UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

Treball final de grau

GRAU DE MATEMÀTIQUES
GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

PROOF VERIFICATION IN ALGEBRAIC TOPOLOGY

Autor: Xavier Ripoll Echeveste

Director: Dr. Carles Casacuberta
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 21 de juny de 2020

Abstract

Homotopy type theory is a relatively new field which results from the surprising blend of algebraic topology (*homotopy*) and type theory (*type*), that tries to serve as a theoretical base for theorem-proving software. This setting is particularly suitable for synthetic homotopy theory.

In this work, we describe how the programming language Agda can be used for proof verification, by examining the construction of the fundamental group of the circle $\mathbb{S}^1$. Then, trying to obtain the fundamental group of the real projective plane $\mathbb{R}\mathrm{P}^2$, we end up exploring a new construction of $\mathbb{R}\mathrm{P}^2$ as a higher inductive type.

Resum

La teoria homotòpica de tipus és un camp relativament nou que resulta de la sorprenent combinació de topologia algebraica (*homotopia*) i teoria de tipus (*tipus*), que intenta servir com una base teòrica per a *software* per demostrar teoremes. Aquest context és particularment adient per a la teoria d'homotopia sintètica.

En aquest treball, expliquem de quina manera es pot utilitzar el llenguatge de programació Agda per a verificar demostracions, examinant la construcció del grup fonamental de la circumferència $\mathbb{S}^1$. Després, intentant obtenir el grup fonamental del pla projectiu real $\mathbb{R}\mathrm{P}^2$, n'acabem explorant una nova construcció com a tipus inductiu d'ordre superior.

2010 Mathematics Subject Classification. 03B35, 03B70, 18G55

Contents

Introduction

Computer science was born as a tool *by* mathematicians *for* mathematicians, with the goal of aiding in repetitive calculations. The field has diversified a lot since its inception, but the idea of helping in mathematical research still lies in its heart. The ambition of computer scientists and mathematicians has kept growing, up to the question posed today: can computers help us prove theorems? Homotopy type theory is a field that can potentially help with this problem.

In a few words, type theory is a foundation of mathematics, alternative to set theory or category theory. In set theory, one way to build the natural numbers is through the recursive definition $0 := \emptyset$, $n' := n \cup \{n\}$. This yields a functional representation of the Peano naturals, but it also produces valid expressions like $0 \in 1$, which are often unwanted, as they mix the nature of the sets with the "pure" idea of the numbers. Of course, this does not really suppose any trouble to the working mathematician, as it is pretty easy to just "forget" about the particular implementation of the natural numbers and think of them as objects on their own rather than sets. This issue is only relevant when working on certain formal systems, in particular when one wants to do *computational* mathematics. In that case, it is important for the computer to be explicitly aware of what sets represent natural numbers and what sets do not.

The problem becomes unsolvable when dealing with more complex objects. For example, viewing functions as sets of ordered pairs makes them not generally representable in a finite setting. Even though one can define a function by just stating its action on any element of the domain (for example, $f(x) := x + 1$), the underlying set that represents the function can easily be infinite.

Type theory tries to tackle this problem by taking a more semantic view on mathematical objects. It operates with terms that have assigned types. In the case of the natural numbers, for example, instead of building them out of more elementary atoms, we just state them as what we want them to be: a type with an initial term and a way to build a new term for each existing one. Similarly, functions become simple computation rules instead of literally being made out of all the possible mappings they entail.

This philosophy—the idea that mathematical structures should be finitely describable—is known as constructivism, and is the quintessence of Martin-Löf's type theory. Although type theories precede Martin-Löf's by some decades, the one he published, known as intuitionistic type theory, is the first one to implement predicative logic, which is vital for producing meaningful mathematics. The notion of constructivism not only applies to objects such as algebraic structures, but also to predicates about these. In a set-theoretic setting, constructing the set

of all even natural numbers and the *statement* that such set is the set of all even numbers are two completely different things.

The first would be something like $\{n \in \mathbb{N} : 2 \mid n\}$. This is just a set of numbers that, by definition, are all the even natural numbers. We can name this set E, then forget about how it was defined, and we would lose the notion that it is the set of even naturals. In fact, by writing $\{0, 2, 4, 6, 8, ...\}$ we could describe that same set without using the notion of evenness.

The second could be expressed as $\forall n \in \mathbb{N}, n \in E \iff 2 \mid n$. This assertion is expressed using predicative logic, it is not an object in the same sense as E.

In type theory, we record the properties we care about of objects in the objects themselves. For example, the "set" of even naturals would be realized as a type consisting of ordered pairs, where the first component is any even number, and the second one is the *proof* that it is even. This way, not only we obtain the object itself (all the even natural numbers), we also encode the definition of the object with it. As a result of this technique, and the existence of certain native constructions (such as the dependent types introduced by Martin-Löf), intuitionistic type theory does not need of predicative or even propositional logic. It just needs a formal language that allows us to introduce some syntactical rules and create new types when needed.

How is a type theory *actually* used in a computer setting?

Computers can only understand the machine language, which is just a series of instructions that the processing unit is capable of executing. In order for humans to write complex programs, compilers were created, programs capable of translating a higher level language (e.g. C) into machine language. A compiler is not simple. The translation process actually consists of a series of different steps.

Traditionally, the way computer programs could aid in mathematics was pretty straightforward: one writes a program in, say, C, and then compiles the code into an executable. Then, the executable is run, yielding the desired results, for example making difficult numerical calculations.

But writing computer programs is no easy task, and is very error-prone. Modern compilers are equipped with tons of utilities that help minimize mistakes on the side of the human programmer. One of such tools is the type system.

A type system furnishes each variable and function in a program with a type (integers, floating point numbers, booleans, etc.), maybe automatically through analysis of the code, or maybe by hand by the programmer themself. This way it is harder for the human to mistakenly assign a wrong value to a variable. When the compiler validates that all variables match their corresponding type, we say that it is type checking the program.

Type systems have grown in complexity, allowing things like function types, product types, parametrized types (types that admit other types as parameters), etc. The last big addition to type systems was dependent types. These are types that depend on the values of other types. It turns out that this rather innocent-looking improvement is strong enough to allow type systems to implement an intuitionistic type theory.

By *abusing* types in this manner, a way was found to write mathematical proofs through the

type system. When doing this, we do not care about the rest of the compilation, or even the resulting executable: we only care about whether the program type-checks or not. Given sufficiently complex types, the mere fact that an element of such a type exists is a proof of a mathematical fact by itself.

Now that we have a picture of what an (intuitionistic) type theory is, what is *homotopy* type theory? First, we must introduce homotopy theory.

Homotopy theory is a branch of algebraic topology which deals with how paths in topological spaces can be deformed into one another. When paths starting and ending at a given base point are "loosened" keeping only the endpoints fixed, together with an adequate notion of composition, we obtain a natural group, called the fundamental group of the space (together with the base point). When the base point is disregarded, the group becomes a groupoid. One can then interpret the deformation of one path into another as a 2-path, or *homotopy*, or a path in the space of paths. This idea can be applied recursively, considering homotopies between homotopies, and so on. It can be easily seen how this builds up to an infinite tower of path spaces. When ignoring stronger topological structure and just looking at the paths, each topological space can be assigned what we call a homotopy type.

Homotopy type theory is a kind of *intensional* type theory, this is to say, there is a differentiation between propositional and definitional equalities. Intensionality is a double-edged sword. The good side is that propositional equality is always given by a type, which contains the information on *how* the two terms are related. This means that a statement of the style $x = y$ also contains the proof (in fact, *all the proofs*) that x equals y, in some sense. On the other hand, intensionality gives rise to a family of types which are very hard to study, the identity types, those representing equality statements. In fact, it turns out that identity types actually become a model for the higher path structures that appear in topological spaces, if we take equalities and think of them as paths joining the two equal things.

So, we have the word *type* in three different settings:

- As constructions used by logicians in type theory as a foundation of mathematics.
- As a method used by computer scientists in type systems in order to reduce bugs.
- As an invariant used by topologists to study and classify spaces (the homotopy type).

Homotopy type theory is what happens when these three concepts are interpreted as one.

Although homotopy type theory can be used as a basis for *all* of mathematics, it is particularly good for studying homotopy theory as it offers a synthetic setting for a subject that has traditionally been studied analytically, and, furthermore, allows it to be computerized so that the proofs can be checked automatically.

The idea of dependent types, which is necessary for this reasoning system to be possible, is fairly recent, and has not yet been implemented to many existing programming languages, mainly due to its great technical complexity. For the last three decades, a few research-oriented programming languages with dependent types have been popping up. One of them is **Agda**, a programming language developed by the universities of Chalmers and Gothenburg (Sweden) with the idea of exploiting dependent types in mind. A mathematician writes a program that constructs an element of a certain type, a type representing the statement to prove through the Curry-Howard correspondence. The Agda type checker then validates

the types, effectively verifying that the proof is correct. Even though Agda is, in principle, more oriented to general programming than other proof assistants (e.g. Coq), its lightweight syntax and good integration of higher equality types make it very suitable for the formalization of homotopy type theory.

Learning constructive mathematics is not easy. After explaining the basic concepts of type theory, we began our journey by examining the first non-trivial proof one encounters in algebraic topology, namely that of the fundamental group of the circle $\mathbb{S}^1$. This alone led us to study the rich concept of covering spaces. The theoretical study of this proof was accompanied by the practical implementation of an Agda program that implements it.

Afterwards, we wanted to try our luck with a more complicated case: the fundamental group of the real projective plane $\mathbb{R}\mathrm{P}^2$. While we were observing the problem, we realized that a potentially easier construction of $\mathbb{R}\mathrm{P}^2$ could perhaps be made as an alternative to those found in publications. In particular, our approach tries to remove all accessory constructions and leave only the minimal components that would give a type the homotopical structure of $\mathbb{R}\mathrm{P}^2$. The implications are that we have to work with second order inductive types, the difficulties of which we expose along the way. Albeit we accomplish the task of obtaining the fundamental group, we have difficulties proving that the construction is indeed a projective space, although we make substantial advances towards it.

Both parts of this thesis display the elegance of translating concepts from set-theoretic topology to homotopy type theory, where many times the definitions better represent the ideas behind them. Not only does homotopy type theory help us translate the theorems for the computer to verify, but it also succinctly encapsulates the essence of *paths* in a way that allows us to tell what topological constructions are homotopical and which are not.

Chapter 1

Preliminaries

1.1 Homotopy Theory

We begin by giving a brief introduction to homotopy theory.

Definition 1.1.1. In a topological space X, a **path** from $x \in X$ to $y \in X$ is defined as a continuous map $f : [0,1] \to X$ such that $f(0) = x$ and $f(1) = y$.

Often, we want to consider the composition of paths f (connecting x to y) and g (connecting y to z). This would be a path that walks over f in $[0, \frac{1}{2}]$ and walks over g in $[\frac{1}{2}, 1]$, denoted by $f \cdot g$. Then, we would expect to be able to define extended compositions like $f \cdot g \cdot h \cdot \cdots$. Unfortunately, path composition is not associative due to parametrization issues: in $(f \cdot g) \cdot h$, f and g are each "completed" in the subset of the domain $[0, \frac{1}{2}]$, whereas h has the other half of the domain for itself. Reciprocally, in $f \cdot (g \cdot h)$, f has as much of the domain as g and h together.

A way to allow associativity is by weakening our notion of path equality. For this we introduce the following notion:

Definition 1.1.2. Suppose two paths f_0 and f_1 with the same endpoints. A **homotopy** between f_0 and f_1 is a function $H : [0,1] \times [0,1] \to X$ such that $H(0,t) = x$, $H(1,t) = y$, $H(t,0) = f_0(t)$, $H(t,1) = f_1(t)$ for any $t \in [0,1]$. We say that f_0 and f_1 are **homotopic** (or equal up to a homotopy, or that they have the same homotopy class), if there exists a homotopy H between them, and we write $H : f_0 \Rightarrow f_1$.

Now, it is true that, given composable paths f, g, and h, there exists a homotopy $(f \cdot g) \cdot h \Rightarrow f \cdot (g \cdot h)$.

A homotopy $H : f \Rightarrow g$ can be seen as a "path" between f and g. In fact, H is nothing else than a path between f and g in the space of paths from x to y equipped with the compact-open topology. This gives us an insight of what higher homotopies could be: paths in higher path spaces. Alternatively, an n-path can be regarded as a map $[0,1]^n \to X$ which agrees with the homotopies "below" itself.

Another way of visualizing higher path spaces is categorically. We start by thinking of all

points in X as objects in a category, and all of the paths as morphisms. This is not a category, as path composition is only associative up to homotopy. If we instead take homotopy classes as morphisms, then composition is indeed associative. Not only that, but all paths can be reversed, in such a way that they are invertible up to homotopy. When all morphisms are invertible (such as in this case), the category is known as a groupoid. But, if we take homotopy classes, we are losing important topological information. To preserve the higher homotopical structure, we need *higher morphisms*. We can build a new category from each pair of objects, consisting of all the paths between them, and whose morphisms are the homotopies between such paths. If we repeat this at higher levels indefinitely, we obtain a "castle" of groupoids, which is called an ∞-groupoid.

The idea of paths and higher paths will be used all throughout this work.

1.2 Homotopy Type Theory

A very brief introduction to type theory is given in this chapter. Unfortunately, not every concept is explained as detailed as it is desirable. The recommended reference material is the Homotopy Type Theory book (The Univalent Foundations Program 2013).

