ℕ)  → C n)
f' {C} c₀ cs zero = c₀
f' {C} c₀ cs (succ n) = cs n (f' c₀ cs n)
```

We pass c_0 and cs as parameters so they do not have to be proved separately, as before. Then applying a dependent function to $zero$ returns c_0 and applying a function on ($succ$ n) applies the inductive step function on every $n-1$ until we get to the base case. This is exactly the same as the recursor, but now we are applying it to a dependent return type. Therefore this is exactly the same as the induction rule for natural numbers:

```
ℕ-ind : {C : ℕ → U} → C zero → ((n : ℕ) → C n → C (succ n))
   → ((n : ℕ) → C n)
ℕ-ind {C} c₀ cs zero = c₀
ℕ-ind {C} c₀ cs (succ n) = cs n (ℕ-ind {C} c₀ cs n)
```

In the introduction we mentioned that propositions can be represented as types. Now we can see that by defining a dependent function on the natural numbers we can define it on the base case and the inductive step, so this is exactly how we would usually do a proof by induction. As an example, we can prove that addition is associative in this way:

First we have to define a type for our proposition. For addition to be associative, it must be the case that given any i, j and k, $i + (j + k) = (i + j) + k$. The the type is just $(i, j, k : \mathbb{N}) \to add\ i\ (add\ j\ k) \equiv add\ (add\ i\ j)\ k$. We have defined addition using prefix notation so must present it accordingly. Then we solve this using induction, by first creating a function for the base case:

```
data _≡_ {A : U} : A → A → U where
refl : {a : A} → a ≡ a

add-assoc₀ : (j k : ℕ) →  add zero (add j k) ≡ add (add zero j) k
add-assoc₀ j k = refl
```

We use the identity type to show equality of the two functions. Then we do not need the parameter i because it is obviously just $zero$ in this case. Now we have$0+(j+k) = (0+j)+k$. This reduces to $j+k = j+k$ which are both the same, so it is easy to see that this function will always just return refl, the proof of reflexivity.

Then we can define the inductive step. If addition on i j and k is associative then it must be the case that addition the successors of i j and k will also be associative. In this sense the

associativity of adding i j and k represents the inductive hypothesis and we want to prove it for their succesors. Therefore we want to prove that if two natural numbers are equal then so are their successors. Obviously this should just reduce to refl when defined in Agda:

```
ap-succ : {m n : ℕ} → (m ≡ n) → (succ m ≡ succ n)
ap-succ refl = refl
```

Then we can apply this in our proof of the inductive step. This says that given a natural number i, we can prove that for any j and k, if their addition is associative, so is the addition of their successors. In Agda we write this as:

```
add-assocs : (i : ℕ)  → ((j k : ℕ) → add i (add j k) ≡ add (add i j) k)
                      → ((j k : ℕ ) → (add (succ i) (add j k) ≡ add (add (succ i) j) k))
add-assocs i h  j  k = ap-succ (h j k)
```

Here h is the proof that adding i j and k is associative. Then $ap - succ$ on this function of j and k will return a proof that addition is associative for the successors of j and k.

Now we can put this all together using the induction rule for natural numbers:

```
add-assoc : (i j k : ℕ) → add i (add j k) ≡ add (add i j) k
add-assoc = ℕ-ind add-assoc₀ add-assocs
```

2.8 Propositions as Types

Now we have defined everything necessary for Martin Löf Type Theory. This means that we can show exactly how propositions as types works. We can define propositions as types due to the **Curry Howard correspondence** (or isomorphism) defined in [Howard, 1980] and applied to MLTT in [Martin-Löf, 1998].

The exact version of the Curry Howard Isomorphism used is the BHK (Brouwer-Heyting-Kolmogorov) interpretation, as described in [Troelstra, 2011]. This is a correspondence between the rules of MLTT and the rules of intuitionistic logic, as demonstrated below:

Logic	Type Theory
True	$\mathbb{1}$
False	$\mathbb{0}$
$\neg A$	$A \to \mathbb{0}$
$A \Rightarrow B$	$A \to B$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$

True and False

True corresponds to the unit type. This is because there is only one element of $\mathbb{1}$ and that if we are just given true then it cannot be reduced to anything else. False is the empty type, $\mathbb{0}$ because to give an element of false would be the same as a contradiction and therefore we don't want to give any elements of this type.

Negation

The empty type $\mathbb{0}$ corresponds to false, but deriving an element of $\mathbb{0}$ is a contradiction. Therefore $\neg A$ is a function of type $A \to \mathbb{0}$. Given an element $x : A$, there is a mapping from it to the empty type, so we get an element of $\mathbb{0}$, which is a contradiction. Therefore $\neg A$ says we cannot have an element of A because having an element of A is a contradiction. To write this in Agda we write:

```
{-# OPTIONS –without-K #-}

open import Product
open import Coproduct

¬ : U → U
¬ A = A → 𝟘
```

This means we can use proof by contradiction in the case where we assume A is true and prove that it is not true by obtaining a contradiction. However we cannot do this in the other direction, double-negation elimination, that is, from $\neg\neg A$ obtain A. The reason is that to establish A we have to exhibit an element of A, but $\neg\neg A$ merely says that A is non-empty, or that it is impossible that A does not have any element, without claiming to exhibit an element of A. Hence our logic is intuitionistic, as it lacks double-negation elimination.

Implication

A function type corresponds to implication. This is because to prove an implication, we assume a proposition A, then prove B. When constructing function types we apply a value of type A to the function to get a value in type B dependent function. This means that we have proved A because we have an element of it and will then be able to prove B by providing an element of it.

Conjunction

Product types correspond to conjunction, because to form an element of the product type we need a proof of A and a proof of B, which is the same as giving an element of A and an element of B.

Disjunction

Coproduct types correspond to disjunction, because to define an element of a product type $A + B$, you must have an element of A or an element of B and to prove a disjunction you must prove either A or B.

As an example, we can give a proof for one of De Morgan's Laws, that $\neg A \wedge \neg B \Rightarrow \neg(A \vee B)$:

```
deMorgan : {A B : U} → (¬ A) × (¬ B) → ¬(A + B)
deMorgan {A} {B} (f , g) = h
where
  h : A + B → 𝟘
  h (inl a) = f a
  h (inr b) = g b
```

Therefore we reperesent the $\wedge$ as a product type and the $\vee$ as a coproduct. The $\neg$ we have previously defined as the type $A \to \mathbb{0}$. Then the propostion is defined as a type and we need to give an element of it. This type is a function type so we can give this as an implication, by assuming we have an element of $(\neg A) \times (\neg B)$ and getting an element of $\neg(A + B)$. By our definition of $\neg$, this is also a function, which we define as h. Then te value we pass to h is a coproduct, so we define it for if we have an element of A or if we have an element of B. If we have an element of a then we want the negation of a. This is the function f that we passed as an argument to our proof, so we can give $f\ a$. Then we do the same for b, in this case the function for its negation being g.

We can also define this function using the recursor for coproducts:

```
deMorganRec : {A B : U} → (¬ A) × (¬ B) → ¬(A + B)
deMorganRec {A} {B} (f , g) = +-rec f g
```

Here we pass the negation of A (f) and the negation of B (g) to the recursor instead of defining each function separately with pattern matching.

We could also write this out explicitly without using our type of $\neg$ to make it more clear:

```
-using pattern matching
pm-proof : {A B : U} → (A → 𝟘) × (B → 𝟘) → (A + B → 𝟘)
pm-proof (x , y) (inl a) = x a
pm-proof (x , y) (inr b) = y b

-using the recursor
rec-proof : {A B : U} → (A → 𝟘) × (B → 𝟘) → (A + B → 𝟘)
rec-proof (x , y) = +-rec x y
```

We can also write the converse of this law similarly:

```
converse-proof : {A B : U} → (A + B → 𝟘) → (A → 𝟘) × (B → 𝟘)
converse-proof {A} {B} f = (λ a → f (inl a)) , (λ b → f (inr b))
```

Now we are defining a function that returns an element of a product, given an element of a coproduct. Therefore we want the pair of $(\neg A\ ,\ \neg B)$. We get this by applying $\neg(A + B)$ to an the coproduct constructor for A and then doing the same for an element of B. We can define these functions using lambda abstraction.

We cannot prove the other de Morgan's law in intuitionistic logic, that $\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$. This is because there is a proof that A and B are not true at the same time, but not a proof that A is always false and B always false, so a type cannot be formed for it, because we do not know which of A and B are true:

```
-invalid :  {A B : U} → ((A × B) → 𝟘) → (A → 𝟘) + (B → 𝟘)
-invalid {A} {B} f = inl (λ a → f (a , _)) - the type B cannot be inferred
-invalid {A} {B} f = inr (λ b → f (_ , b)) - the type A cannot be inferred.
```

Obviously this wouldn't hold in Agda, so this code would not compile, but we can show what the type would look like if it were to be written in Agda.

Universal and Existential Quantifiers

We can also use dependent types to define predicate logic. This is because the proof now depends on extra information, given by the predicate, so the type of proofs will be $P : A \to U$. Then for every element $x : A$ we will have a proof of $P\ x$:

Logic	Type Theory
$\exists(x : A).P(x)$	$\Sigma_{(x:A)}P(x)$
$\forall(x : A).P(x)$	$\Pi_{(x:A)}.P(x)$

We use the dependent pair type for existential quantifiers because to prove there is an $x : A$ that satisifies the predicate, we we need to give an element of $P\ x$ for the predicate $(x : A)$ and we can do this using a dependent pair.

We use the dependent function type for universal quantifiers because it defines an element of the predicate for every element of A, and to prove a universal quantifier we need to prove it for all values of A.

As an example we can prove the statement $\forall(x : A)\ .\ P(x) \wedge Q(x) \Rightarrow (\forall(x : A).\ P(x)) \wedge (\forall(x : A)\ .\ Q(x))$ by defining an element of its type:

```
predicative-proof : {A : U} {P Q : A → U} → ((x : A) → P x × Q x)
  → ((x : A) → P x) ×((x : A) → Q x)
predicative-proof p = (λ x → pr₁ (p x)) , (λ x → pr₂ (p x))
```

We use dependent functions for this type as all of the predicates are universal quantifiers. This proof is an implication, so we are given an element of $((x : A) \to P\ x \times Q\ x)$ and can use this to prove (or define an element of) $((x : A) \to P\ x) \times ((x : A) \to Q\ x)$. This is a pair, because we using conjunction, so the result of our proof must be a pair. Then we can extract the proof of $((x : A) \to P\ x)$ from the proof given using the first projection and extract the proof for $((x : A) \to Q\ x)$ from the second projection.

Another example is defining inequalities on natural numbers. This can be done using dependent pairs, as we want to show that there exists a number that is less than or equal to another number, not that every number is less than or equal to some number:

```
data ℕ : U where
  zero : ℕ
  succ : ℕ → ℕ

add : ℕ → ℕ → ℕ
add zero n = n
add (succ m) n = succ (add m n)
```

We define these equalities using our type of natural numbers and the addition function that we previously defined. We say that a number $n \leq m$ if there exists another number k such that $n + k = m$. We write this in predicate logic as $\exists_{(k:\mathbb{N})}(n + k = m)$. Then we define this in Agda as:

```
_≤_ : ℕ → ℕ → U
n ≤ m = Σ (\ k → add n k ≡ m)
```

Then we can also define $n < m$ in the same way, but by saying that it must be the case that

$n + (succ\ k) = m$. In Agda this is:

```
_<_ : ℕ → ℕ → U
n < m = Σ (\k → add n (succ k) ≡ m)
```

As this definition is very similar, we can use $n \leq m$ to prove $n < m$.

```
_<'_ : ℕ → ℕ → U
n <' m = (n ≤ m) × ¬ (n ≡ m)
```

This says that it must be the case that $n \leq m$ and n is not equal to m.

2.9 Identity Types

For two types to be equal, they must satisfy **propositional equality**. This means that they must satisfy a proposition that two types are equal. As we can represent propositions as types, there will be a type to represent equality. It will be a proposition that two elements a and b, of an arbitary type A, are equal, so we will need to have a type dependent on two copies of A. In Agda we can define this as:

```
{-# OPTIONS –without-K #-}
U = Set

infixr 0 _≡_
data _≡_ {A : U} : A → A → U where
  refl : (x : A) → x ≡ x
```

where $\equiv$ is a data type that takes two values of a type A (in the universe U). Elements of the equality type are then constructed using reflexivity (denoted by refl here), so reflexivity is the formation rule for the identity type. It can be defined as a dependent function of type $(a : A) \to (a \equiv a)$, which means given any element $a : A$, then there will be an element of $a \equiv a$, so a must be equal to itself. The homotopical interpretation of reflexivity is that $(\mathrm{refl}\, a)$ is the constant path at a point a (a path from a to itself).

There can be more than one proof that two elements of the type A are equal to each other so there can be more than one element of $\equiv$. In homotopy theory, this corresponds to the idea of paths. If elements of the type A (so two points in the homotopy space) are equal to each other, then this is the same as saying that there is a path between them. There are many paths that can constructed between two objects, which corresponds to there being many different proofs of equality between two elements.

If there are two elements a and b are equal by $\equiv$ and are judgementally equal, then this reduces to refl, because $a \equiv b$ will also be judgementally equal to $a \equiv a$, so we get refl a. If the two elements are not judgementally equal, then we have to prove that they are propositionally equal using induction, which is the elimination rule for identity types.

Induction and Recursion on Identity Types

Before defining induction, we can define recursion on identity types using a property called **indiscernability of identicals**:

```
≡-Rec : (A : U) (C : A → A → U) → ((x : A) → C x x)
      → (x y : A) → x ≡ y → C x y
≡-Rec A C c = f
  where
    f : (x y : A) → x ≡ y → C x y
    f x .x (refl .x) = c x
```

This states that every family of types, $C : A \to A \to U$, respects equality, so applying C to equal elements of A will give a function between the resulting types. More formally, we use the recursor to define a function $f : (x\ y : A) \to (p : x \equiv y) \to C\ x\ y$ such that $f\ (x, x,$ refl $x) = c\ x$. As refl is the only element of the identity type, c is the function defined for refl x and therefore the entire identity type, so it is the identity function. This states that all $x : A$ are equal to the x given, so we have a proof that $x \equiv x$. Then we can use this to define the function f for any two elements of A. f states that any proof of $x \equiv y$ can be reduced to reflexivity.

The recursion here is defined in a similar way to recursion for other types, such as boolean values and natural numbers. In the recursion for $\mathbb{N}$, there is a map from $\mathbb{N} \to C$ for any other type $C : U$, so we can define the function by giving a definition in the case of zero and for any successor of a number in $\mathbb{N}$. In the recursion for identity types, this map is replaced with a map from $x \equiv y$ to other types of the form $C\ x\ y$ and giving a defintion for refl allows us to make this map:

```
data ℕ : U where
zero : ℕ
succ : ℕ → ℕ

ℕ-rec : (A : U) → A → (A → A) → (ℕ → A)
ℕ-rec C zero' succ' = h
where
  h : ℕ → C
```

```
h zero = zero'
h (succ n) = succ'(h n)
```

Path Induction

Induction on identity types is also known as path induction, due to the homotopical interpretation of equalities as paths. It is also sometimes referred to as J. The family of equality types is generated from refl (similar to how a property on natural numbers can be generated from *zero* and the successor of a number n). It is similar to the recursor, but now the return type has an additional parameter, which is the evidence that two types are equal, represented by p. It is defined in Agda as:

```
≡-Ind : (A : U) (C : (x y : A) → x ≡ y → U)
      → ((x : A) → C x x (refl x))
      → (x y : A) → (p : x ≡ y) → C x y p
≡-Ind A C c = f
where
  f : (x y : A) (p : x ≡ y) → C x y p
  f x .x (refl .x) = c x
```

This means that given a family of proofs of the equality of two elements of A , $C : (x\ y : A) \to x \equiv y \to U$ and a function $c : ((x : A) \to C\ x\ x\ (\text{refl}\ x))$, there is a function $f : (x\ y : A) \to (p : x \equiv y) \to C\ x\ y\ p$, such that $f\ x\ x\ (\text{refl}\ x) = c\ x$.

Less formally, the type C is a predicate we want to satisfy. It takes two elements, x and y and a property p. By proving $C\ x\ y$, we are proving that the property holds on x and y. The inductive hypothesis of path induction will then be an element of $C\ x\ x$, the proof that C holds for x and x. The inductive hypothesis says that $C\ x\ x$ holds for all $x : A$ by reflexivity. Then we know that $C\ x\ y$ holds whenever x is equal to y. This is done using the parameter p, which is a witness that $x \equiv y$. Knowing this, we can replace $x\ y$ by $x\ x$ and replace p with reflexivity.

Therefore to prove a property for all elements $x\ y : A$ and paths/equalities $p : x \equiv y$, then we just consider the cases where the elements are x, x, and $(\text{refl}\ x)$.

Then we can show that indiscernability of identicals follows from induction:

```
≡-Rec' : (A : U) (C : A → A → U) → ((x : A) → C x x)
  → (x y : A) → x ≡ y → C x y
≡-Rec' A C c = ≡-Ind A (λ x y _ → C x y) c
```

When we use path induction, with a proof , $p : x \equiv y$, we replace both x and y with

(the same) unknown element x, so to define an element of C, then we can define it on the diagonal, for all pairs of elements of A.

However, sometimes it is simpler to just replace all occurences of y with x, or vice versa, as then the rest of the proof can be completed on just x (or just y) instead of the general unknown element x'. Then we can use another induction principle to do this, called **based path induction**.

Based Path Induction

Based path induction states that a family of types $a \equiv x$ (or dependent function $(x : A) \to a \equiv x \to U$) where $x : A$ is generated by $(\text{refl}\, a)$. We can define it directly by pattern matching in Agda:

```
based-path-induction : {A : U} (a : A) (C : (x : A) → a ≡ x → U) (c : C a (refl a))
  → (x : A) (p : a ≡ x) → C x p
based-path-induction a C c .a (refl .a) = c
```

This is defined in a similar way to the induction but we now have an additional parameter $a : A$, which is the point that we want to prove is equal to any $x : A$ and then we can give the proof for it using refl a instead of refl x.

It is also definable from path induction, but we define it directly by pattern matching as otherwise we would be required to use path induction at higher universes. Another way of defining it is by using singleton types, as in [Coquand, 2014]. By representing the input type as a singleton type, we can prove that there is only one element $a : A$, by proving that any x in A is equal to x and therefore there is no other x in A, only a.

Based path induction using singleton types

Singl is our type of singleton types (types that only contain one element). It can be read as a proposititon that says for every $(x : A)$, there exists an element y that is equal to x. Therefore A can only contain one element. In Agda we write this as:

```
data Σ {A : U} (B : A → U) : U where
_,_ : (a : A) → (b : B a) → Σ B

singl : {A : U} (x : A) → U
singl x = Σ \y → x ≡ y
```

Then, using this type we can show that, all elements y and p (a proof that $x \equiv y$), are equal to x and (refl x) respectively:

```
sigl-lemma : {A : U} (x y : A) (p : x ≡ y) → (x , refl x) ≡ (y , p)
sigl-lemma {A} = ≡-Ind A D (λ x → refl (x , refl x))
where
  D : (x y : A) → x ≡ y → U
  D x y r = _≡_ {singl x} (x , refl x) (y , r)
```

This can then be used to prove that based path induction is equivalent to path induction as in [Coquand, 2014]. We do this by using induction to get the return type D, applying a function which when defined for reflexivity, gives a proof of the return type. The function in the base case is ($\lambda\ x \to$ refl (x , refl x)), which says given any $x : A$ we can construct the element refl (x , refl x). This reduces to (x , refl x) = (x , refl x). This covers the base case of our induction because (y , p) will just be (x , refl x), so we simply want to prove refl (x , refl x).

Then for the inductive step we need to define a function type that will be the proof for any (y , p), which we do by defining the return type D. This type is a proposition that states $\forall$ $(x, y : A)$. $x \equiv y \to U$, so for any x and y in A there must be a proof that they are equal. We define a proof for this by using our singleton type we just defined. D is then defined using the definition for the equality type and our proof for *singl*.

Then by (x , refl x) = (y, p), we are saying is that they must both be proofs of *singl* (whereas before they were both elements of an arbitary type A) that are equal to each other. Then we can see that (x , refl x) is an element of *singl* because if an x exists, then $x = x$ holds by reflexivity, so it is the case that x is the only element of A. For (y . p), if an element y exists, then $x = y$ will hold by r, as r is a parameter to the function that contains a proof that $x = y$. Therefore y is equal to every x in A, so it it must be the only element of A. Then by the definition of the equality type, they will be equal, so we have a proof for D and we have proved $sigl - lemma$ for all cases.

Before we define based path induction we must also define **transport**. This is a property where if there are two elements x and $y : A$ that are equal to each other, then we can have a function from $B\ x \to B\ y$, where $B : A \to U$. We define this using the recursor for identity type. In this case of the recursor, our return type is a function $C\ x\ y = B\ x \to B\ y$ and we can provide a function f that gives us $B\ x \to B\ x$, because we already have an element $b : B\ x$ as an argument to the function. Then applying the recursor to C and f will give us $B\ x \to B\ y$ for any x and y:

```
transport : {A : U} (B : A → U) (x y : A) → x ≡ y → B x → B y
```

```
transport {A} B = ≡-Rec A C f
where
  C : A → A → U
  C x y = B x → B y
  f : (x : A) → C x x
  f x b = b
```

Now we have everything we need to define based path induction. The type of $\equiv -Ind'$ says that given $a : A$ (a point a in path space A), we want the return type that for any $x : A$, we can give a proof a is equal to x. Then the parameter c is a proof that $a \equiv (\text{refl } a)$, so it is the base case. Defining a function of the return type $C\ x\ p$ gives us a proof that for any point $x : A$, it is the case that $a \equiv x$, so we can return this for all elements including refl a.

We use our transport function on the singleton types that contain the point a (so the type $singl\ a$. Here, x is replaced by $(a, \text{refl } x)$ and y is replaced by (x, p). Our proof that $x \equiv y$ is the lemma $sigl - lemma$ we previously proved. c is an element of $B(a\ ,\ \text{refl } a)$, which we can use to obtain an element of $B(y\ ,\ p)$.

The return type B is a placeholder for the return type of the induction, $C\ x\ y\ p$, so given a singleton type representing the points in A, we can use the point and proof to construct the return type:

```
≡-Ind' : {A : U} (a : A) (C : (x : A) → a ≡ x → U) (c : C a (refl a))
      → (x : A) (p : a ≡ x) → C x p
≡-Ind' {A} a C c x p = transport {singl a} B (a , refl a) (x , p) (sigl-lemma a x p) c
where
  B : singl a → U
  B (x , p) = C x p
```

Now we have everything we need in MLTT to define Homotopy Type Theory.

Chapter 3

Homotopy Type Theory

As previously discussed, Homotopy Type Theory (HoTT) is based on homotopy theory. Traditionally homotopy theory is defined on points and paths between these points, but in HoTT these are primitive notions. This is the same as the distinction between analytic geometry, where lines are sets of points (for example using coordinates) and synthetic geometry, where points and lines are primitive notions and there are axioms (rules) and constructions are built on them, as discussed in [Univalent Foundations Program, 2013].

In HoTT the primitive notions are the types we have defined in MLTT that can be used to represent homotopical structures and the axioms are the type rules we have defined on these types. Then we can use the induction rule on identity types to construct things in HoTT, as it is now considered an induction rule on paths.

A **homotopy** is defined as in [Univalent Foundations Program, 2013] as:

Definition 1. *Homotopy A homotopy between a pair of continuous maps* $f : X \to Y$ *and* $g : X \to Y$ *is a continuous map* $H : X \times [0,1] \to Y$ *such that* $H\ (x\ ,0) = f(x)$ *and* $H\ (x\ , 1) = g(x)$.

Then if paths p *and* $q : x \equiv y$ *are given, a homotopy is a continous map* $H : [0,1] \times [0,1] \to Y$ *such that* $H\ (s\ , 0) = p(s)$ *and* $H\ (s\ , 1) = q(s)$ *for all* $s \in [0,1]$. *Then it must be the case that* $H\ (0\ ,t) = x$ *and* $H\ (1\ , t) = y$ *for all* $t \in [0,1]$ *, so that the destination of the path corresponds to the result of the map given to the homotopy. This is also caled a endpoint-preserving homotopy.*

Having a homotopy between paths allows us to specify paths between paths, or a 2-dimensional path. In type theory this corresponds to having a type of equalities of equalities. Then you could also have homotopies between homotopies and a path between this and keep on doing

this (defining paths between paths between ...) forever.

Having an infinite structure of points and paths like this corresponds to a structure in algebra called a **weak ω groupoid**.

A **groupoid**, as defined in [Burris and Sankappanavar, 2012], is a group that has a partial function instead of a binary function. A group conatains a set, G and a binary operation $\bullet$ defined on that set that satifies associativity and has an inverse operation, $^{-1}$ and an identity element 1. Then the set G will represent a path space, the partial operation $\bullet$ will be concatenation of paths and then we can prove that this is associative. The inverse operation, $^{-1}$ will be the inverse of a path p, p^{-1} and the identity element, 1 will be refl. As paths are represented by equality, then there is a correspondence between properties of each as defined in [Univalent Foundations Program, 2013]:

Equality	Homotopy	ω-groupoid
reflexivity	constant path	identity element (1)
symmetry	inversion of paths	inverse operator ($^{-1}$)
transitivity	concatenation of paths	associativity of the operation $\bullet$

Categorically a groupoid is defined as a category where all the morphisms are isomorphisms. It contains a set of objects G, a set of morphisms between the objects of G (that correspond to the operation $\bullet$ in the algebraic definition), an identity object corresponding to 1 and functions for inverse and concatenation. Then we can have morphisms between morphisms, and morphisms between these and so on, as we would with paths. Each higher dimension of path (morphism) added gives a higher dimension to the groupoid, as defined in [Univalent Foundations Program, 2013] a morphism at a level k is a k-**morphism**. For example if we just had morphisms between morphisms, this would be a 2-morphism and we would have a 2-groupoid.

In homotopy theory we can keep adding paths between paths forever, therefore we will have a weak ∞ groupoid. This is another name for the weak ω-groupoid. The groupoid is called weak because all the morphisms have identity, inverse and concatenation only up to the next homotopy level. This creates interactions between the levels that can be used to construct homotopical structures.

3.1 Properties of Equality

To show that types can be represented as weak ω groupoids, we must show that we can have associativity, inverse and identity for them. We can do this by proving that we have reflexivity, symmetry and transitivity on the equalities that represent paths:

Reflexivity

Reflexivity is easy to represent in HoTT, as a constant path is just a path from a point to itself, represented by the element refl, so in Agda a proof that any $x : A$ gives us the type $x \equiv x$ is just refl:

```
reflexivity : {A : U} (x : A) → x ≡ x
reflexivity = λ x → refl x
```

Symmetry

Symmetry of equalities in homotopy theory is the reversal of paths, so for any path $x \equiv y$, there is a path back from y to x $(y \equiv x)$. We can define symmetry type theoretically as $\Pi_{(A:U)}\Pi_{(x,y:A)}\ (x \equiv y) \to (y \equiv x)$. Then for every type A and every $x\ y : A$, there will be a function $(x \equiv y) \to (y \equiv x)$ such that $y \equiv x$ for each $x\ y : A$.

To prove symmetry we need to construct an element, $p^{-1} : y \equiv x$, of this type for each x $y : A$. We can use induction as x and y are the same value and therefore p reduces to refl x. In the reflexivity case, the proof is that we will have $x \equiv x$, which is the same as (refl x). Then the other cases all reduce to this case.

This is done in Agda by defining the function $D : \Pi_{(x,y:A)} \to (x \equiv y) \to U$ which assigns the type $y \equiv x$ to any $x\ y$ and $p : x \equiv y$. Then there will be an element where p is (refl x) and both the values x and y are x. This is d in the proof. Then all the other elements of D are generated from d:

```
symInd : (A : U) (x y : A) → x ≡ y → y ≡ x
symInd A = ≡-Ind A D d
  where
    D : (x y : A) → (x ≡ y) → U
    D x y _ = y ≡ x
    d : (x : A) → D x x (refl x)
    d = λ x → refl x
```

As the evidence p is not used at any point in the induction, we can also define symmetry just by using the recursor for identity types as the recursor does not require a proof that $x \equiv y$:

```
sym : (A : U) (x y : A) → x ≡ y → y ≡ x
sym A = ≡-Rec A D refl
where
```

```
D : A → A → U
D x y = y ≡ x
```

Symmetry of two paths is equivalent to the inverse of a path, so we can define the inverse of any path as, p^{-1}, using implicit arguments in Agda :

```
_⁻¹ : {A : U} {x y : A} → x ≡ y → y ≡ x
p ⁻¹ = sym _ _ _ p
```

We can also claim separately that $(\text{refl}\ x)^{-1} \equiv (\text{refl}\ x)$ for each $x : A$. As $(\text{refl}\ x) = x \equiv x$ and it should be equal to $(\text{refl}\ x)^{-1}$, this is the same as saying $(x \equiv x) \equiv (x \equiv x)$. This is the same as refl (refl x). In Agda we can define the type $\Pi_{(A:U)}\Pi_{(x:A)} \to (\text{refl}\ x)^{-1} \equiv \text{refl}\ x$. Then the element of this type will be refl (refl x), the proof that $(\text{refl}\ x)^{-1} \equiv (\text{refl}\ x)$:

```
sym-claim : (A : U) (x : A) → (refl x)⁻¹ ≡ refl x
sym-claim A x = refl(refl x)
```

Transitivity

Concatenation of paths is the same as transitivity for equality. For every type A and every x y $z : A$, there is a function $p \bullet q$ of type $(x \equiv y) \to (y \equiv z) \to (x \equiv z)$, such that $(\text{refl}\ x) \bullet (\text{refl}\ x) \equiv \text{refl}\ x$ for any $x : A$. Formally we define the type $\Pi_{(xyz:A)}(x \equiv y) \to (y \equiv z) \to (x \equiv z)$

To prove concatenation we need to construct an element of $x \equiv z$, for each x y $z : A$ and each $p : x \equiv y$ and $q : y \equiv z$. We can use induction on p, from the case where x and y are the same value and p is refl x. In the reflexivity case, q will now be $x \equiv z$. Now we have $(x \equiv x) \bullet (x \equiv z)$. Then we can use induction again on q to get reflexivity. Then we assume x is equal to z, then $q : x \equiv x$ and will be refl. Then we finally have $(x \equiv x) \bullet (x \equiv x)$ which reduces to (refl x). Then we can generate all other elements of the type $\Pi_{(xyz:A)}$ $(x \equiv y) \to (y \equiv z) \to (x \equiv z)$ from this case.

This is done in Agda by defining the first induction on p as a function $D : \Pi_{(x,y:A)} \to (x \equiv y) \to U$ which assigns the type $x \equiv z$ to any x and $y : A$ and $p : x \equiv y$. The result of this is the type family $\Pi_{(z:A)}\Pi_{(q:y\equiv z)}$ $(x \equiv z)$. Then c is an element of this type where x and z are equal to an element x and q is refl x. Then the result of this is refl x. So c is the reflexivity case for the first induction.

Then the second induction on q is defined by the function $E : \Pi_{(x,z:A)}\Pi_{(q:x\equiv z)} \to U$ .Then the reflexivity case for this induction is defined as E x x (refl x) which is equal to refl x. Using the induction principle we get the function $\Pi_{(x,z:A)}\Pi_{(q:x\equiv z)}$ E x z q. As the type of this is $x \equiv z$ we then get the function d, that we need for the induction on type D.

```
transInd : (A : U) (x y z : A) → x ≡ y → y ≡ z → x ≡ z
transInd A x y z p q = f x y p z q
where
  D : (x y : A) → x ≡ y → U
  D x y p = (z : A) → (q : y ≡ z) → x ≡ z
  c : (x : A) → D x x (refl x)
  c x z q = q
  E : (x z : A) (q : x ≡ z) → U
  E x z q = x ≡ z
  e : (x : A) → E x x (refl x)
  e x = refl x
  d : (x z : A) (q : x ≡ z) → E x z q
  d = ≡-Ind A E e
  f : (x y : A) (p : x ≡ y) → D x y p
  f = ≡-Ind A D d
```

We can also prove this using the recursor for the identity type. This is simpler than using induction, because by using the recursor on p we get $x \equiv z$, which is an element of q:

```
trans : (A : U) (x y z : A) → x ≡ y → y ≡ z → x ≡ z
trans A x y z p q = trans' x y p z q
where
  trans' : (x y : A) → x ≡ y → (z : A) → y ≡ z → x ≡ z
  trans' = ≡-Rec A D c
    where
      D : (x y : A) → U
      D x y = (z : A) → y ≡ z → x ≡ z
      c : (x : A) → D x x
      c x z q = q
```

However, when we want to have proof relevance, proving this way only gives a proof on p and we may have a completely different proof of q, so using induction means we have the same proof for p and q. In the case of transitivity, we can show that these proofs are equivalent, so it is ok to just use the recursor:

```
both-trans-equal' : (A : U) (x y z : A) (p : x ≡ y) (q : y ≡ z)
          → transInd A x y z p q ≡ trans A x y z p q
both-trans-equal' A = III
where
  I : (y z : A)(q : y ≡ z) → transInd A y y z (refl y) q ≡ trans A y y z (refl y) q
  I = ≡-Ind A C r
```

```
    where
      C : (y z : A) → y ≡ z → U
      C y z q = transInd A y y z (refl y) q ≡ trans A y y z (refl y) q
      r : ((y : A) → C y y (refl y))
      r y =  refl (refl y)