1.2.1 Types and terms

In type theory, every *term* has a unique assigned *type*. Note two important differences with respect to set theory:

1. A term must belong to, at least, one type. In type theory it makes no sense to talk about a typeless term, and we cannot do much with it if we do not know its type. All the equations and computations involving a term expect it to have an assigned type.
2. A term must belong to, at most, one type. That is to say, terms cannot belong to two or more types at the same time. Note that, in particular, an analogous notion to that of a subset is not possible–at least not without some intermediate structure.

We write

$$a : A$$

to express that the term a has type A. The words *element* and *point* will be used interchangeably with term.

Types themselves are regarded as terms of a special type $\mathcal{U}$, called the "universe type". For example, when we say "A is a type", this is the same as denoting $A : \mathcal{U}$.

There are two main ways to obtain types:

1. To create them from scratch, via inductive types. We will see this option in sec. 1.2.6.
2. To create them from previously existing types, via type constructors.

Type constructors allow to build new types from existing ones. We can think of them as functions whose codomain is $\mathcal{U}$. To specify a new type constructor, one has to give the following rules for it:

- **Formation rules**. Preconditions to be met by the types used to build the constructed type.
- **Introduction rules** or constructors. Ways to build new terms. These are functions (possibly nullary) whose return type is the constructed type.
- **Elimination rules** or eliminators. Ways to use terms. These are functions that have, at least, one argument of the constructed type. Eliminators can often be non-dependent (called recursion principles or recursors) or dependent (called induction principles).
- **Computation rules**. They explain how the eliminators act on the constructors.

1.2.2 Function types

The function types are special in that their elements cannot be defined from simpler type-theoretic terms. From the function type constructor we will deduce most other constructors. This comes from the fact that type theories are often instances of typed lambda calculi, i.e. we are furnishing functions with types, rather than adding functions to types.

- Formation rule. Given any two types A and B, the (non-dependent) **function type** from A to B, denoted $A \to B$, contains all the functions $f : A \to B$ that assign to each term $a : A$ an element in B, $f(a)$.
- Introduction rules. There are a few ways to define functions. One is direct definition: $f(x) :\equiv \Phi$, where Φ is a formula that contains x as an unbound variable. Equivalently, we can use λ-abstraction: we denote by $\lambda(x : A).\Phi$ the function that takes an argument of type A and replaces all occurrences of x in Φ with it. So, we can also define $f :\equiv \lambda(x : A).\Phi$.
- Elimination rule. The obvious eliminator is application: given a function $f : A \to B$ and a term $a : A$, we can think of $f(a) : B$ as applying a to f to produce a term of B.
- Computation rule. The computation rule for functions tells us that $a : A$ applied to $\lambda(x : A).\Phi$ is Φ with all occurences of x replaced with a. Observe that this goes a step further than the elimination rule, as we are describing the function in terms of its construction (as a λ-abstraction in this case).

The types of functions with codomain $\mathcal{U}$ ($A \to \mathcal{U}$) are called **type families** or dependent types. Think carefully about what this means. Ordinary functions give us a value in a type for each value given. On the other hand, dependent types give us a whole type, for each value given.

Given a type family $B : A \to \mathcal{U}$, the dependent function type (also known as Π-type) $\prod_{(x:A)} B(x)$ comprises the functions whose codomain is a type family depending on the input *value*, i.e., given $f : \prod_{(x:A)} B(x)$ and $a : A$, then $f(a) : B(a)$. The rules for the dependent types are analogous to those of the non-dependent types.

If we want to build a function with two arguments, its type would be $f : A \to B \to C$. This

means that, when applied to a value $a : A$, f returns a value $f(a) : B \to C$. Then, we can apply a value $b : B$, to obtain $f(a)(b) : C$. Although it is also possible to define multiple parameters via Cartesian products, just like in set theory, that is not the most natural way to do it in type theory. When a function is presented in this way, we say it is curried. We often write $f(a, b)$ to mean $f(a)(b)$, for convenience.

1.2.3 Product types

As with function types, we have a both a non-dependent and a dependent version.

- Formation rule. Types A and B can form a type $A \times B$ called the (non-dependent) **product type** of A and B. If we have types A and $B : A \to \mathit{Ul}$, then $\sum_{(x:A)} B(x)$ is the dependent product of A and B. In the dependent case, b has to belong to the type $B(a)$.
- Introduction rule. Given $a : A$ and $b : B$, we obtain $(a, b) : A \times B$.
- Elimination rules. If we have a function $f : A \to B \to C$, then this rule gives us another function $g : (A \times B) \to C$. The dependent eliminator is akin to this one, but with f a dependent function.
- Computation rule. The eliminator tells us there exists a function $g : (A \times B) \to C$. The computation rule tells us how this function acts on the elements created by the introduction rules, namely pairs (a, b). Imagine we have $a : A$, $b : B$, and $g : A \to B \to C$. This rule defines $f((a, b))$ as $g(a)(b)$.

1.2.4 Coproduct types

The coproduct is the analogue of the disjoint union in set theory. It does not have a dependent version.

- Formation rule. As with product types, we take types A and B to obtain their **coproduct type** $A + B$.
- Introduction rule. There are two ways to introduce elements of $A + B$. One is, given a term $a : A$, we have a term $\mathsf{inl}(a) : A + B$. The other, given $b : B$, we have $\mathsf{inr}(b) : A + B$.
- Elimination rule. The non-dependent eliminator is very simple: for any type C, and given functions $f : A \to C$, $g : B \to C$, there is a function $h : (A + B) \to C$.
- Computation rule. As with product types, this just tells us how to apply the eliminator on the constructor. In this case, the function h given above behaves like this:

$$h(\mathsf{inl}(a)) = f(a)$$
$$h(\mathsf{inr}(b)) = g(b)$$

1.2.5 Identity types

In intensional type theories, such as homotopy type theory, two terms can be definitionally equal (also known as judgmentally equal) or they can be propositionally equal. Two terms are definitionally equal only when we impose so, and, in such case, they are fully interchangeable by one another. We denote that a and b are definitionally equal by writing $a \equiv b$. We

sometimes write $a :\equiv b$ to emphasize that a is being defined. The claim that two terms are definitionally equal cannot be disputed, it is not a proposition. This is used mainly when defining new terms and types, or for notation purposes.

On the other hand, two (not necessarily equal in the previous sense) terms of the same type can be compared for propositional equality. This means that, given two terms a and b of a type A, it makes sense to ask whether they are equal or not. As we will see, propositions are implemented through types in homotopy type theory, so we reserve the notation $a = b$ for the type of equalities between a and b, or, in the homotopical sense, the type of paths between a and b. We call this an **identity type** or path type.

The identity type is what mainly differentiates homotopy type theory from other kinds of type theory. Given a type $A : \mathcal{U}$, there exists a (possibly empty) type $a =_A b$ of identifications between $a : A$ and $b : A$. We often omit A when it is clear and just write $a = b$. The idea is that every term in $a =_A b$ is a proof that a is equal to b. This type is not always trivial: two elements in a type can be equal in many different ways. In fact, the complexity of identity types is what makes homotopy type theory interesting and what gives rise to the homotopical structure.

Path types are also type constructors. They are a bit different from the other seen so far in that they do not only require types, but also elements of such type. We can think of the type constructor $(- =_- -)$ as a function of type $\prod_{(A:\mathcal{U})} A \to A \to \mathcal{U}$. Now, as a type constructor, we should also give the rules on how it behaves.

The introduction rule tells us that, given a term of a type, it is equal to itself. In other words, for any $A : \mathcal{U}$ and $a : A$, we have $\mathsf{refl}_a : a = a$. The fact that this is the only introduction rule does not mean that there are not other paths.

The elimination rule for path types is one of the most important reasoning tools in homotopy type theory. Often described in its dependent form, hence called **path induction**, it allows us to build functions and prove statements about all paths in a path type, by just proving them on a few paths of the type. Suppose we have a type family C that assigns a type to every possible path in a type A (i.e. $C : \prod_{(x,y:A)}(x =_A y) \to \mathcal{U}$). We want to build a dependent function f that, for each path $p : x = y$, gives us a value of $C(x, y, p)$. The induction principle for path types states that, in order to obtain f, it is only necessary to define it on the paths refl. Then, the computation rule says that the f that we obtain in fact respects the values we have given at each refl.

1.2.6 Inductive types

An inductive type, in its purest form, is given by introducing a series of *constructors*. The idea is that an inductive type is "freely generated" by its constructors. The simplest inductive type has no constructors:

$$\mathbf{0} : \mathcal{U}$$

This is known as the **empty type**.

The type with a single constructor is known as the **unit type**:

$$\mathbf{1} : \mathcal{U}$$
$$\star : \mathbf{1}$$

Finally, we define the **type of booleans**:

$$\mathbf{2} : \mathcal{U}$$
$$0_{\mathbf{2}}, 1_{\mathbf{2}} : \mathbf{2}$$

Inductive types accept other kinds of constructors, not just elements. For example, the **naturals** can be defined as such:

$$0 : \mathbb{N}$$
$$\mathsf{succ} : \mathbb{N} \to \mathbb{N}$$

What this is telling us is that every element of the type $\mathbb{N}$ can be built as either 0, or as succ applied to another element of $\mathbb{N}$. Thus, the possible naturals are 0, $\mathsf{succ}(0)$, $\mathsf{succ}(\mathsf{succ}(0))$, etc.

Inductive types are regular enough that we can mechanically deduce what their elimination principles look like. In simple terms, to define a function out of an inductive type, we provide the image for each constructor.

For example, to build a function with domain $\mathbb{N}$, we must provide a value for 0, and a function that for every $n : \mathbb{N}$ gives us a value for $\mathsf{succ}(n)$. In the case of the naturals, this matches the traditional notion of recursion. Similarly, if we do this with a dependent function, we obtain the induction on the naturals. Elementarily, when given a family of types $P : \mathbb{N} \to \mathcal{U}$, if we provide an element of $P(0)$, and for every $n : \mathbb{N}$ we provide a function $P(n) \to P(\mathsf{succ}(n))$, then we effectively prove P for all naturals.

Inductive types are an idea that already appears in older type theories. Homotopy type theory presents a whole new class of inductive types, known as **higher inductive types**. These allow constructors whose codomain is not only the type being described, but also path types on that.

The most iconic example is the **circle**, $\mathbb{S}^1$, which is made up by a single point and a loop from the point to itself.

$$\mathsf{base} : \mathbb{S}^1$$
$$\mathsf{loop} : \mathsf{base} = \mathsf{base}$$

Or, taking higher paths, we can build higher dimensional spheres:

$$\mathsf{base} : \mathbb{S}^2$$
$$\mathsf{surf} : \mathsf{refl}_{\mathsf{base}} = \mathsf{refl}_{\mathsf{base}}$$

1.2.7 Univalence

In type theory there is no notion of "subtype". Hence, it can be hard to work with inclusions. To ease this work, we introduce a series of notions related to the idea of "equivalent types".

Definition 1.2.1. Two functions $f, g : A \to B$ are **homotopic** when they are point-to-point equal, i.e. $\prod_{(x:A)} f(x) = g(x)$. We write this type as $f \sim g$, and call its elements homotopies.

Definition 1.2.2. Two types A and B are **equivalent** when there exist functions $f : A \to B$ and $g : B \to A$ such that $g \circ f \sim \mathsf{id}_A$ and $f \circ g \sim \mathsf{id}_B$. We call f and g equivalences and say that they are mutually quasi-inverses. We write $A \simeq B$ for the type of equivalences between A and B. Formally, this type is:

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

One of the new concepts that homotopy type theory brought was that of univalence. Essentially, univalence is a way of easing the problem of identifying types.

Lemma 1.2.3 (Transport)**.** *Given a type family $P : A \to \mathcal{U}$ and a path $p : x = y$ of elements $x, y : A$, there is a function $p_* : P(x) \to P(y)$.*

Proof. It is enough to do path induction on the equality type $x = y$. This means we can assume $y \equiv x$ and $p \equiv \mathsf{refl}_x$, and hence $P(x) = P(y)$. Thus, we can take p_* to be the identity function of $P(x)$. □

When we need to make explicit the family over which we transport the path, we can write $\mathsf{transport}^P(p, x)$ for $p_*(x)$.

Lemma 1.2.4. *Given types A and B, there is a function* $\mathsf{idtoeqv} : (A = B) \to (A \simeq B)$.

Proof. Observe how $\mathsf{id}_{\mathcal{U}} : \mathcal{U} \to \mathcal{U}$ is a type family. Thus we can apply the transport lemma. We want to assign $\mathsf{idtoeqv}(p) = p_*$. For this, we must prove that each p_* is an equivalence. By path induction, we can suppose p is refl_A, and then p_* is id_A, which is trivially an equivalence. □

Axiom 1.2.5 (Univalence)**.** idtoeqv is an equivalence.

The inverse function of idtoeqv is known as ua for "univalence axiom". It basically states that, whenever two types are equivalent, we can treat them as equal. This axiom is due to Voevodsky and plays a very important role in homotopy type theory. Whereas there exist theorems that allow us to exchange two propositionally equal terms in expressions, there is no analogue for equivalent types, thus the axiom is introduced to fill this gap.

1.3 The Curry-Howard Correspondence

As we have hinted at, homotopy type theory, as a foundation for mathematics, is very apt for computation. This is due to the **Curry-Howard correspondence**. The Curry-Howard correspondence is the Rosetta stone between concepts in first order logic and type theory.

Each construction in classical logic has an analogue in type theory, usually as a type or type constructor. This means that the objects carrying the statements and proofs, are of the same nature as those that contain the things talked about. The best way to understand this is to take a look at the most relevant examples:

First order logic	Type theory
A statement	A type
A theorem	An inhabited type
A proof	An element of a type

All other principles can be deduced from these. Some types can be regarded as statements. When that is the case, whether they are inhabited or not is equivalent to whether they are true or false as a statement. That is why type theory is regarded as *constructivist*: when we prove a theorem, what we do is build an element of a type. We consider each term of a type to be a different proof of it. We often call these elements **witnesses**, so we can reserve the word proof for the process of building one. Let us see *how* to build statements out of types:

First order logic	Type theory
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \Rightarrow B$	$A \to B$
$\neg A$	$A \to \mathbf{0}$

The explanations for these equivalences are very intuitive.

$A \wedge B$ (representing *A and B*) is represented by the Cartesian product type $A \times B$. Having a proof of $A \wedge B$ amounts to having a proof of A and having a proof of B, or, under the type-theoretic interpretation, having an element of A and an element of B. But, we have those two if and only if we have an element $(a, b) : A \times B$.

Similarly, having a proof of $A \vee B$ is the same as having either a proof of A or a proof of B. Which is the same as having an element of $A + B$, that we know must have the form $\mathsf{inl}(a)$, with $a : A$, or $\mathsf{inr}(b)$ with $b : B$.

Perhaps the most surprising one is the implication. An implication $A \Rightarrow B$ is true whenever a proof for A yields a proof for B. This is exactly what a function does: given an element of A, we obtain an element of B.

And from that, we build the negation. Although we must beware of *reductio ad absurdum* in type theory, because in general it does not work, this construction comes from the idea that something is false whenever it implies a contradiction. In this case, if we have a function $A \to \mathbf{0}$, then A cannot be inhabited, as there is no point in $\mathbf{0}$ to take the points of A.

Nonetheless, it is often more comfortable for humans to describe proofs in the classical style, exposing a chain of arguments.

We have seen the key elements of propositional logic; now we need to construct quantifiers:

First order logic	Type theory
For some a in A, $P(a)$.	$\sum_{(a:A)} P(a)$
For all a in A, $P(a)$.	$\prod_{(a:A)} P(a)$

This is the key reason for which we need dependent functions and pairs, and the reason (as we will see later) that the introduction of dependent types in programming languages was so important for type theory. Dependent types are the analogues of the universal and existential quantifiers, and so give us the full power of first order logic.

Let us unpack this. For the existential, we provide dependent pairs. Each pair contains an element of A together with the proof of the statement P we claim about a. Observe that, once again, the corresponding type contains *all* possible proofs of $P(a)$. Conversely, if $b : A$ does not satisfy proposition P, then there does not exist any pair with b as the first component. From this point of view, $\sum_{(a:A)} P(a)$ is not only the claim that there exists an a such that $P(a)$, but it is also the type-theoretic analogue of the subset $\{a \in A : P(a)\}$.

For the universal quantifier, we do not want to show *some* elements of $a : A$; we want a proof of $P(a)$ for *every* $a : A$. This is what the dependent function does: for each a, it returns an element of $P(a)$. Again, this does not only say that all a satisfy $P(a)$, but also gives us the proof itself for each a.

We introduce the final ingredient for our recipe: identity types.

First order logic	Type theory
$a = b$	$a = b$

Now that we have been introduced to identity types in sec. 1.2.5, this table does not say anything trivial. The proof that $a = b$ is an element of the type $a = b$, for any two $a, b : A$. In fact, without this last translation, we would not be able to do much at all, because a large amount of mathematical statements can be reduced to saying that two things are the same.

As an example, if we have $p : a = b$, and $q : b = c$, then we can construct an element $p \cdot q : a = c$. This matches the idea of transitivity of equality.

This is also the point where homotopy type theory diverges from other (more extensional) type theories. Intensionality means that the type $a = b$ is not "boolean", i.e. it can have different, non-equal proofs. As it only seemed to add complexity, this property was traditionally discarded through the introduction of an axiom known as axiom K. Axiom K essentially imposes what we call **uniqueness of identity proofs**, or in other words, that all path types are mere propositions, either inhabited by a single path or empty. This kills the higher homotopical structure, rendering type theory more approachable from a certain computational point of view, but making elements less faithful to the proof they represent. The recent study of path types shows that preserving them results in a good foundation of homotopy theory.

Chapter 2

Agda

In this chapter we introduce the Agda programming language and our main work with it. An explanation of the main features of the language is given. More in-depth resources can be found at the official documentation.[1]

2.1 Setup

In order to be able to type check and compile Agda files, the Agda compiler should be set up. Optionally, Agda offers what is known as the Emacs mode, which is an Emacs extension that makes Agda into an interactive proof assistant. We will not worry about the Emacs extension in this work, but rather just on being able to compile Agda files via terminal.

2.1.1 Using Docker

Installing Agda itself is not extremely difficult, even for users unfamiliar with Haskell (in which Agda is written) or compilation processes in general. For our particular purposes, though, there is an important blocking issue. As our intention is to develop proofs in homotopy type theory, it is appropriate to try and work with the HoTT-Agda library developed by the Univalent Foundations Program researchers Brunerie et al. (n.d.). Unfortunately, the library has not been updated to work with the latest version of the Agda compiler. If one wants to be able to develop on this library and, simultaneously, try newer features of the Agda compiler, they have to manage two (or more) parallel versions of Agda. This is not very practical, as different versions of Agda require different versions of the Haskell compiler and its dependencies, and Agda needs configuration files which can differ depending on the particular libraries one wants to use. Besides, one must then be careful not to update the Agda installation until the libraries' maintainers have adapted the code for the latest version. Users who have worked with global package managers (e.g. Python, Cabal, tlmgr, and most operating system package managers) know how big of a hassle this can be.

With this in mind, a system has been developed to "encapsulate" each Agda and its depen-

[1] https://agda.readthedocs.io/

dencies in a per-project basis. The system uses Docker images developed by Ting-gian Lua[2], which instead of installing Agda on the host computer, run it inside a virtualized environment so the user can switch between different versions easily. On top of that, a script has been implemented to manage libraries.

The sources and instructions for using such development can be found in the attachments.

2.1.2 Local installation

If the reader still wants to install Agda on their computer, there are a few ways to do so.

The official project does not offer the precompiled binaries, but there are some platforms that distribute them. For UNIX-like systems, one can look in their preferred package manager. For Windows, a precompiled installer for the latest version exists at http://homepage.divms.uiowa.edu/~astump/agda/. Nonetheless, these may be outdated and sometimes have setup issues.

A viable alternative is to compile Agda's compiler from source. This is a resource-heavy task, so it is advised to try only on more powerful computers. The official recommendation is to install through Haskell's Cabal, a toolchain for building and packaging Haskell programs. Unfortunately, Cabal generally works with a global package index[3], and so installing Agda's dependencies might conflict with already installed Haskell packages, if there were any.

In order to do so one just has to install directly using the Cabal command line interface[4], specifying the desired Agda version. For example, for version 2.5.3, one would use:

```
cabal install Agda-2.5.3
```

If that fails, it might be necessary to install the Alex and Happy packages (dependencies for parsing Agda code) beforehand:

```
cabal install alex
cabal install happy
```

A preferable method might be Haskell's Stack, which aims to solve a few issues that Cabal presents, lack of isolation being one of them. For this, one has to download the required version of Agda from the repository[5]. Then, inside the root folder, execute:

```
stack install --stack-yaml=<X>
```

Where `<X>` stands for any of the `stack-*.yaml` files in the root folder. Stack will compile the binaries and copy them to `~/.local/bin` (or the location shown by running `stack path --local-bin`). This location should be added to the PATH environment variable.

Then, one should be able to run Agda successfully:

[2] https://hub.docker.com/r/banacorn/agda/.

[3] It is possible to use a newer mode known as *sandboxes*. This posed technical issues with older Agda versions and so was not taken as the main route, but the reader is encouraged to consider the option.

[4] https://agda.readthedocs.io/en/v2.6.1/getting-started/installation.html.

[5] https://github.com/agda/agda/releases.

```
agda --version
```

When installing via Stack, it is possible that Emacs mode does not work due to relative location issues.[6]

Finally, independently of whether Cabal or Stack was used, the dependencies need to be set up. For each necessary library, the following steps have to be taken:

1. Download the library to some location where it can stay. Most Agda libraries are available on GitHub. It is very important to download a release of the library that is documented to work with the Agda version installed, otherwise things will fail.
2. Copy the absolute location of the `*.agda-lib` file inside the library to the `libraries` file. This file should be placed in `~/.agda`, or in `%appdata%\Agda` in the case of Windows, and should contain one file location per line.
3. Write the library name, which can be found inside the `*.agda-lib` file seen before, inside the `defaults` file. The `defaults` file should be placed in the same location as `libraries` and also admits one library name by line. Setting the `defaults` file is not strictly necessary, as one can use the `--library` flag when running Agda to achieve the same result.

If everything has been done correctly, one should be able to check the installation of Agda:

```
agda --version
```

Running Agda is very simple. One executes `agda` followed by the name of the `*.agda` file. The default action of Agda when given a file is to type check it, if we want to compile it into an executable binary, we must also pass a `--compile` flag. This is because, in general, when using Agda for mathematics, the interesting part is usually just the type checking, due to the Curry-Howard correspondence.

If the values defined in the file are correctly typed, Agda will either display no output or just display a message stating that it is checking the file. Either way, if there are no errors displayed, this means that the file type-checks correctly and, therefore, that the proof it contains is correct.

Agda admits other flags, such as `--without-K`, that will later be explained in detail.

The most important thing to have in mind is that, in order for Agda to properly recognize the modules we define, it must be run from the "root" directory of our project. For instance, if we define a file `this/is/a/Test.agda` with a root module `this.is.a.Test`, then Agda must be executed from the directory that contains `this` (e.g. `agda this/is/a/Test.agda`), otherwise it will fail to find the file. Modules are explained in further detail in sec. 2.2.1.

2.2 The Language

Agda is a purely functional programming language. A more common programming paradigm is imperative, in which the programmer writes instructions for the computer to

[6] See the following issue: https://github.com/commercialhaskell/stack/issues/848.

execute in order, modifying the state of a program (e.g. changing the value of variables). In contrast, a functional language does not hold any state, hence there are no varibles with mutable values, only constants. Instead of changing the values along the way, the programmer instead tries to build a constant the value of which is obtained through chains of function applications.

2.2.1 Modules

Code in Agda is compartmentalized in modules, which can be nested (`moduleA.moduleB`). This is useful in order not to pollute the global namespace with lots of definitions.

Every Agda program must declare a module named after the file that contains it, for example a file `Test.agda` should expose a module:

```
module Test where

-- Contents of the module go here
```

If we are working on a project with multiple files, then the root module of each file has to respect the folder structure respect to some folder that we want Agda to consider the "root":

```
-- File this/is/a/Test.agda
module this.is.a.Test where
```

To use other modules in our project, we just import them:

```
import some.other.things
```

We can choose to import only part of the definitions in the module (`import A using (a1; a2; a3)`), import all but a few (`import A hiding (a4; a5; a6)`) or import with a different name (`import A renaming (a1 to a)`).

It is often useful to be able to re-export the definitions in a module, as if they had been defined locally. For example:

```
module A where
  a = ?

-- We can use A.a
b = A.a

open A

-- Now we can directly use the definitions of A
c = a
```

2.2.2 Function types

Agda incorporates very few basic constructs. The two most important ones are type assignment and definition.

We write

```
a : A
```

to express that there is a constant `a` of type `A`.

We write

```
a = b
```

To define `a` as having value `b`. This makes Agda's `=` operator equivalent to homotopy type theory's :≡, rather confusingly. In the homotopy type theory library (Brunerie et al. n.d.), == is used for identity types, and we follow this convention, although outside of that ≡ is generally used.

Agda incorporates dependent function types. A non-dependent function between types `A` and `B` is written like so:

```
f : A → B
```

If `B` is actually a type family on `A` (`B : A → Type`), then we can write:

```
f : (a : A) → (B a)
```

Function definition can be done through λ-abstraction:

```
suc : Nat → Nat
suc = λ n → (n + 1)
```

But often it is done through case analysis, which corresponds to applying the principle of type induction of the domain type.

```
factorial : Nat → Nat
factorial 0 = 1
factorial (suc n) = (suc n) * (factorial n)
```

Agda provides many syntactic facilities for functions. For example, we can write `(a : A) → (b : B)` as `(a : A) (b : B)`, and `(a : A) → (b : A)` as `(a b : B)`. If we trust Agda can infer the type of an argument, we can write `(a : _)` or `∀ a` instead.

Functions admit optional arguments, which the type checker has to be able to deduce. The most common use case is when a type that follows already implies the argument, such as in the case of dependent types:

```
f : {a : A} → (b : B a) → (C b)
{- No need to provide a as it was marked as {optional} and we can deduce it
from the type of b -}
f b = ?
```

Finally, we can present functions as infix operators by using _ as a placeholder for the arguments:

```
_+_ : Nat → Nat → Nat
0 + 0 = 0
0 + (suc n) = suc n
```

```
(suc n) + 0 = suc n
(suc n) + (suc m) = suc (suc (n + m))
```

Notice that Agda admits recursive definitions as long as they can be reduced down to an initial step, as the compiler incorporates a termination checker that ensures we do not write infinite recursions.