II : (x y : A)(p : x ≡ y)(z : A)(q : y ≡ z) → transInd A x y z p q ≡ trans A x y z p q
II = ≡-Ind A C r
    where
      C : (x y : A) → x ≡ y → U
      C x y p = (z : A)(q : y ≡ z) → transInd A x y z p q ≡ trans A x y z p q
      r : (x : A) → C x x (refl x)
      r = I

III : (x y z : A)(p : x ≡ y)(q : y ≡ z) → transInd A x y z p q ≡ trans A x y z p q
III x y z p = II x y p z
```

To say that each proof is equal, we construct an element of the identity type for each of them. Then to prove this we use induction. First we use induction on p, which is represnted in the code by II. The return type of the induction C states that given x and y and a proof they are equal, p, we will have a proof that both of our proofs of transitivity are equal. Our proof of this in the reflexivity case is represented by I

To prove our statement in the reflexivity case, we need to use induction on q, to show that q also reduces to refl. This means the return type of the induction will be given y and z and a proof they are equal p, we can give a proof that our transitivty proofs are equal. Then r is the reflexivity case. Given y and $y = y$, then $C\ y\ y$ refl y will be $transInd\ A\ y\ y\ y$ (refl y) (refl y) = $trans\ A\ y\ y\ y$ (refl y) (refl y), which will reduce to refl (refl y).

As transitivty of paths is equivalent to the concatenation of paths, we can define the concatenation of any two paths p and q as, $p \bullet q$, using implicit arguments in Agda :

```
_•_ : {A : U} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
p • q = trans _ _ _ _ p q
```

Then we can claim separately that the concatenation of a constant path with itself is the same as the constant path, or equivalently, that $x = x$ and $x = x$ is equivalently to just saying $x = x$:

```
trans-claim : (A : U) (x : A) → trans A x x x (refl x) (refl x) ≡ refl x
trans-claim A x = refl (refl x)
```

More lemmas for equality

Now we have to show that our operations on equality are well-behaved, because they define the structure of paths as well as properties of equality. Also for HoTT, in order to represent the higher groupoid structure, we need to show some other properties that hold for groupoids. These are associativity, inverse and identity and are given in the Lemma 2.1.4 in [Univalent Foundations Program, 2013], which we can represent in Agda:

The first lemma is used to show the **identity**. The first part states that $p \bullet \text{refl}\ x = p$. This means that concatenating a path $p : x \equiv y$ with the constant path at x will not make any difference to the path p, so it is a left unit law for concatenation. We can prove this using induction. Our return type will be a function where given any x and y and proof that they are equal, p, we can construct an element of $p \equiv (p \bullet (\text{refl}\ y)$. Our proof for refl will then be the proof that concatenating constant paths gives us the constant path from before:

```
Lemma-2-1-4[i] : (A : U) (x y : A) (p : x ≡ y) → p ≡ (p • (refl y))
Lemma-2-1-4[i] A = ≡-Ind A (λ _ y p →  p ≡ (p • (refl y))) (trans-claim A)
```

Then we can do the same for the right unit law, this time using refl x instead of y:

```
Lemma-2-1-4[i'] : (A : U) (x y : A) (p : x ≡ y) → p ≡ ((refl x) • p)
Lemma-2-1-4[i'] A = ≡-Ind A (λ x _ p →  p ≡ ((refl x) • p)) (trans-claim A)
```

Then the second lemma shows that **inverse** is always defined. It states that the concatenation of a path and its inverse will be refl and that this is commutative. In the first case, we will have $p : x \equiv y$ and $p^{-1} : y \equiv x$ so by transitvity we will get $y \equiv y$, which reduces to refl y. In Agda we can prove this by induction as we did for $Lemma 2-1-4-[i]$.

```
Lemma-2-1-4-[ii] : (A : U) (x y : A) (p : x ≡ y) → (p ⁻¹ • p) ≡ (refl y)
Lemma-2-1-4-[ii] A = ≡-Ind A (λ _ y p → ((p ⁻¹ • p) ≡ refl y)) (trans-claim A)
```

In the second part we concatenate $p^{-1} : y \equiv x$ with $p : x \equiv y$, giving us $x \equiv x$, which will reduce to refl x. Then we prove this in Agda, in the same way as the first part:

```
Lemma-2-1-4-[ii'] : (A : U) (x y : A) (p : x ≡ y) → (p • p ⁻¹) ≡ (refl x)
Lemma-2-1-4-[ii'] A = ≡-Ind A (λ x _ p → (p • (p ⁻¹)) ≡ refl x) (trans-claim A)
```

One thing to note with this statement is that although it reduces to refl in Agda, strictly in a homotopical interpretation it will not be the constant path, but there will be a homotopy from $p \bullet q$ to the constant path. This is because homotopies are equalities of equalities, so if two proofs p and q are equal, they will be equal up to the next homotopy. Then $p \equiv q$ is of type $p \equiv_{x \equiv y} q$.

The third lemma is a property that follows from the inverse and identity. It states that the inverse of the inverse of a proof p will be p. We can see this is true as $p^{-1} : y \equiv x$ so $(p^{-1})^{-1})$ will just be $x \equiv y$ which is p. Then when p is refl x, $(p^{-1})^{-1}$ will reduce to refl(refl x). We use induction again with our proof, as the induction reduces to refl (refl x), which is an element of $trans - claim$.

```
Lemma-2-1-4-[iii] : (A : U)(x y : A) (p : x ≡ y) → ((p ⁻¹) ⁻¹) ≡ p
Lemma-2-1-4-[iii] A = ≡-Ind A (λ x y p  → ((p ⁻¹) ⁻¹) ≡ p) (trans-claim A)
```

Finally, we must know that concatenation is **associative**. As we have 3 parameters now, we could use induction more than once, but using it once is enough to prove the lemma. We define a function D where given x and y and a proof p that $x \equiv y$ we can use this to get the return type. Then D is defined as $(p \bullet (q \bullet r)) \equiv ((p \bullet q) \bullet r)$, given a $z : A$ and a proof $q : y \equiv z$ and $r : z \equiv w$. Then the reflexivity case will just be refl x as every parameter given is x, so we have (refl $x \bullet$ (refl $x \bullet$ refl x)) $\equiv$ ((refl $x \bullet$ refl x) $\bullet$ refl x), which reduces to refl x.

Then if we want to use induction on q and r, the return type of the induction on q is E. For E we must give x and z and a proof that $x \equiv z$. Then we want to return (refl $x \bullet$ (refl $x \bullet r$) $\equiv$ ((refl $x \bullet$ refl x) $\bullet r$), as we have now shown that q reduces to refl. The reflexivity case here will be refl (refl x) as we assume that q is equal to refl x.

The induction on r is represented by $f - ind$ and the return type is F, where we must give x and w and a proof that $x \equiv w$. Then we want to return (refl $x \bullet$ (refl $x \bullet$ refl x)) $\equiv$ ((refl $x \bullet$ refl x) $\bullet$ refl x), as we have now shown that r reduces to refl. The reflexivity case here will be refl (refl x) as we assume that r is equal to refl x.

Then we specify the induction on D using the induction on E as its reflexivity case, because in E we have already proved that p is refl x. Then for induction on E give induction on F as the reflexivity case as we already know that q is refl x:

```
Lemma-2-1-4-[iv] : (A : U) (x y z w : A) (p : x ≡ y) → (q : y ≡ z) → (r : z ≡ w)
                                → (p • (q • r)) ≡ ((p • q) • r)
Lemma-2-1-4-[iv] A x y z w p q r = d-ind x y p z q r
  where
    D : (x y : A) → x ≡ y → U
    D x y p = (z : A) → (q : y ≡ z) → (r : z ≡ w) → (p • (q • r)) ≡ (p • q) • r
    d : (x : A) → D x x (refl x)
    d x .x (refl .x) = refl
    E : (x z : A) → (q : x ≡ z) → U
    E x z q = (w : A) → (r : x ≡ w) → refl x • (refl x • r) ≡ (refl x • refl x) • r
    e : (x : A) → E x x (refl x)
    e x .x (refl .x) = refl (refl x)
```

```
F : (x w : A) → (r : x ≡ w) → U
F x w r = refl x • (refl x • r) ≡ (refl x • refl x) • r
f : (x : A) → F x x (refl x)
f x = refl (refl x)
f-ind :  (x w : A) → (r : x ≡ w) → F x w r
f-ind = ≡-Ind A F f
e-ind :  (x z : A) → (q : x ≡ z) → E x z q
e-ind = ≡-Ind A E f-ind
d-ind :  (x y : A) → (p : x ≡ y) → D x y p
d-ind  = ≡-Ind A D d
```

In order to specify a higher groupoid structure, the properties of a groupoid must satisfy **coherence laws**, which are higher paths that go up to infinity (giving us the -groupoid). We use induction to define paths up to n dimensions.

The simplest example of this a **loop space**.

Loop Spaces

A loop is a path $a \equiv a$ from a to itself. It is different from a constant path, which is just a point $a : A$. Given a path space (type) A with a point $a : A$, its **loop space** is defined in Agda as Ω A a to be the type a ≡ a :

```
{-# OPTIONS –without-K #-}

open import Induction

Ω : (A : U) → A → U
Ω A a = a ≡ a
```

We can also have loop spaces of loop spaces, in the same way we have higher dimensional groups. For example, the loop space Ω^2 is also written as $\Omega^2(A, a)$, where a is a point so that the loop space is the space of 2 dimensional loops on the identity loop at a. It is represented type theoretically by the type $\text{refl}_a =_{(a=_a a)} \text{refl}_a$.