2.2.3 Universe types

Agda provides a type `Set` of types. The name is a little misleading, as, from a homotopy type theory point of view it represents the universe of small types, rather than the universe of sets (what we call 0-types). For this reason, in the homotopy type theory library this is renamed to `Type`.

`Type` is actually shorthand for `Type lzero`, which represents the first type in a hierarchy of universes indexed by a type `Level`. This exists to avoid Russell's paradox (which makes `Type : Type` impossible). Instead, Agda defines a chain of universes: `Type lzero : Type (lsuc lzero)`, etc. While this is not relevant for the math behind the proofs we will develop, it is very much present in the actual code, especially in the form of universe polymorphism, which allows us to parametrize the universe level. For example, if we wanted to write an identity function that applies to *all* types, we would have to do it like so:

```
id : {n : Level} → {A : Type n} → A → A
id x = x
```

This is saying: "`id` is a function that, for any universe level, and for any type in that level, is the identity function".

2.2.4 Record types

Besides function types and universes, Agda has a native structure called **`record`**. In broad terms, it generalizes the dependent pair type to be indexed by names. As a trivial example, tuples can be seen as an instance of a record type:

```
record Pair (A B : Type) : Type where
  field
    fst : A
    snd : B
```

Now, we can instantiate a pair like so:

```
p : Pair Nat Nat
p = record { fst = 0; snd = 1 }
```

By providing a **`constructor`** directive, we can define a custom notation for the introduction rule of that type:

```
record Pair (A B : Type) : Type where
  constructor _,_
  field
```

```
    fst : A
    snd : B

p : Pair Nat Nat
p = 0 , 1
```

2.2.5 Data types

Finally, Agda offers the **data** syntax that plays the role of inductive types.

```
data Nat : Type where
  zero : Nat
  suc  : Nat → Nat
```

Data types also admit parameters themselves:

```
data List (A : Type) : Type where
  [] : List A
  _::_ : List A → List A
```

When a given parameter can have different values for each constructor, we move it to the right side of the `:` and call it an index. In the following example, `(A : Type)` is a parameter but `Nat` is an index:

```
data Vector (A : Type) : Nat → Type where
  []  : Vector A 0
  _::_ : {n : Nat} → A → Vector A n → Vector A (suc n)
```

2.2.6 Built-ins

In some instances, Agda offers special treatment for some types, type constructors, and functions. We explain the most common ones.

The definition of natural numbers as seen above (i.e. an inductive type with two constructors) is not really efficient from the machine's point of view. To harness the potential of the computer but still be able to treat the naturals as an inductive type, Agda offers a `Nat` built-in type. In order to use it we just have to import it:

```
open import Agda.Builtin.Nat
```

This adds the type `Nat` with constructors `zero` and `suc` to the current scope, as well as some operators like `+` and `-` which are also optimized.

To offer the utmost flexibility, we can also mimic these built-ins ourselves and then tell the compiler we want it to use the efficient internal representation:

```
data Nat : Type where
  zero : Nat
  suc  : Nat → Nat

{-# BUILTIN NATURAL Nat #-}
```

This also allows us to type natural numbers with digits (e.g. `3` instead of `suc (suc (suc zero))`) and have the compiler translate them to the corresponding value automatically.

Another case that appears in most programs is the equality type. It is defined in the `Agda.Builtin.Equality` module as follows:

```
data _==_ {a} {A : Type a} (x : A) : A → Type a where
  refl : x == x
```

Observe how it is defined like we would define an inductive type with a single constructor `refl`. Although equality types are a primitive concept in homotopy type theory, their introduction and elimination rules already suggest that they behave like inductive types. This may help the reader get comfortable with the way path induction works.

With the tools seen so far, we can already show how programming in Agda is not as straightforward as transcribing a homotopy type theory proof from the paper to the screen. For example, pair types can be defined in two different ways. One, as record types:

```
record Σ {a b} (A : Type a) (B : A → Type b) : Type (a ⊔ b) where
  constructor _,_
  field
    fst : A
    snd : B fst
```

Another, as data types:

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

These two work almost identically for many purposes, but each have their own advantages. Agda will not recognize them as equal, though, so translating proofs using one to proofs using the other might sometimes be necessary.

2.2.7 `--without-K`

When run via terminal, we can pass various options to the Agda type checker. Some of these are related to the code itself, so it is more fitting to include them with the sources. A way to do this is by adding a special kind of comment which sets the desired options:

```
{-# OPTIONS [options...] #-}
```

For us, the most important option is *`--without-K`*:

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

By default, Agda allows formulating axiom K:

```
K : {A : Type} {a : A} (P : a == a → Type) → P refl → (loop : a == a) → P loop
K P p refl = p
```

Roughly, axiom K states that, in order to prove some property for all paths, it is enough to prove it for `refl`, which implies uniqueness of identity proofs (UIP), or, in other words, that all identity types are either trivial or empty. Although due to path induction this is true for

many types in homotopy type theory (0-types, for example), in general it is not. In order to prevent ourselves from accidentally writing something that implies UIP, we should use the `--without-K` flag. This way, formulating K or any equivalent statement will make our code not pass the type check.

2.3 Homotopy Type Theory in Agda

2.3.1 An example

Let us treat a simple hands on example: the commutativity of addition of natural numbers. The full source code of this proof is available in the attachments.

First, as a preamble, we deactivate uniqueness of identity proofs using `{-# OPTIONS --without-K #-}` and construct a few basic definitions regarding equality.

We begin the actual content by defining the naturals and their addition operation:

```
-- Natural numbers
data ℕ : Type lzero where
  zero : ℕ
  succ : ℕ → ℕ

-- Addition of naturals
_+_ : ℕ → ℕ → ℕ
zero + m = m
(succ n) + m = succ (n + m)
```

The definition of the natural numbers as an inductive type has already been seen. For the addition, we take the classical recursive definition.

These two definitions might seem simple enough, but their translation into type theory actually requires quite some of the theory explained so far. The data type is actually an inductive type, which means that it naturally has an elimination principle. This elimination principle is implicitly used in the definition of `_+_`, as it is a function that "eliminates" natural numbers (even if it does so into *other* natural numbers). The appearance of the eliminator is patent in the fact that we use case analysis (also known as pattern matching) on each constructor: one entry for `zero` and one for `succ`. Case analysis could be further used on the second argument, but this definition does not require it.

Now, we proceed to prove a lemma:

```
-- Proof that 0 is right identity of addition
add-right-id : (n : ℕ) → (n + zero) == n
add-right-id zero = idp
add-right-id (succ n) =
  (succ n) + zero
    =⟨ idp ⟩
  succ (n + zero)
    =⟨ ap (succ) (add-right-id n) ⟩
```

```
    succ n
      =∎
```

A few things to note here.

First of all, we remember that theorems (or lemmas) take the shape of types, whereas proofs are their inhabitants. We can identify every part in the example above. The statement of the lemma is the type of `add-right-id`, namely `(n : ℕ) → (n + zero) == n`. This is the type of functions that take a natural number `n`, and return an equality between `n + zero` and `n`. The proof will be the function `add-right-id` that we are defining.

Once again we use the eliminator through pattern matching. In the case of `zero`, we only write `refl`. This means that we trust Agda to apply the addition to the two terms (`zero + zero`). By the definition of `_+_`, any addition with `zero` as a left-hand side term, is reduced (definitionally) to the right-hand side. Hence, we obtain `zero`, which is equal to itself through `refl`. So, effectively, when building the proof, we are saying "`zero + zero` is the same thing as `zero`, and the proof is the path `refl`".

The second part might look more daunting, but is not much more difficult. Some of the imports at the top of the file define a syntax that allows us to define proofs given by the concatenation of multiple paths by writing them out in a readable way.

Suppose we have paths `p : a == b`, `q : b == c`, and `r : c == d`. Instead of just writing the concatenation of the paths `p • q • r`, the homotopy type theory library (and other libraries too) gives us a way to write it out as:

```
a =⟨ p ⟩ b =⟨ q ⟩ c =⟨ r ⟩ d =∎
```

This is actually the same as `p • q • r`, but resembles much more what we have in mind when writing the proof, and so is the generally preferred style.

Back to the proof, we first state that `(succ n) + zero` is the same thing as `succ (n + zero)`. We do not provide any path as proof, as they are definitionally equal (in particular, by the definition of `_+_` when the first operator uses `succ`).

The second step is more interesting. We use `ap` to take a path of type `(n + zero) == n`, apply `succ` to both sides of the equality, and obtain a new path of type `succ (n + zero) == succ n`. Observe that the path of type `(n + zero) == n` chosen is `add-right-id` itself. One might fear that the recursive call gets stuck for ever. But, since we invoke `add-right-id n` from the case of `add-right-id (succ n)`, we are going "down" the stack of `succ`s, and so it will end up reducing to the base case `add-right-id zero`. In contrast to some other programming languages, Agda is capable of knowing whether a recursive call will end or not, so we must not worry about infinite recursion.

We are now ready to prove the main theorem:

```
  -- Proof of the commutativity of addition
  add-comm : (n m : ℕ) → (n + m) == (m + n)
```

Similarly to the lemma, this proof is a dependent function. It takes any two natural numbers, `n` and `m`, and returns a path joining `n + m` and `m + n`. This proof is longer than that of the lemma, mainly because now we have to pattern match on two variables instead of one.

```
add-comm zero zero = idp
```

The case for `zero + zero` is trivial.

```
add-comm (succ n) zero =
  (succ n) + zero
    =⟨ add-right-id (succ n) ⟩
  succ n
    =⟨ idp ⟩
  zero + (succ n)
    =∎
```

This case does not introduce any new tools either. We start off by applying the lemma. Then, `succ n` is equal to `zero + (succ n)` by definition of `_+_`.

```
add-comm zero (succ m) =
  zero + (succ m)
    =⟨ idp ⟩
  succ m
    =⟨ ! (add-right-id (succ m)) ⟩
  (succ m) + zero
    =∎
```

Here we do the same, but in the opposite order. In consequence, we have to "reverse" the path of type `(succ m) + zero == succ m` to obtain one of type `succ m == (succ m) + zero` via `!`, which is known as the path reversal operator $p \mapsto p^{-1}$ in homotopy type theory.

The last case is the longest:

```
add-comm (succ n) (succ m) =
  (succ n) + (succ m)
    =⟨ idp ⟩
  succ (n + (succ m))
    =⟨ ap (succ) (add-comm n (succ m)) ⟩
  succ ((succ m) + n)
    =⟨ idp ⟩
  succ (succ (m + n))
    =⟨ ap (succ) (ap (succ) (add-comm m n)) ⟩
  succ (succ (n + m))
    =⟨ idp ⟩
  succ ((succ n) + m)
    =⟨ ap (succ) (add-comm (succ n) m) ⟩
  succ (m + (succ n))
    =⟨ idp ⟩
  (succ m) + (succ n)
    =∎
```

It is reduced to previous cases through recursion with the help of `ap` and the definition of `_+_`.

This concludes the proof. Now, it can be applied as such:

```
_ : (1 + 2) == (2 + 1)
_ = add-comm 1 2
```

We could go a step further and mark the arguments `n` and `m` as implicit, so that we could invoke this theorem just by writing `add-comm` without parameters, but we leave them explicit for illustrative purposes.

An interesting observation is that both the definitions and the theorems are implemented by defining Agda functions. What does this entail? First, this clearly embodies the philosophy of constructivism: the proofs are elements of a type. The consequences are subtle but important: proving something is the same as making a valid definition. Conversely, all definitions in type theory (and thus Agda) are "valid". In conventional mathematics, one might impose a definition and then study it to prove it is valid in some sense. Of course, this just is a way of hiding the fact that we prove a theorem that enables such definition to exist, we just present it in the opposite order. In type theory the same is as true, if not even more. The construction of an object often requires pieces that we have to previously construct, and those can sometimes be theorems, so many times we will see that defining something is as hard as proving something. In the end, one could even argue that whether a particular term should be deemed as a "definition" or as a "proof" on paper, is a matter of opinion.

2.3.2 Higher inductive types

Higher inductive types, although easy to explain, are very hard to implement. Classical Agda does not offer a native way to implement higher inductive types. A variant, known as Cubical Agda, which enables creating some higher inductive types, has recently been released, but the homotopy type theory project has not yet been ported to Cubical Agda.

Mathematicians have looked for alternate tricks for implementing higher inductive types.

The "naive" way to introduce higher inductive types is through postulates. **`postulate`** is a keyword in Agda that introduces an axiom. One could easily define the circle as:

```
data S¹ : Set where
  base : S¹

postulate
  loop : base == base
```

First, we create a type with a single point constructor `base`, and then tell Agda that there exists a path `loop` from `base` to `base`.

This method has a couple of issues. The first and most obvious, postulates are a dangerous tool. One could easily define:

```
data ⊥ : Set where
-- Nothing here
```

```
postulate
  impossible : ⊥
```

We have defined the empty type $\bot$ (what we call $\mathbf{0}$ in homotopy type theory), and through postulates we have introduced an element of the type. This creates an inconsistency which allows us to prove anything.

Another reason why this is complicated is that we do not get the full elimination principle, as Agda is not aware that `loop` is a constructor of the higher inductive type. This issue can be taken care of by being aware of it and introducing `loop` "by hand" in the definitions that require it.

A refinement of this technique, due to Licata (2011), and which came to be known as "Licata's trick", makes it all safer by using modules.

The idea is to build the type inside a module. Now, instead of exposing the whole type, we postulate a function that simulates the elimination principle and expose that instead. The users of the type then only use the elimination principle as a function, which is more cumbersome but also safer, as there is not need for further postulates. One only has to trust whoever has defined the type to do it correctly in the first place.

Chapter 3

The Circle

In order to gauge the complexity of proofs in homotopy type theory and Agda, we study an idiosyncratic exercise of homotopy theory: calculating the fundamental group of the circle. To do this, we have to explain a couple of concepts first: truncations and coverings.

3.1 Truncation

Intuitively, a type $A : \mathcal{U}$ is called an n-type (or just n-truncated) when all its identity spaces of order greater than n are trivial. A few key examples:

- $A : \mathcal{U}$ is a (-2)-type, or **contractible**, when there is a point all other points are equal to, i.e., $\sum_{(a:A)} \prod_{(x:A)} a = x$ ("there exists an a such that for every x, $a = x$").
- $A : \mathcal{U}$ is a (-1)-type when all of its points are trivial, i.e., $\prod_{(x,y:A)} x = y$ ("for all x and y of type A, there is a proof that x is equal to y"). This kind of types are also called **(mere) propositions**, because they contain no other information than "true" (they are inhabited) or "false" (they are not inhabited).
- $A : \mathcal{U}$ is a 0-type when all of its identity types are trivial. So, given any $x, y : A$ that are equal, then the type $x = y$ has a single element, i.e., it is a (-1)-type. This can be read as saying that "there is only one way in which x and y are equal to each other". A 0-type is also called a **set**, because it is a type that has no further homotopical information about its elements other than whether they are equal or not. This point of view agrees with looking at their identity types as mere propositions: two elements in a set are either equal or different, nothing else can be said about their equality status.

In view of this, we can define the notion of n-type inductively.

Definition 3.1.1. A type $A : \mathcal{U}$ is called a $\boldsymbol{-2}$**-type** if it is contractible, this is, if there exists a term $a : A$ such that $\prod_{(x:A)} a = x$. A type $A : \mathcal{U}$ is called a $(\boldsymbol{n+1})$**-type** if, for every x and y in A, $x = y$ is an n-type.

The idea of contractibility in homotopy type theory tries to represent the homonymous property in topology. Nonetheless, one might look at the definition and wonder whether, for

example, the circle $\mathbb{S}^1$ is contractible or not. The answer, just as in classical topology, is no. But why not? After all, we only have to provide a center of contraction (for example, base), and a function contr : $\prod_{(x:\mathbb{S}^1)} \mathsf{base} = x$. Every point in $\mathbb{S}^1$ is path connected to base by a piece of loop, so what is stopping us from building such a function?

The reason is continuity. All functions in homotopy type theory are naturally continuous, as they have to respect paths. This means that the function contr not only assigns a path from base to x for every $x : \mathbb{S}^1$, but does so in a *continuous* way. Pick any valid path for the case $x \equiv \mathsf{base}$. Then, as we travel along loop, the path to base changes as well, until we reach base again. Now, the image of the function contr at base should be the same as before, with a loop appended, as we have made one turn around the circle. But this amounts to saying that $\mathsf{refl}_{\mathsf{base}} = \mathsf{loop}$, which would imply the existence of a two-dimensional cell in the circle that just does not exist. In fact, the circle is not even a mere proposition, nor a set! We will see a formal proof of this in sec. 3.3.

Given any type A, we can introduce its ***n*-truncation** $\|A\|_n$ as the "n-type that best approximates A".

Definition 3.1.2. For $n \geq -1$, we take $\|A\|_n$ to be the higher inductive type generated by:

- a function $|-|_n : A \to \|A\|_n$,
- for each $r : \mathbb{S}^{n+1} \to \|A\|_n$, a "hub" point $h(r) : \|A\|_n$, and
- for each $r : \mathbb{S}^{n+1} \to \|A\|_n$ and each $x : \mathbb{S}^{n+1}$, a "spoke" path $s_r(x) : r(x) = h(r)$.

This definition uses a construction technique known as "hub and spokes" that we will later see in detail. The idea is that it adds the $(n+2)$-cells necessary for A to become an n-type (remember that in an n-type all $(n+1)$-loops on a point have to be equal, so we need to add $(n+2)$-paths between those that are not).

3.2 Covering Spaces

We remind some topological definitions which will be relevant in this chapter.

Definition 3.2.1. A **covering space** of a topological space X is a space C (known as the **total space**) together with a continuous function $p : C \to X$ (the **projection**) satisfying the following condition: each point x in X has a neighborhood U such that its preimage $p^{-1}(U)$ is the disjoint union of open sets, each homeomorphic to U.

We say that U is **evenly covered** by $p^{-1}(U)$. We call the preimage of each point in X its **fiber**. We can visualize C as "lying over" X, and the fiber of each point lying over it.

A very important property of covering spaces is that of path lifting.

Theorem 3.2.2. *Given a path $f : [0,1] \to X$ starting at $x \in X$, for each $\tilde{x} \in p^{-1}(x)$ there is a unique path $\tilde{f} : [0,1] \to C$ starting at $\tilde{x}$ and projecting onto f. We call $\tilde{f}$ a **lift** of f.*

We can also uniquely lift homotopies.

Theorem 3.2.3. *Given a homotopy $h : [0,1] \times [0,1] \to X$ starting at $x \in X$, for each $\tilde{x} \in p^{-1}(x)$ there is a unique homotopy $\tilde{h} : [0,1] \times [0,1] \to C$ starting at $\tilde{x}$ and projecting onto h. We call $\tilde{h}$ a **lift** of h.*

A proof of both theorems is given in Hatcher (2000), Section 1.1.

In a covering space, the base X parametrizes the total space: each point x of X "represents" its fiber. Each fiber is discrete, but the elevation of the topological structure of X bundles all the fibers, bringing a non-trivial topological structure to C. In homotopy type theory, we take the idea of parametrizing the fibers via the base space to be the very definition.

Definition 3.2.4. A **covering** of a type X is a type family $P : X \to \mathit{Ul}$ such that $P(x)$ is a set for each $x : X$. We call $P(x)$ the **fiber** of x, and $\sum_{(x:X)} P(x)$ the **total space**.

The fiber of each $x : X$ is now its image $P(x)$, which is a type. This allows us to assign entire "subspaces" to every point in the base space, in the same way as p^{-1} would do in the classical definition. The sum type $\sum_{(x:X)} C(x)$ not only joins all the fibers, but also gives them the appropriate homotopical structure of the analogous classical construction.

Just as with the classical version, homotopy type theory coverings have a unique path lifting property.

Theorem 3.2.5. *Given a path $p : x = y$ of X, for each $\tilde{x} : P(x)$ there is a unique path* $\mathsf{lift}(\tilde{x}, p) : (x, \tilde{x}) = (y, p_*(\tilde{x}))$ *in the type* $\sum_{(x:A)} P(x)$.

Proof. We apply path induction. It suffices to assume $y :\equiv x$ and $p :\equiv \mathsf{refl}_x$. Hence, we want to build a witness $\mathsf{lift}(\tilde{x}, \mathsf{refl}_x)$ of $(x, \tilde{x}) = (x, (\mathsf{refl}_x)_*(\tilde{x}))$. By definition of the transport operation $(-)_*$, $(\mathsf{refl}_x)_*$ equals $\mathsf{id}_{P(x)}$. So it is enough to take $\mathsf{refl}_{(x,\tilde{x})}$ as $\mathsf{lift}(\tilde{x}, \mathsf{refl}_x)$. □

In the following section we will gain a deeper understanding on how to deal with coverings in homotopy type theory.

3.3 The Fundamental Group of the Circle

We finally approach the calculation of the fundamental group of $\mathbb{S}^1$. This was used as an exercise in learning both homotopy type theory and Agda. The proof showcased here is practically the same that has been implemented and attached to this work. Compared to that of the commutativity of addition exposed in sec. 2.3, this is non-trivial as it deals with topological concepts (fibrations, paths and homotopies), which result in more complex type-theoretic constructions (type families, identity types). The program for the proof is well self-documented and tries to guide the reader along the path explained in this section.

In classical topology, the fundamental group is defined as such:

Definition 3.3.1. Given a topological space X together with a base point x, we define the **fundamental group** of (X, x) as the group $\pi_1(X, x)$ formed by the homotopy classes of the paths from x to x, and the group operation defined as $[f] \cdot [g] = [f \cdot g]$.

That this is indeed a group requires proving that the equivalence classes respect composition, and that the group laws are fulfilled. This can be found in Hatcher (2000) Proposition 1.3. The identity is none other than the constant path on x, and the inverses are given by walking the paths backwards, i.e. $f^{-1}(t) = f(1 - t)$.

In homotopy type theory, the quotient is not necessary, as paths that are homotopy equivalent are *propositionally equal*. What we need to do, though, is make sure that the fundamental group is a discrete set, so we have to kill all higher order paths:

Definition 3.3.2. The **fundamental group** of a based type (X, x), denoted as $\pi_1(X, x)$, is the 0-truncation of its loop space at x, i.e.:

$$\pi_1(X, x) = \|\Omega(X, x)\|_0$$

The classical proof uses the universal cover of $\mathbb{S}^1$, which is the real line $\mathbb{R}$ with the projection $p(t) = e^{it}$. This can be visualized as $\mathbb{R}$ "going around" $\mathbb{S}^1$ in circles, like a helix. One can lift paths from $\mathbb{S}^1$ to its cover, and there they can be classified by the number of turns they do.

In this proof, we do something similar. We build the covering space and, for each point $x : \mathbb{S}^1$, show an equivalence between its fiber and the paths from base to x. Conceptually, we are doing the same thing: we have "as many" paths as elements in the fiber, because each point in the fiber is another turn completed by the path starting at base. Let us start by defining the cover.

Definition 3.3.3. Define $\mathsf{code} : \mathbb{S}^1 \to \mathit{Ul}$ by circle recursion:

$$\mathsf{code}(\mathsf{base}) = \mathbb{Z}$$
$$\mathsf{ap}_{\mathsf{code}}(\mathsf{loop}) = \mathsf{ua}(\mathsf{succ})$$

The fiber at base is $\mathbb{Z}$ by definition. But we need to assign a fiber to all the points in $\mathbb{S}^1$, not just base. Instead of saying what the fiber is for a given point, we say how the path loop of $\mathbb{S}^1$ should be lifted to a path in Ul (where $\mathbb{Z}$ belongs). succ induces an equivalence between $\mathbb{Z}$ and $\mathbb{Z}$, which can be converted into a path via the univalence axiom.

The next step consists in introducing two functions, encode and decode, which will become the two directions of the equivalences we want to find.

Definition 3.3.4.

$$\mathsf{encode} : \prod_{x:\mathbb{S}^1} (\mathsf{base} = x) \to \mathsf{code}(x)$$
$$\mathsf{encode}(x, p) = \mathsf{transport}^{\mathsf{code}}(p, 0)$$

This is a natural step in the direction hinted before. Suppose a fixed x; we want to assign to each path joining x and base an element of $\mathsf{code}(x)$. $\mathsf{transport}$ takes the path $p : \mathsf{base} = x$ to a function $\mathbb{Z} \to \mathsf{code}(x)$ (which can be thought of as p lifted to the covering space).

We observe that, because $\mathsf{transport}$ is functorial, encode takes any loop on base to a composition of functions, like so:

$$\begin{aligned}
&\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{\pm 1} \cdot \mathsf{loop}^{\pm 1} \cdot \mathsf{loop}^{\pm 1} \cdot \cdots, -) \\
=&\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{\pm 1}, -) \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{\pm 1}, -) \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{\pm 1}, -) \circ \cdots \\
=&\mathsf{succ}^{\pm 1} \circ \mathsf{succ}^{\pm 1} \circ \cdots
\end{aligned}$$

The key idea here is to think of $\mathbb{Z}$ as a pointed type $(\mathbb{Z}, 0)$, and the application of code to loop as the path from $(\mathbb{Z}, n)$ to $(\mathbb{Z}, \mathsf{succ}(n))$. Now we can see what the result of transport above is when applied to 0: the winding number of the path p, i.e., the number of "net" turns it does.

On the other hand, we have the decode function of type $\prod_{x:\mathbb{S}^1} \mathsf{code}(x) \to (\mathsf{base} = x)$. The definition in this direction is not so easy: we must use circle induction. This means we have to provide an image for base of type $\mathsf{code}(\mathsf{base}) \to (\mathsf{base} = \mathsf{base})$ and a path from this function to itself lying over loop. For the image of base, we pick the natural choice $n \mapsto \mathsf{loop}^n$, which takes any integer to a loop with that as its winding number. For the path, as this is a dependent function, we have to prove $\mathsf{loop}_*(\mathsf{loop}^-) = \mathsf{loop}^-$, or, writing the full type of the transport:

Lemma 3.3.5. *There is a path of type* $\mathsf{transport}^{y \mapsto \mathsf{code}(y) \to (\mathsf{base} = y)}(\mathsf{loop}, \mathsf{loop}^-) = \mathsf{loop}^-$.