```
Ω² : (A : U) → A → U
Ω² A a = Ω (a ≡ a) (refl a)
```

Higher path spaces such as higher loop spaces have additional properties to those of path spaces. We can prove that Ω^2 is commutative using a homotopical version of the Eckmann Hilton Theorem:

3.2 The Eckmann Hilton Theorem

Set theoretic proof

Our first proof of the Eckmann Hilton principle is based on the set theoretic version of it:
Eckmann Hilton 1. *Let X be a set equipped with two binary operations, which we will write as . and * and suppose:*

1. ** and . are unital (identity elements*
2. *For all a, b, c, d ∈ X, (a * d) . (c * d) = (a . c) * (b . d) Then * and . are the same and are fact commutative and associative.*

We use a slightly different notation for this proof, in order to present the proof as a sequence of equal statements, so each statement gives the next one. We implement this in Agda as follows:

```
infix  0 finally
infixr 0 _≡⟨_⟩_

_≡⟨_⟩_ : {A : Set} (x : A) {y z : A} → x ≡ y → y ≡ z → x ≡ z
_ ≡⟨ p ⟩ q = p • q

finally : {A : Set} (x y : A) → x ≡ y → x ≡ y
finally _ _ p = p

syntax finally x y p = x ≡⟨ p ⟩■ y ■
```

The notation $_{\equiv}\langle\ p\ \rangle q$ states that given a proof $p : x \equiv y$ then we can feed this into a proof $q : y \equiv z$ that follows from p to prove the overall proposition that $x \equiv z$. When there is nothing left to prove we use $finally$ to show that from our overall proof $x \equiv y$, we have now obtained an element of y equal to x.

First we define ap, which we will use in our proof. ap states that given two equal elements x and $x' : X$ and a function $f : X \to Y$, then $fx \equiv fy$. We can give a proof that $x \equiv x'$ that reduces to refl. Then the proof of $fx \equiv fx'$ will just be refl:

```
ap : {X Y : Set} → (f : X → Y) → {x x' : X} → x ≡ x' → f x ≡ f x'
ap {X} {Y} f (refl _) = refl _
```

Then our proof of the Eckmann Hilton theorem is constructed using a module in Agda. The

module keeps track of defintions that can then be used in other modules. For a large proof such as this it is useful to keep track of the assumptions necessary to complete the proof:

```
module Eckmann-Hilton-Argument
(X : U)
(1× 1* : X)
(_×_ _*_ : X → X → X)
(right-unit-law-for-× : (x : X) →  x × 1× ≡ x)
(left-unit-law-for-×          : (x : X) → 1× × x ≡ x)
(right-unit-law-for-* : (x : X) → x * 1* ≡ x)
(left-unit-law-for-*          : (x : X) → 1* * x ≡ x)
(transposition : (x y z w : X) → (x * y) × (z * w) ≡ (x × z) * (y × w))
where
  units-coincide : 1× ≡ 1*
  units-coincide =
    1×                ≡⟨ (left-unit-law-for-× 1×)⁻¹ ⟩
    1× × 1×           ≡⟨ ap (λ x → x × 1×) eq1 ⟩
    (1* * 1×) × 1×    ≡⟨ ap (λ x → (1* * 1×) × x) eq2 ⟩
    (1* * 1×) × (1× * 1*) ≡⟨ transposition 1* 1× 1× 1* ⟩
    (1* × 1×) * (1× × 1*) ≡⟨ ap (λ x → x * (1× × 1*)) eq3 ⟩
      1* * (1× × 1*)  ≡⟨ ap (λ x → 1* * x) eq4 ⟩
      1* * 1*         ≡⟨ left-unit-law-for-* 1* ⟩■
    1* ■
    where
      eq1 : 1× ≡ 1* * 1×
      eq1 = (left-unit-law-for-* 1×)⁻¹
      eq2 : 1× ≡ 1× * 1*
      eq2 = (right-unit-law-for-* 1×)⁻¹
      eq3 : 1* × 1× ≡ 1*
      eq3 = right-unit-law-for-× 1*
      eq4 : 1× × 1* ≡ 1*
      eq4 = left-unit-law-for-× 1*
```

We define X to be the set and our binary operations on that set are $\times : X \to X \to X$ and $* : X \to X \to X$, then their identity elements are $1\times$ and $1*$ respectively. We then define types for the right and left unit laws for each operation and a transposition operation, where given elements x y z and $w : X$, $(x * y) \times (z * w) \equiv (x \times z) * (y \times w)$. This is now enough for the assumptions in our proof.

We can do the steps of our proof sequentially by using a series of equalities and the unit laws and transpostion operations we have assumed, so we use the new notation we have just defined. By using the left-unit-law for $1^\times$, $1^\times\times$ anything, including itself, will be equal to that argument, so we can have $1^\times \times 1^\times$ when $x = 1^\times$. We have to use the inverse because the law is defined strictly in Agda to be $1^\times \times x \equiv x$, so in order to have $x \equiv 1^\times \times x$ we just reverse the direction of it.

Then we can use our *ap* property to show that $1^\times \times 1^\times$ is equal to $(1^* * 1^\times) \times 1^\times$. We use $eq1$ to show that $1^\times \equiv 1^* * 1^\times$. We prove this by using the left unit law for *. Then in *ap* our x is $1^\times$ and our x' is $(1^* * 1^\times)$. Then we apply this to a function $(\lambda x \to x \times 1^\times)$, so that $f\ x$ is $1^\times \times 1^\times$ and $f\ x'$ is $(1^* * 1^\times) \times 1^\times$. Then by *ap*, these defintions must both be equal, so we have proved this equality.

We can do this for the other side using the same method. $eq2$ gives the other side of the equation, $1^\times * 1^*$ and is proved using the right unit law for *. Now we have $(1^* * 1^\times) \times (1^\times * 1^*)$ and we can easily get the next equality using our transposition function, giving us $(1^* \times 1^\times) * (1^\times \times 1^*)$.

Then we can get 1^* by using *ap* again but this time reducing the expression. We use $eq3$ to get from $(1^* \times 1^\times) * (1^\times \times 1^*)$ to $1^* * (1^\times \times 1^*)$. $eq3$ states that $(1^* \times 1^\times)$ is equal to 1^*, which we prove using the right unit law for $\times$. Then using ap with this proof and the function $(\lambda x \to x * (1^\times \times 1^*))$, we can see that $(1^* \times 1^\times) * (1^\times \times 1^*) = 1^* * (1^\times \times 1^*)$.

We now use *ap* for the final time, to get from $1^* * (1^\times \times 1^*)$ to $1^* * 1^*$. $eq4$ is $1^\times \times 1^* \equiv 1^*$ and is proved using the left unit law for $\times$, where x is 1^*. Then using *ap* with the function $(\lambda x \to 1^* * x)$, we see that the two steps are equal. Finally we just need to get from $1^* * 1^*$ to 1^*, which we can do using the left unit law for * (or the right unit).

Now we have prove the Eckmann Hilton theorem for units, but we still need to prove it in the general case. We have two elements a and $b : X$. Then we need to prove that $a \times b$ is equal to $a * b$. We do this in stages. In $claim1$, we prove that $a \times b = b * a$, then in $claim2$ we prove that $b \star a = a \times b$.

For $claim1$ we start by obtaining $(1^* * a) \times (b * 1^*)$ from $a \times b$. First we use *ap* on a proof that $a \equiv (1^* * a)$, which will be the left unit law for * and apply a function $(\lambda x \to x \times b)$ to it, which gives us a proof that $a \times b \equiv ((1^* * a) \times b)$. Then we use this and to get $(1^* * a) \times (b * 1^*)$. We apply a function $(x \to (1^* * a) \times x)$ to the proof that $b \equiv (b * 1^*)$, which is the right unit law for *.

Now we can use transposition to obtain $(1^\times * b) * (a * 1^\times)$, which exchanges the units. Then we can easily replace $(1^* \times b))$ with $(1^\times \times b)$ because we now know that $1^* \equiv 1^\times$. Therefore we can use our proof that units coincide and apply the function $(\lambda x \to (x \times b) * (a \times 1^*))$ to get $(1^\times \times b) * (a \times 1^*)$ then use it with the function $(\lambda x \to x * (a \times 1^\times))$ to replace $(a \times 1^*)$

with $(a \times 1^{\times})$.

We now have $(1^{\times} \times b) * (a \times 1^{\times})$, so to get b from this we apply a function $(\lambda x \to x * (a \times 1^{\times}))$ where x is b and x' is $(1^{\times} \times b)$, so we can just use the left unit law for $\times$ as the proof. Then we need to get from $(a \times 1^{\times})$ to a so we use the right unit law for $\times$ and apply the function $(\lambda x \to b * x)$ to get $b * a$.