Proof.

$$
\begin{aligned}
&\mathsf{transport}^{\mathsf{code}(-) \to (\mathsf{base} = -)}(\mathsf{loop}, \mathsf{loop}^-) \\
&= \mathsf{transport}^{(\mathsf{base} = -)}(\mathsf{loop}, -) \circ \mathsf{loop}^- \circ \mathsf{transport}^{\mathsf{code}(-)}(\mathsf{loop}^{-1}, -) && (1)\\
&= (- \cdot \mathsf{loop}) \circ \mathsf{loop}^- \circ \mathsf{transport}^{\mathsf{code}(-)}(\mathsf{loop}^{-1}, -) && (2)\\
&= (- \cdot \mathsf{loop}) \circ \mathsf{loop}^- \circ \mathsf{succ}^{-1} && (3)\\
&= n \mapsto \mathsf{loop}^{\mathsf{succ}^{-1}(n)} \cdot \mathsf{loop} && (4)\\
&= n \mapsto \mathsf{loop}^n && (5)
\end{aligned}
$$

(1) and (2) are by the action of transport on type families of the form $y \mapsto A(y) \to B(y)$ and $y \mapsto \mathit{base} = y$, correspondingly (see The Univalent Foundations Program 2013, 2.9.4 and 2.11.2). (3) is due to the functoriality of transport, and (4) and (5) due to reducing the function composition and then the path concatenation. □

Definition 3.3.6. Define $\mathsf{decode} : \prod_{x:\mathbb{S}^1} \mathsf{code}(x) \to (\mathsf{base} = x)$ by circle recursion. For the image at base, pick loop^-. For the lifting of loop, use the path from lemma 3.3.5.

Next, we prove that code and decode are inverse functions:

Lemma 3.3.7. *For each* $x : \mathbb{S}^1$, $p : \mathsf{base} = x$, *and* $c : \mathsf{code}(x)$, *we have:*

$$\mathsf{decode}_x(\mathsf{encode}_x(p)) = p$$

$$\mathsf{encode}_x(\mathsf{decode}_x(c)) = c$$

Proof. For the first equality, we apply path induction. So it suffices to prove the case $x = \mathsf{base}$, $p = \mathsf{refl}_{\mathsf{base}}$.

$$
\begin{aligned}
&\mathsf{decode}_{\mathsf{base}}(\mathsf{encode}_{\mathsf{base}}(\mathsf{refl}_{\mathsf{base}})) \\
&= \mathsf{decode}_{\mathsf{base}}(\mathsf{transport}^{\mathsf{code}}(\mathsf{refl}_{\mathsf{base}}, 0)) && (\text{definition of encode})\\
&= \mathsf{decode}_{\mathsf{base}}(0) && (\text{transport over refl is trivial})\\
&= \mathsf{loop}^0 && (\text{definition of decode})\\
&= \mathsf{refl}_{\mathsf{base}} && (\text{path composition})
\end{aligned}
$$

For the second equality, we apply circle induction. Usually, we would have to supply an image for base and check that the application respects loop. But, in the case of base, the

codomain is $\mathbb{Z}$, which is a set, and sets do not have non-trivial paths. Or, in other words, any loop (including loop) will always be lifted to a trivial path. So we only need to check that $\mathsf{encode}_{\mathsf{base}}(\mathsf{decode}_{\mathsf{base}}(n)) = n$ for all $n : \mathbb{Z}$. We apply integer induction. The case for $n = 0$ is true by definition. The positive and negative cases are analogous to each other, so we do the positive case:

$$
\begin{aligned}
&\mathsf{encode}_{\mathsf{base}}(\mathsf{decode}_{\mathsf{base}}(\mathsf{succ}(n))) \\
&= \mathsf{encode}_{\mathsf{base}}(\mathsf{loop}^{\mathsf{succ}(n)}) && \text{(definition of decode)} \\
&= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{\mathsf{succ}(n)}) && \text{(definition of encode)} \\
&= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{n} \cdot \mathsf{loop}, 0) && \text{(composition of paths)} \\
&= (\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{n}, -) \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, -))(0) && \text{(functoriality of transport)} \\
&= (\mathsf{succ}^{n} \circ \mathsf{succ})(0) && \text{(inductive hypothesis)} \\
&= \mathsf{succ}(n) && \text{(application)}
\end{aligned}
$$

□

Theorem 3.3.8. *There is a family of equivalences* $\prod_{x:\mathbb{S}^1}(\mathsf{base} = x) \simeq \mathsf{code}(x)$.

Proof. We apply lemma 3.3.7 to see that encode and decode act as mutual quasi-inverses. □

Corollary 3.3.9.

$$\pi_1(\mathbb{S}^1) = \mathbb{Z}.$$

Proof. We use theorem 3.3.8 with $x = \mathsf{base}$. This gives us an equivalence $(\mathsf{base} = \mathsf{base}) \simeq \mathbb{Z}$. We apply the univalence axiom to obtain an equality from the equivalence. Applying 0-truncation to both sides gives us $\|\mathsf{base} = \mathsf{base}\|_0 = \|\mathbb{Z}\|_0$. On the left side, we have the definition of $\pi_1(\mathbb{S}^1)$. On the right, because $\mathbb{Z}$ is a set, we obtain $\mathbb{Z}$ again. So we have $\pi_1(\mathbb{S}^1) = \mathbb{Z}$. □

The last step is to prove that the equivalence on base takes path composition to addition, in order to see that it is a group homomorphism as well.

Theorem 3.3.10. *$\pi_1(\mathbb{S}^1)$ and $\mathbb{Z}$ are isomorphic as groups.*

Proof. It is enough to see that, for all p, q in $\pi_1(\mathbb{S}^1)$, $\mathsf{encode}_{\mathsf{base}}(p \cdot q) = \mathsf{encode}_{\mathsf{base}}(p) + \mathsf{encode}_{\mathsf{base}}(q)$.

$$
\begin{aligned}
&\mathsf{encode}_{\mathsf{base}}(p \cdot q) \\
&= \mathsf{transport}^{\mathsf{code}}(p \cdot q, 0) && \text{(definition of encode)} \\
&= (\mathsf{transport}^{\mathsf{code}}(q, -) \circ \mathsf{transport}^{\mathsf{code}}(p, -))(0) && \text{(functoriality of transport)} \\
&= (\mathsf{succ}^{\mathsf{encode}_{\mathsf{base}}(q)} \circ \mathsf{succ}^{\mathsf{encode}_{\mathsf{base}}(p)})(0) && \text{(path lifting)} \\
&= \mathsf{encode}_{\mathsf{base}}(q) + \mathsf{encode}_{\mathsf{base}}(p) && \text{(definition of succ)}
\end{aligned}
$$

□

Chapter 4

The Real Projective Plane

4.1 Classical Construction

For the final part of this thesis, we will take a closer look at an interesting family of types and how they are built in homotopy type theory: the real projective spaces.

In classical topology, the **real projective space of dimension *n***, denoted by $\mathbb{R}\mathrm{P}^n$, is the topological space $\mathbb{S}^n/R$, where R is the relation that identifies each point with its antipode. We can also describe the real projective spaces as CW complexes using the following facts:

- The sphere $\mathbb{S}^n$ has a CW complex structure with two 0-cells, two 1-cells, two 2-cells, etc. up to two n-cells.
- The quotient space by the relationship R glues each pair of i-cells antipodally ("flipping" one of the i-cells).

Thus, $\mathbb{R}\mathrm{P}^n$ is a CW complex with one cell for each dimension from 0 up to n. It is important to notice that there is more than one CW complex with these cells. For example, if one were to try and build a CW complex with one 0-cell, one 1-cell, and one 2-cell, it would most probably *not* end up being the real projective plane $\mathbb{R}\mathrm{P}^2$. To get the projective plane, one has to mind using the 2-cell to glue the 1-cell to itself *reversed*.

The projective spaces owe most of their homotopical structure to the spheres they come from: $\mathbb{R}\mathrm{P}^0$ is a single-pointed space, $\mathbb{R}\mathrm{P}^1$ is homeomorphic to $\mathbb{S}^1$. For the rest, they all share their higher homotopy groups with the sphere: $\pi_k(\mathbb{R}\mathrm{P}^n) \cong \pi_k(\mathbb{S}^n)$ for all $k > 1$.

But what about the fundamental group? Calculating the fundamental group of projective spaces is very educational, because it helps us conceptualize the relationship between them and the spheres in a more visual way.

Take any projective space $\mathbb{R}\mathrm{P}^n$ as $\mathbb{S}^n/R$, with R as above. Consider then the covering space $p : \mathbb{S}^n \to \mathbb{R}\mathrm{P}^n$ given by the projection of the equivalence relation R. Each fiber has exactly two elements: the two antipodal points that have been identified. Covering spaces have a unique path lifting property, which states that, given a point $x \in \mathbb{R}\mathrm{P}^n$ and an element $\tilde{x}$ of its fiber $p^{-1}(x)$, then any path starting at x "lifts" to a unique path in the covering space starting at $\tilde{x}$. In our case, if we take any loop $\gamma : [0, 1] \to \mathbb{R}\mathrm{P}^n$, $\gamma(0) = \gamma(1) = x$, we can lift it to a path

in $\mathbb{S}^n$. As the fibers have two elements, we say that the covering space has two "sheets". So, depending on our choice of γ, it can be lifted in two different ways: taking both endpoints of the lifted path to be the same (say, $\tilde{x}$), or taking each endpoint to be the antipodal of the other ($\tilde{x}$ and $-\tilde{x}$).

For the first, we obtain a loop in the sphere. As $\mathbb{S}^n$ has a trivial fundamental group for $n > 1$, that loop is homotopic to the constant path on $\tilde{x}$. This homotopy on the covering space induces a homotopy on the base space taking γ to the constant path on x. So the loop is then trivial.

For the second, we see that the lifted path is not a loop, so it cannot be contracted to a constant path without moving its endpoints, which we cannot do as that would change the endpoints of the underlying loop γ. So, we have a non-trivial loop γ on $\mathbb{R}\mathsf{P}^n$. What can be said of such loop? It seems to be the only generator of the fundamental group of $\mathbb{R}\mathsf{P}^n$, so the only thing left to do is to check what order it has. Hence, we want to see what $\gamma \cdot \gamma$ is. As we have done before, we can lift the path to $\mathbb{S}^n$. The first γ is, as before, a path from $\tilde{x}$ to $-\tilde{x}$. In order for $\gamma \cdot \gamma$ to be a valid loop, the second γ has to be lifted to a path from $-\tilde{x}$ to $\tilde{x}$, otherwise the endpoints do not match. But then, we obtain a loop on $\tilde{x}$, which makes $\gamma \cdot \gamma$ homotopic to the constant path on x.

So, we have seen that the loops on x are of two kinds: those homotopic to the constant path, and those which are not. But those of the second kind, when repeated, are homotopic to the constant path. Therefore, the fundamental group of $\mathbb{R}\mathsf{P}^n$ is isomorphic to $\mathbb{Z}/2\mathbb{Z}$.

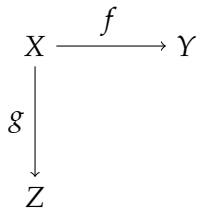
Why does this not work for $\mathbb{R}\mathsf{P}^1$? Because, for $n > 1$, $\mathbb{S}^n$ is simply connected, but $\mathbb{S}^1$ is not.

4.2 Pushouts

As we have seen, higher inductive types allow mimicking CW complexes by using n-dimensional paths as n-cells. Unfortunately, this is not always possible. An n-path can only connect *two* $(n-1)$-paths, whereas cells can join an arbitrary number of lower dimensional cells.

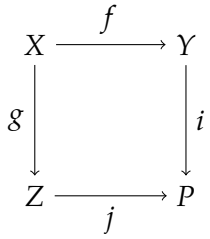
So the question is raised: how does one build more complex spaces in homotopy type theory? In classical topology, there are tools like *gluing* (quotient spaces) that allow us to join or collapse spaces in different ways. In homotopy type theory we need more refined techniques, because we want these to preserve the homotopical invariants we are working with. In sec. 3.1 we have seen a way to do this, parametrizing 1-types by lower dimensional cells. As an alternative, more robust method, in this chapter we present the **pushout**.

In category theory, a pushout is a kind of colimit. This roughly means that it is a unique (up to factorization) object out of a diagram. The diagram for a pushout is usually represented as such:

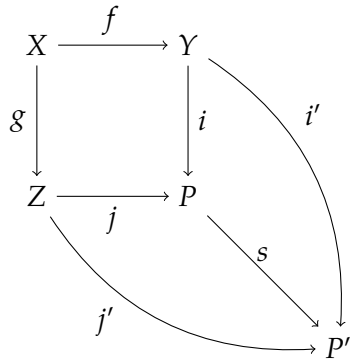


In certain categories, there is a pushout of such diagram.

Definition 4.2.1. The **pushout** of a diagram $Y \xleftarrow{f} X \xrightarrow{g} Z$ is an object P together with morphisms $Y \xrightarrow{i} P \xleftarrow{j} X$ such that the following diagram commutes:



and such that, for any other such diagram $Y \xrightarrow{i'} P' \xleftarrow{j'} X$, there exists a unique morphism $s : P \to P'$ such that the full diagram commutes:



This last property is know as the universal property of the pushout, which makes it unique up to isomorphism.

The category Top of topological spaces is cocomplete, which means that it contains all (set-indexed) colimits, such as pushouts. The pushout of two topological spaces Y and Z is the space $(Y \sqcup Z)/\sim$, where $\sim$ is the equivalence relationship generated by $\iota_Y(f(x)) \sim \iota_Z(g(x))$ for all x in X, and ι_Y, ι_Z are the inclusion functions. This can be visualized as "gluing" Y and Z along the points that share a preimage in X.

Things get ugly once we step into homotopy theory territory. Homotopy equivalences can be studied from a categorical point of view by taking homotopy classes of maps as morphisms. The resulting category, known as $\mathsf{Ho}(\mathsf{Top})$, does *not* have all colimits. In particular, we cannot always build pushouts. A classical example: consider the diagram $\mathbb{D}^2 \hookleftarrow \mathbb{S}^1 \hookrightarrow \mathbb{D}^2$ given by the standard inclusion of the circle into two copies of the closed 2-dimensional disk. The

pushout of this diagram is the sphere $\mathbb{S}^2$, because it is the result of gluing the two disks along their boundaries:

$$\begin{array}{ccc} \mathbb{S}^1 & \hookrightarrow & \mathbb{D}^2 \\ \downarrow & & \downarrow \\ \mathbb{D}^2 & \hookrightarrow & \mathbb{S}^2 \end{array}$$

One would expect that, in $\mathsf{Ho}(\mathsf{Top})$, replacing all the spaces by homotopy equivalent ones would result in a homotopy equivalent pushout. This is not the case. For example, as the disk is contractible, it can be replaced with $*$ (the single-point space), and then the resulting pushout is $*$ as well:

$$\begin{array}{ccc} \mathbb{S}^1 & \longrightarrow & * \\ \downarrow & & \downarrow \\ * & \longrightarrow & * \end{array}$$

But $*$ is not homotopy equivalent to $\mathbb{S}^2$.

As a fix, mathematicians came up with the **homotopy pushout**.

Definition 4.2.2. The **mapping cylinder** of a map $f : X \to Y$ is the space:

$$M_f = ((X \times [0,1]) \amalg Y)/\sim$$

where $\sim$ is the equivalence relationship generated by $(x, 0) \sim f(x)$ for all $x \in X$, and $\amalg$ denotes disjoint union.

Lemma 4.2.3. *M_f is homotopically equivalent to Y.*

Proof. It is enough to show that Y is a deformation retract of M_f. To do this, we define $F : M_f \times [0,1] \to M_f$ to be constant on the inclusion of Y into M_f, and map $((x,t),s) \in (X \times [0,1]) \times [0,1]$ to $(x, t(1-s))$. This definition is correct because for $(x, 0)$, both parts of the definition concur. Now, we observe that $F(-, 0)$ is the identity of M_f, as $t(1-s) = t$ for $s = 0$. We also see that $F(p, 1)$ belongs in Y for any $p \in M_f$, because $F((x,t), 1) = (x, 0)$, which is identified with $f(x) \in Y$ by $\sim$. Finally, $F(-, 1)$ is the identity function in Y, as we have defined M_f to be constant on Y. □

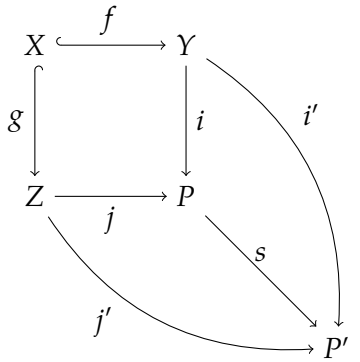
Lemma 4.2.4. *Given any map $f : X \to Y$, there exists an injective map $\tilde{f} : X \to M_f$ that makes the follow diagram commute:*

$$\begin{array}{ccc} X & \xrightarrow{\tilde{f}} & M_f \\ & \searrow^{f} & \downarrow \simeq \\ & & Y \end{array}$$

Proof. We take $\tilde{f}$ to be $F(-,0) : X \to M_f$. □

Definition 4.2.5. The **homotopy pushout** of the diagram $Y \xleftarrow{f} X \xrightarrow{g} Z$ is the topological pushout of the diagram $M_f \xleftarrow{\tilde{f}} X \xrightarrow{\tilde{g}} M_g$.

Equivalently, it is the quotient of $M_f \amalg M_g$ by the relation that identifies each $(x, 1)$ in M_f with each $(x, 1)$ in M_g. As we are replacing spaces with homotopy equivalent ones, and arrows with homotopic ones, we are respecting the diagram from a categorical point of view. The homotopy pushout, once defined, *does* respect homotopy types: if we replace any (or all) of X, Y, or Z with homotopy equivalent objects, then the homotopy pushout will also be homotopy equivalent to the original one. But, as a tradeoff, the homotopy pushout is *not* a pushout in the homotopy category $\mathsf{Ho}(\mathsf{Top})$. This costs us the universal property. Suppose we have two different commuting squares:



The arrow $s : P \to P'$ is not unique up to homotopy, in fact there are infinitely many different choices, in general. Uniqueness is not completely lost, though, it is only weakened. In a categorical pushout, the triple (P', i', j'), determines a unique s. In a homotopy pushout, this is not enough.

Remember that the pushout makes a commutative square. This means that there exists at least one homotopy between $i \circ f$ and $j \circ g$. In a sense, the function s extends one such homotopy into another one $s \circ i \circ f \sim s \circ j \circ g$. If we choose one particular homotopy $H : i \circ f \sim j \circ g$, then $s : P \to P'$ is indeed the unique map extending H in this way.

For example, in the case of the diagram $\mathbb{D}^2 \hookleftarrow \mathbb{S}^1 \hookrightarrow \mathbb{D}^2$ seen before, the homotopy pushout is the sphere $\mathbb{S}^2$. The homotopy in the pushout square relates the inclusion of $\mathbb{S}^1$ in $\mathbb{S}^2$ to itself. But there are many ways to do that, in particular, the homotopy classes of those homotopies form a group isomorphic to $\mathbb{Z}$. On the other hand, if we build a different commutative square to $\mathbb{S}^2$, we have multiple arrows $s : \mathbb{S}^2 \to \mathbb{S}^2$. Again, the mappings of $\mathbb{S}^2$ into $\mathbb{S}^2$ generate a group isomorphic to $\mathbb{Z}$. This shows how the choices of the homotopy in the pushout square correspond to the choices of arrows with the pushout as domain.

With all this information, it is now appropriate to introduce the pushout in homotopy type theory.

Definition 4.2.6. A **pushout** of functions $f : X \to Y$ and $g : X \to Z$ is a higher inductive type

$Y +_X Z$ presented by:

- a function $\mathsf{inr} : Y \to Y +_X Z$,
- a function $\mathsf{inl} : Z \to Y +_X Z$, and
- a dependent function $\mathsf{glue} : \prod_{x:X} \mathsf{inr}(f(x)) = \mathsf{inl}(g(x))$.

The homotopy type theory pushout retains the idea of "gluing": for each x in X, the function glue gives us a path from $f(x)$ to $g(x)$ (in the pushout type, via the inclusions inr and inl). As this definition is type-theoretic, it accepts replacing types with equivalent ones (through the univalence axiom) and preserve the pushout type.

Pushouts allow us to make topological constructions in homotopy type theory. As an example, given a type A, one can build its **suspension** as the pushout of the unique span $\mathbf{1} \leftarrow A \to \mathbf{1}$. Notice what we obtain in this case:

- a function $\mathsf{inr} : \mathbf{1} \to \mathbf{1} +_A \mathbf{1}$,
- a function $\mathsf{inl} : \mathbf{1} \to \mathbf{1} +_A \mathbf{1}$, and
- a dependent function $\mathsf{glue} : \prod_{a:A} \mathsf{inr}(\star) = \mathsf{inl}(\star)$.

Here inr and inl are just inclusions, they do not contribute anything meaningful. But glue is interesting: instead of literally gluing $\mathbf{1}$ to $\mathbf{1}$ via A, we just use A to parametrize the paths, but ignore its points $(a : A)$ altogether. The natural functoriality of functions in homotopy type theory will guarantee that the "copy" of A of the suspension will retain the homotopical structure of A.

As with any higher inductive type, we can form its elimination principles. For example, the recursion principle states that, to define a function $s : Y +_X Z \to D$ into another type D, we just have to provide:

- for each $y : Y$, the value of $s(\mathsf{inr}(y)) : D$,
- for each $z : Z$, the value of $s(\mathsf{inl}(z)) : D$, and
- for each $x : X$, the value of $\mathsf{ap}_s(\mathsf{glue}(x)) : s(\mathsf{inr}(f(x))) = s(\mathsf{inl}(g(x)))$.

From this principle, one can deduce a uniqueness principle, which states that given two functions $s, s' : Y +_X Z \to D$ that coincide on the values above, then $s = s'$. This uniqueness principle corresponds to our previous statement about choosing a particular homotopy in a homotopy pushout. The higher inductive type is given with a constructor glue, which is a choice of homotopy in the square. Observe that s is unique whenever it also respects the equalities given by glue.

In the following section we discuss how cells are attached via pushouts.

4.3 Constructions Using Pushouts

The common procedure to attach cells in homotopy type theory is known as the "hub and spokes" technique. One possible interpretation is as follows:

Definition 4.3.1. Suppose given a type X and a function $f : \mathbb{S}^{n-1} \to X$ with $n > 0$. We define the **attachment of an *n*-cell** to X via f as the higher inductive type $\hat{X}$ defined by:

- an inclusion function $i : X \to \hat{X}$,

- a "hub" point $h : \hat{X}$, and
- a family of "spokes" $s : \prod_{(p:\mathbb{S}^{n-1})} f(x) = h$.

We visualize the attachment as taking X and gluing an n-disk along the image of f.

Most often, we do not use this definition but rather apply the concept of attachment *ad hoc*. For example, this can be done through the following pushout:

$$\begin{array}{ccc} \mathbb{S}^{n-1} & \xrightarrow{f} & X \\ \downarrow & & \downarrow{\scriptstyle \mathsf{inr}} \\ \mathbf{1} & \xrightarrow[\mathsf{inl}]{} & P \end{array}$$

which attaches an n-cell via the path family $\mathsf{glue} : \prod_{(p:\mathbb{S}^{n-1})} \mathsf{inr}(f(x)) = \mathsf{inl}(\star)$, with $\mathsf{inl}(\star)$ acting as the hub.

We will now see how to build the real projective spaces in homotopy type theory. Using pushouts, we can build $\mathbb{R}\mathsf{P}^{n+1}$ from $\mathbb{R}\mathsf{P}^n$ as such:

$$\begin{array}{ccc} \mathbb{S}^n & \longrightarrow & \mathbf{1} \\ {\scriptstyle \alpha_n}\downarrow & & \downarrow \\ \mathbb{R}\mathsf{P}^n & \longrightarrow & \mathbb{R}\mathsf{P}^{n+1} \end{array}$$

This amounts to attaching an $n+1$ cell to $\mathbb{R}\mathsf{P}^n$, as we said before. The function α_n, which states *how* to attach the cell, is the canonical covering function. Unfortunately, this is hard to express in the language of homotopy type theory.

We can take a closer look at that $\mathbb{S}^n$ in the pushout diagram. We have just seen that the sphere is a covering space of $\mathbb{R}\mathsf{P}^n$. This suggests rewriting the diagram as:

$$\begin{array}{ccc} \sum_{(x:\mathbb{R}\mathsf{P}^n)} \mathsf{cov}^n(x) & \longrightarrow & \mathbf{1} \\ \downarrow & & \downarrow \\ \mathbb{R}\mathsf{P}^n & \longrightarrow & \mathbb{R}\mathsf{P}^{n+1} \end{array}$$

where $\mathsf{cov}^n(x)$ is the fiber of $x : \mathbb{R}\mathsf{P}^n$. Now we have to define $\mathbb{R}\mathsf{P}^n$ and cov^n inductively on n.

As the fiber of a covering space, we would usually define cov to be of type $\mathsf{cov}^n : \mathbb{R}\mathsf{P}^n \to \mathit{U}$. In this case, though, we need a little bit more precision, and so we first define the "subuniverse" $\mathit{U}_{\mathbb{S}^0}$ of 2-element sets, so that we can have $\mathsf{cov}^n : \mathbb{R}\mathsf{P}^n \to \mathit{U}_{\mathbb{S}^0}$. As there are no subtypes in homotopy type theory, this is actually a fibration itself:

Definition 4.3.2. The **universe of 2-sets** $\mathit{U}_{\mathbb{S}^0}$ consists of types together with a proof that they are equal to $\mathbb{S}^0$, i.e., $\sum_{(A:\mathit{U})} \|A = \mathbb{S}^0\|_{-1}$.

We take $\mathbb{S}^0$ as the inductive type with two constructors, N and S.

Luckily, the fibers are mere propositions, so we can omit them for brevity without changing the constructions, and treat $\mathcal{U}_{\mathbb{S}^0}$ like a universe of types. Similarly, we consider $\mathbb{S}^0$ to be a pointed type, with center N, but we will not write it explicitly throughout the text.

Consider the map $\mathsf{encode} : \prod_{(A:\mathcal{U}_{\mathbb{S}^0})}(\mathbb{S}^0 = A) \to A$ given by taking the center of $\mathbb{S}^0$ to the element that the chosen equality transports it to. In other words, $\mathsf{encode}(A, p) = \mathsf{idtoeqv}(p)(\mathsf{N})$, where $\mathsf{idtoeqv}$ is the equivalence induced by the equality of types.

Lemma 4.3.3. *The function* $\mathsf{encode} : \prod_{(A:\mathcal{U}_{\mathbb{S}^0})}(\mathbb{S}^0 = A) \to A$ *given by* $\mathsf{encode}(A, p) = \mathsf{idtoeqv}(p)(\mathsf{N})$ *is an equivalence.*

This encode, like the others we have seen, tries to give a combinatorial interpretation of an equality type, in this case $\mathbb{S}^0 = A$. To correctly interpret this function, we disregard the first argument $(A : \mathcal{U}_{\mathbb{S}^0})$, as it is only necessary in order to introduce the path p. Then we see the actual meaning of the lemma: every element of a 2-set A corresponds uniquely with an equality between A and the canonical pointed 2-set $\mathbb{S}^0$. After all, there are only two ways to identify $\mathbb{S}^0$ with A. This identification is uniquely determined by the image of N through the path.