```
claim1 : (a b : X) → a × b ≡ b * a
claim1 a b = a × b          ≡⟨ ap (λ x → x × b) (left-unit-law-for-* a)⁻¹ ⟩
  ((1* * a) × b)            ≡⟨ ap (λ x → (1* * a) × x) (right-unit-law-for-* b)⁻¹ ⟩
  (1* * a) × (b * 1*)       ≡⟨ transposition 1* a b 1* ⟩
  (1* × b) * (a × 1*)       ≡⟨ ap (λ x → (x × b) * (a × 1*)) (units-coincide ⁻¹) ⟩
  (1× × b) * (a × 1*)       ≡⟨ ap (λ x → (1× × b) * (a × x)) (units-coincide ⁻¹) ⟩
  (1× × b) * (a × 1×)       ≡⟨ ap (λ x → x * (a × 1×)) (left-unit-law-for-× b) ⟩
  b * (a × 1×)              ≡⟨ ap (λ x → b * x) (right-unit-law-for-× a) ⟩■
  b * a                     ■
```

We then prove *claim*2 in a similar way, exchanging $*$ for $\times$ and left unit law for the right unit law and supplying the appropriate functions to *ap*:

```
claim2 : (a b : X) → b * a ≡ b × a
claim2 a b =
  b * a                 ≡⟨ ap (λ x → x * a) (right-unit-law-for-× b)⁻¹ ⟩
(b × 1×) * a            ≡⟨ ap (λ x → (b × 1×) * x) (left-unit-law-for-× a)⁻¹ ⟩
(b × 1×) * (1× × a)     ≡⟨ (transposition b 1× 1× a)⁻¹ ⟩
(b * 1×) × (1× * a)     ≡⟨ ap (λ x → (b * x) × (1× * a)) units-coincide ⟩
(b * 1*) × (1× * a)     ≡⟨ ap (λ x → (b * 1*) × (x * a)) units-coincide ⟩
(b * 1*) × (1* * a)     ≡⟨ ap (λ x → x × (1* * a)) (right-unit-law-for-* b) ⟩
  b × (1* * a)          ≡⟨ ap (λ x → b × x) (left-unit-law-for-* a) ⟩■
  b × a                 ■
```

Now we have proved that $\times$ and $*$ are the same for any two arguments a and b in X, so we know that the operations are equal. What is left is to prove that the operations are associative and commutative.

First we prove the commutativity of $\times$. Given any x and $y : X$, then $x \times y$ is exactly the same as $y \times x$. We can prove this by using transitivity and our proofs for the operations being equal. By concatenating claim1 and claim2, we get for any x and $y : A$ $x \times y \equiv y * x \equiv y \times x$, whcih reduces to $x \times y \equiv y \times x$

```
commutativity-of-× : (x y : X) → x × y ≡ y × x
commutativity-of-× x y = claim1 x y • claim2 x y
```

Then we can do the opposite to prove the commutativity of $*$, by concatenating claim2 with claim1 to get $x * y \equiv x \times y \equiv y * x$:

```
commutativity-of-* : (x y : X) → x * y ≡ y * x
commutativity-of-* x y = claim2 y x • claim1 x y
```

Then we can use the commuativityof * to give a more readable proof that $\times$ and $*$ are the same by concatenating claim1 with it. This will give as an element of $x \times y$ equal to $y * \times \equiv x * y$:

```
the-same : (x y : X) → x × y ≡ x * y
the-same x y = claim1 x y • commutativity-of-* y x
```

Finally we prove the associativity of both operations.
This is given by *http://ncatlab.org/nlab/show/Eckmann-Hilton+argument*. For the associativity of $\times$ we want to show that $(a \times b) \times c \equiv (a \times b) \times (1^\times \times c) = (a \times 1^\times) \times (b \times c) = a \times (b \times c)$ for $\times$ and the same but with the other unit, 1^*. We then equate the statements as we did before, but if we want to replace $\times$ with $*$ we can use our proof that they are the same for any argument (as before we could only use units-coincide where we had arguments of $1^\times$ and/or 1^*:

```
associativity-of-× : (a b c : X) → (a × b) × c ≡ a × (b × c)
associativity-of-× a b c = (a × b) × c ≡⟨ ap (λ x → (a × b) × x) (left-unit-law-for-× c ⁻¹) ⟩
  (a × b) × (1× × c)                   ≡⟨ the-same (a × b) (1× × c) ⟩
  (a × b) * (1× × c)                   ≡⟨ transposition a 1× b c ⁻¹ ⟩
  (a * 1×) × (b * c)                   ≡⟨ ap (λ x → x × (b * c)) (the-same a 1×)⁻¹ ⟩
  (a × 1×) × (b * c)                   ≡⟨ ap (λ x → x × (b * c)) (right-unit-law-for-× a) ⟩
  a × (b * c)                          ≡⟨ ap (λ x → a × x) (the-same b c)⁻¹ ⟩∎
  a × (b × c)                          ∎

associativity-of-* : (a b c : X) → (a * b) * c ≡ a * (b * c)
associativity-of-* a b c = (a * b) * c   ≡⟨ ap (λ x → (a * b) * x) (left-unit-law-for-* c ⁻¹) ⟩
  (a * b) * (1* * c)                     ≡⟨ the-same (a * b) (1* * c) ⁻¹ ⟩
  (a * b) × (1* * c)                     ≡⟨ transposition a b 1* c ⟩
  (a × 1*) * (b × c)                     ≡⟨  ap (λ x → x * (b × c)) (the-same a 1*)  ⟩
  (a * 1*) * (b × c)                     ≡⟨ ap (λ x → x * (b × c)) (right-unit-law-for-* a) ⟩
  a * (b × c)                            ≡⟨ ap (λ x → a * x) (the-same b c)  ⟩∎
  a * (b * c)                            ∎
```

Now we have proved everything we need and the Eckmann Hilton Theorem is complete. We can get the homtopy theoretic version of this by instead of having a set with two binary operations, we have two groupoids, where composition of paths is their binary operations. Then the assumptions in the set theoretic version of the proof are satisfied by the structure of the groupoid (ie. associativity, operations being unital). Therefore we just need to prove that the composition of paths is commutative because this does not necessarily hold for a standard groupoid.

Homotopy theoretic proof

The agda formalization for this section is not completed for lack of time, but we fully explain the proof in English prose.

Eckmann Hilton 1. *The composition operation on the second loop space* $\Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$ *is commutative:* $\alpha \star \beta = \beta \star \alpha$ *for any* $\alpha, \beta : \Omega^2(A)$

First we show the **horizontal composition** of paths. For example, given elements a ,b and $c : A$ amd paths p and $q : a \equiv b$, r and $s : b \equiv c$. and paths $\alpha : p \equiv q$ and $\beta : r \equiv s$ between these [paths, if we concantenate the lower paths we have $p \bullet r$ and $q \bullet s$. This is equivalent to concatenating the higher paths α and B.

We can define this horizontal composition using induction and **whiskering**, which is composing a 2-path with a 1-path. For example $\alpha \bullet r$ would be left whiskering as the 2-path is on the left hand side. In Agda we define these as L and R. Then we can construct both of these by induction.

When we use left whiskering to define $\alpha \bullet r : p \bullet r \equiv q \bullet r$, we will do this by path induction on r. To do this we use a slightly different type of based path induction, to represent the inverse of it. In based path induction we wanted a proof that for a point a, and any $x : A$ that $a \equiv x$, but now we want $x \equiv a$.

We use transport as we did for based path induction with singleton types, but now we are saying that given a proof p and the inverse of its inverse, $((p^{-1})^{-1})$, we can get a proof of the return type Dxp as long as we have $Dx((p^{-1})^{-1})$. Then claim shows we have that element by using based path induction on where the return type C is now the inverse of D. We can give this as an element of D, by supplying a proof of $q : a \equiv x$, then the type of D to satisfy this proposition will be $D\ x\ q^{-1}$. Then the reflexivity element will be the same as the one given to the overall induction, d:

```
≡-Ind'' : {A : U} (a : A) (D : (x : A) → x ≡ a → U) (d : D a (refl a))
  → (x : A) (p : x ≡ a) → D x p
≡-Ind'' {A} a D d x p =
transport {x ≡ a} (D x) ((p ⁻¹)⁻¹) p (Lemma-2-1-4-[iii] A x a p) claim
```

```
  where
    C : (x : A) → a ≡ x → U
    C x q = D x (q ⁻¹)
    c : C a (refl a)
    c = d
    claim : D x ((p ⁻¹)⁻¹)
    claim = ≡-Ind' a C c x (p ⁻¹)
```

Now we can define the horizontal composition. We represent $(\alpha \bullet_l r) \bullet (q \bullet_r \beta)$ as R α $r\bullet$ L q β in Agda:

First we define the right whiskering. We want to use based induction on p, so define a function C, where given a point c in A and a path r there will be a type $p \bullet r \equiv q \bullet r$, Then f will be the reflexivity case, so there will be a type of $p \bullet r \equiv q \bullet r$. We can construct an element of this type by applying a function that given a path s we get the composition of s and refl b. This is then applied to α, so α will be equal to $f\ p \equiv f\ q$ which is the same as $p \bullet (\text{refl}\ b) \equiv q \bullet (\text{refl}\ b)$

```
horizontal-composition : (A : U) (a b c : A) (p q : a ≡ b) (r s : b ≡ c) (α : p ≡ q) (β : r ≡ s) →
    p • r ≡ q • s
horizontal-composition A _ _ _ p q r s α β = R α r • L q β
  where
    R' : (a b : A) (p q : a ≡ b) (α : p ≡ q) (c : A) (r : b ≡ c) → p • r ≡ q • r
    R' a b p q α = ≡-Ind' {A} b C f
      where
        C : (c : A) (r : b ≡ c) → U
        C c r = p • r ≡ q • r
        f : p • refl b ≡ q • refl b
        f = ap (λ s → s • refl b) α
    R : {a b c : A} {p q : a ≡ b} (α : p ≡ q) (r : b ≡ c) → p • r ≡ q • r
    R {a} {b} {c} {p} {q} α r = R' a b p q α c r
```

We want to use induction on q, so we need to have a function that given a $q : a \equiv b$, there will be a type $q \bullet r \equiv q \bullet s : U$. We can construct an element of this type by applying a function that given a path s we get the composition of refl b and s. Then this is applied to B, so B will be equal to $f\ r \equiv f\ s$ which is the same as refl $b \bullet r \equiv$ refl $b \bullet s$. Then f will be a reflexivity case for the induction.

```
    L' : (b c : A) (r s : b ≡ c) (β : r ≡ s) (a : A) (q : a ≡ b) → q • r ≡ q • s
    L' b c r s β = ≡-Ind'' {A} b C f
      where
```

```
    C : (a : A) (q : a ≡ b) → U
    C a q = q • r ≡ q • s
    f : (refl b) • r ≡ (refl b) • s
    f = ap (λ s → refl b • s) β
  L : {a b c : A} {r s : b ≡ c} (q : a ≡ b) (β : r ≡ s) → q • r ≡ q • s
  L {a} {b} {c} {r} {s} q β = L' b c r s β a q
```

Now that we have defined horizontal composition, we need to show that it is the same as actual composition. We can prove this by reusing our set theoretic version of the proof, where $*$ represents the horizontal composition and $\times$ represents composition of paths. We use the reflexivity case on the horizontal composition and the unit laws for these operations, so $*$ is an instance of it where every point is equal so $a \equiv b \equiv c$ and then all the paths between these points are represented by $\text{refl}\, a$. Therefore α and B are now $(\text{refl}\, a \equiv \text{refl}\, a)$. Then the proof for the entire theorem will just be the commutativity-of-$\times$, as we want to prove that $\bullet$ is commutative. Then we can reuse our module we defined before, but for loop spaces.

This means we need to redefine the unit laws in terms of loop spaces. The unit laws for $\times$ will use the proof of the lemma that $p \equiv p \bullet (\text{refl}\, y)$, where $p : x \equiv y$. In this case we have a path α instead of p between $(\text{refl}\, a)$ and itself. Then the lemma becomes $\alpha \equiv \alpha \bullet \text{refl}(\text{refl}\, a)$. The second half of this concatenation is the unit for $\times$, $1^{\times}$, so equivalently we have $\alpha \equiv \alpha \bullet 1^{x}$, the right unit law for $\times$. Then we can do the same with the other part of the lemma, that $p \equiv ((\text{refl}\, x) \bullet p)$ replacing p with $\alpha : \text{refl}\, a$, to get the left-unit law for $\times$.

The right unit law for * will $\alpha * 1* \equiv \alpha$, which reduces to $\alpha * (\text{refl}\ (\text{refl}\ a)) \equiv \alpha$:

```
  horizontal-composition' : (A : U) (a b : A) (p q : a ≡ b) (α : p ≡ q)
    → p • (refl b) ≡ q • (refl b)
  horizontal-composition' A a b p q α = horizontal-composition A a b b p q
      (refl b) (refl b) α (refl(refl b))

  Right-unit-law-for-* : (A : U) (a b : A) (p q : a ≡ b) (α : p ≡ q)
  → horizontal-composition A a b b p q (refl b) (refl b) α (refl(refl b))
  ≡ ap (λ r → r • (refl b)) α
  Right-unit-law-for-* A a b p .p (refl .p) = goal a b p
  where
    goal : (a b : A) (p : a ≡ b) → horizontal-composition A a b b p p
      (refl b) (refl b) (refl p) (refl (refl b))
      ≡ refl (p • refl b)
    goal a .a (refl .a) = refl (refl (refl a))
```

Then defining the rest of the laws, as for the set theoretic proof gives us the module defintition and therefore the proof of the Eckmann-Hilton Theorem in a homotopical context.

Chapter 4

Conclusion and Further Work

So far we have formalised MLTT and begun to formalise HoTT. I have learnt about constructing, and using (dependent) product, (dependent) function, coproduct, boolean, natural number and identity types in MLTT. I have also learnt about the Curry Howard correspondence between these types and intuitionistic logic.

I have learnt that identity types correspond to paths and that this can be used to build up HoTT, showing that properties that hold for equality also hold on paths in a space and homotopies on those paths. I have studied the basics of HoTT but there is much more to explore.

For example, there are many concepts in Homotopy Theory that can be represented in HoTT, that require more knowledge of Homotopy to understand. For example, homotopy groups as defined in [Univalent Foundations Program, 2013], are groups of equivalent spaces that have certain properties. Then there are higher homotopy groups of homotopies between homotopies and so on. Then these groups can be defined on spaces, such as spheres. Formalising all of these groups on spheres is an open problem. Some of these groups can be formalised into Agda, For example, the homotopy group of the circle (S^1) has been found to be equivalent to the set of integers, $\mathbb{Z}$ in [Licata and Shulman, 2013] and a formalisation of this proof in Agda is given.

In [Univalent Foundations Program, 2013] the chapter describing the formalisation of homotopy theory requires us to have understood MLTT, HoTT and Higher Inductive Types, so there is still much to learn. I hope to keep learning more so that I can eventually understand these concepts and see how homotopy can be used to benefit Computer Science and vice versa.

Bibliography

Ana Bove and Peter Dybjer. Language engineering and rigorous software development. chapter Dependent Types at Work, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03152-6.

Stanley Burris and H. P. Sankappanavar. A course in universal algebra. self-published, 2012. newer edition of oop Springer-Verlang edition.

T. Coquand. A remark on singleton types. March 2014.

Thierry Coquand. The paradox of trees in type theory. *BIT*, 32, 1991.

W. A. Howard. *The formulae-as-types notion of construction*, pages 480–490. Academic Press, London-New York, 1980.

Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 223–232, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5020-6.

Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium*, 1975.

Per Martin-Löf. An intuitionistic theory of types, 1998.

Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology and Goteborg University, 2007.

Thomas Streicher. *Investigations Into Intensional Type Theory.* PhD thesis, LMU Munich, 1993.

A. S. Troelstra. History of constructivism in the 20th century. In *Set Theory, Arithmetic, and Foundations of Mathematics*, pages 150–179. Cambridge University Press, 2011. ISBN 9780511910616.

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