The proof of the lemma is given in Buchholtz and Rijke (2017), Corollary II.6. It is slightly technical and requires introducing results about pointed types that are out of scope, so we skip directly to building $\mathbb{R}\mathsf{P}^n$ and cov^n:

- For the base case $(n = -1)$, we take $\mathbb{R}\mathsf{P}^{-1} :\equiv \mathbf{0}$, the empty type. Then, there is only one candidate for cov^{-1}, which is the only function of type $\mathbf{0} \to \mathcal{U}_{\mathbb{S}^0}$.
- For the inductive case, assume $\mathbb{R}\mathsf{P}^n$ and cov^n are defined. Then $\mathbb{R}\mathsf{P}^{n+1}$ is defined as the following pushout:

$$\begin{array}{ccc} \sum_{(x:\mathbb{R}\mathsf{P}^n)} \mathsf{cov}^n(x) & \longrightarrow & \mathbf{1} \\ \Big\downarrow{\scriptstyle \mathsf{pr}_1} & & \Big\downarrow \\ \mathbb{R}\mathsf{P}^n & \longrightarrow & \mathbb{R}\mathsf{P}^{n+1} \end{array}$$

  To define $\mathsf{cov}^{n+1} : \mathbb{R}\mathsf{P}^{n+1} \to \mathcal{U}_{\mathbb{S}^0}$, consider the following diagram:

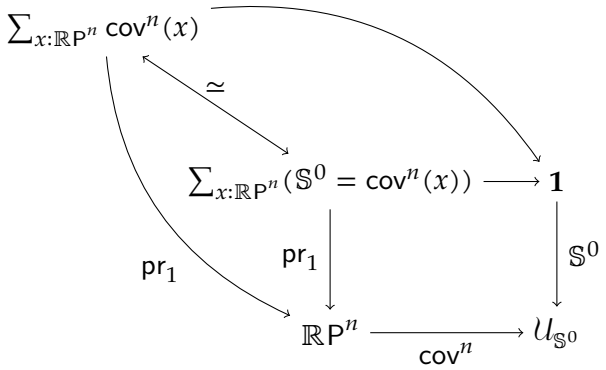
$$\begin{array}{ccc} \sum_{(x:\mathbb{R}\mathsf{P}^n)} \mathsf{cov}^n(x) & \longrightarrow & \mathbf{1} \\ \Big\downarrow{\scriptstyle \mathsf{pr}_1} & & \Big\downarrow{\scriptstyle \mathbb{S}^0} \\ \mathbb{R}\mathsf{P}^n & \xrightarrow[\mathsf{cov}^n]{} & \mathcal{U}_{\mathbb{S}^0} \end{array}$$

  where $\mathbb{S}^0$ represents the function taking $\star$ to $\mathbb{S}^0$. If we can prove that the square commutes, then, by the universal property of the pushout, there has to exist a function from $\mathbb{R}\mathsf{P}^{n+1}$ to $\mathcal{U}_{\mathbb{S}^0}$, which we will take to be cov^{n+1}.

For every $x : \mathbb{R}\mathsf{P}^n$, we have the equivalence $\mathsf{encode}(\mathsf{cov}^n(x)) : (\mathbb{S}^0 = \mathsf{cov}^n(x)) \simeq \mathsf{cov}^n(x)$ given by the lemma. Theorem 4.7.7 of The Univalent Foundations Program (2013) induces an equivalence

$$\sum_{x:\mathbb{R}\mathsf{P}^n} \mathsf{cov}^n(x) \simeq \sum_{x:\mathbb{R}\mathsf{P}^n} (\mathbb{S}^0 = \mathsf{cov}^n(x))$$

So we have the following diagram:



The inner square is an instance of what is known as a pullback square, because its upper left corner is (equivalent to) the double sum type over equalities of the top right and bottom left corners, $\sum_{(y:\mathbf{1})} \sum_{(x:\mathbb{R}\mathsf{P}^n)} (\mathbb{S}^0 = \mathsf{cov}^n(x))$. These are dual to the pushout squares and are also commutative (The Univalent Foundations Program 2013, Exercise 2.11). Hence, the outer square is also commutative, as we wanted to prove.

4.4 Construction as a Higher Inductive Type

We begin by highlighting the fact that in homotopy type theory we use the word homotopy in two different senses. The first, like the one that appears in the definition of a pushout, matches the classical notion of "a path between functions". On the other hand, following the analogy between higher identity types and homotopy types, we can consider the terms of a type as points, their equalities as paths, and the equalities between those as homotopies. To distinguish them, we will call the first kind *function homotopies* and the second kind *path homotopies*.

In the literature of homotopy type theory, cell complexes are mainly built using pushouts. The function homotopy in the definition of the pushout plays the role of the higher cell.

In a few cases, the other notion of homotopy is used. For example, the higher inductive type definitions of the higher spheres are built like this: we provide a 0-cell and an n-cell, in the form of a higher path.

With this in mind, it was hypothesized that path homotopies could be used in other spaces. In particular, the idea appeared when trying to see how to obtain the fundamental group of the real projective plane. If one distills the (topological) idea of what a projective plane is, it

can be seen as a 1-cell, together with a 2-cell that reverses it. We can express this as a higher inductive type:

$$\begin{aligned} X_0 &: X \\ X_1 &: X_0 = X_0 \\ X_2 &: X_1 = X_1^{-1} \end{aligned}$$

This construction is conceptually simpler than the one with pushouts. But not only that: it also allows an easy construction of its fundamental group.

We recall that the fundamental group is defined as the 0-truncation of the path space over the base point $(\|X_0 = X_0\|_0)$.

Theorem 4.4.1. *The fundamental group of the space X is $\mathbb{Z}/2\mathbb{Z}$.*

Proof. Observe that, as this is a higher inductive type, it is freely generated by these constructors. We want to know which paths exist from X_0 to itself. If we had not provided the X_2 constructor, we would have the circle, and thus the fundamental group would be $\mathbb{Z}$, as we have already seen. But the next constructor X_2 is a path homotopy between X_1 and itself. By adding another constructor, we can only further constrain the fundamental group, i.e., add new relationships. In this case, we are stating that $X_1 = X_1^{-1}$. This tells us that the group of loops over X_0 is freely generated by X_1 and the relationship X_2 of type $X_1 = X_1^{-1}$, or, in other words, $\langle X_1 \mid X_1 = X_1^{-1} \rangle$, which is isomorphic to $\mathbb{Z}/2\mathbb{Z}$, the cyclic group of order two. □

This clearly suits our conception of the fundamental group of the projective plane as seen in sec. 4.1. What we called γ in there is represented by X_1 here. And the fact that $\gamma \cdot \gamma$ is the constant path can be seen here as $X_1 \cdot X_1 = \mathsf{refl}_{X_0}$, proven by X_2.

So, we pose the question: is X equivalent to $\mathbb{R}\mathsf{P}^2$?

Following the definition of equivalence, we should find two functions, $\psi : X \to \mathbb{R}\mathsf{P}^2$ and $\varphi : \mathbb{R}\mathsf{P}^2 \to X$, that are mutual quasi-inverses, or, in other words, such that $\psi \circ \varphi \sim \mathsf{id}_{\mathbb{R}\mathsf{P}^2}$ and $\varphi \circ \psi \sim \mathsf{id}_X$.

We start by defining a candidate for φ. This in itself posed a considerable challenge, as it required having a very clear understanding of how 2-paths work that might not be intuitive at first, in particular regarding induction.

As $\mathbb{R}\mathsf{P}^1 = \mathbb{S}^1$ (Buchholtz and Rijke 2017, Example III.3), the type $\mathbb{R}\mathsf{P}^2$ can be seen as the pushout of $\mathbb{S}^1 \leftarrow \sum_{x:\mathbb{S}^1} \mathsf{cov}^1(x) \to \mathbf{1}$.

Definition 4.4.2. We define $\varphi : \mathbb{R}\mathsf{P}^2 \to X$ using the recursion principle of $\mathbb{R}\mathsf{P}^2$.

- For each $a : \mathbb{S}^1$, the value of $\varphi(\mathsf{inl}(a)) : X$.

  We choose to embed $\mathbb{S}^1$ into X by sending base to X_0 and loop to X_1.

- For each $b : \mathbf{1}$, the value of $\varphi(\mathsf{inr}(b)) : X$.

  By the recursion principle of $\mathbf{1}$, it is enough to define the image of $\star$. We pick the base point X_0.

- For each $c : \sum_{(x:\mathbb{S}^1)} \mathsf{cov}^1(x)$, the value of $\mathsf{ap}_\varphi(\mathsf{glue}(c)) : \varphi(\mathsf{inl}(f(c))) = \varphi(\mathsf{inr}(g(c)))$.

  For this pushout, f is the projection onto the first component of the dependent pair, and g is the only function into $\mathbf{1}$. So, what we are looking for is, for every pair (x, y) of $\sum_{(x:\mathbb{S}^1)} \mathsf{cov}^1(x)$, a way to apply φ to a path of type $\mathsf{inl}(x) = X_0$, where $\mathsf{inl}(x)$ is the inclusion of x into X. Notice how, for each $x : \mathbb{S}^1$, there are two different pairs (x, y) of type $\sum_{(x:\mathbb{S}^1)} \mathsf{cov}^1(x)$, as cov^1 is a double cover of $\mathbb{S}^1$.

  We use the induction principle for Σ types, which essentially states that we have to provide a value for each possible pair. Then, we do induction on each component. This leaves us with four values to provide:

  - $\mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{N})) : X_0 = X_0$,
  - $\mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{S})) : X_0 = X_0$,
  - $\mathsf{ap}_s(\mathsf{ap}_{\mathsf{glue}}(\mathsf{loop}, \mathsf{N})) : \mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{N})) = \mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{S}))$, and
  - $\mathsf{ap}_s(\mathsf{ap}_{\mathsf{glue}}(\mathsf{loop}, \mathsf{S})) : \mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{S})) = \mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{N}))$.

  We choose $\mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{N})) := \mathsf{refl}_{X_0}$ and $\mathsf{ap}_s(\mathsf{glue}(\mathsf{base}, \mathsf{N})) := X_1 \cdot X_1$, so that the other two have types:

  - $\mathsf{ap}_s(\mathsf{ap}_{\mathsf{glue}}(\mathsf{loop}, \mathsf{N})) : \mathsf{refl}_{X_0} = X_1 \cdot X_1$, and
  - $\mathsf{ap}_s(\mathsf{ap}_{\mathsf{glue}}(\mathsf{loop}, \mathsf{S})) : X_1 \cdot X_1 = \mathsf{refl}_{X_0}$.

  We can then choose X_2^{-1} and X_2 for those.

For $\psi : X \to \mathbb{R}\mathsf{P}^2$, we try using the recursion principle. We have to provide a point $\psi(X_0)$ for X_0, a path $\psi(X_0) = \psi(X_0)$ for X_1, and a homotopy $\psi(X_1) = \psi(X_1)^{-1}$.

- For $\psi(X_0)$ we choose $\mathsf{inl}(\mathsf{base})$.
- For $\psi(X_1)$, $\mathsf{ap}_{\mathsf{inl}}(\mathsf{loop})$.

Unfortunately, the construction of $\psi(X_2)$ requires providing a proof that

$$\mathsf{ap}_{\mathsf{inl}}(\mathsf{loop}) \cdot \mathsf{ap}_{\mathsf{inl}}(\mathsf{loop}) = \mathsf{refl}_{\mathsf{inl}(\mathsf{base})},$$

which practically amounts to calculating the fundamental group of $\mathbb{R}\mathsf{P}^2$.

If we could define ψ like this, then proving they are mutual quasi-inverses amounts to once again use the eliminators of each type, but in a trickier way. As the induction principle tells us that a function is determined by a series of parameters, we can prove that $\varphi \circ \psi$ and $\psi \circ \varphi$ are equal to the corresponding identity functions by just showing they agree with them on these parameters alone.

In the case of $\varphi \circ \psi : X \to X$, we would use the elimination principle of X, whereas for $\psi \circ \varphi : \mathbb{R}\mathsf{P}^2 \to \mathbb{R}\mathsf{P}^2$, we would use the elimination principle of $\mathbb{R}\mathsf{P}^2$.

The difficulties in defining the function ψ are a clear display of the complexity of higher inductive types that require a second order eliminator (i.e., that have higher order paths), and explain the reason why these constructions are not thoroughly used. Nonetheless, we believe that building them and proving them correct can be useful.

Conclusions

We have high hopes for the synthetic projective plane we are postulating. All the clues have been pointing to this construction being correct, and perhaps more important, being "in the style" of homotopy type theory.

The equivalences between the pushout and synthetic versions take the "skeleton" (points and 1-paths) to the corresponding items with little trouble. Each of the two implementations has a "special" surface: in the case of the pushout, the path family glue, and in the case of the higher inductive type, the constructor X_2. The complexity and defining property of the equivalences is in describing correctly how these surfaces correspond to each other. These surfaces are actually the generators of the second order homotopy group of each of the spaces, and in fact also spawn all the higher homotopy structure. We also had postulated a possible implementation of higher projective spaces following this technique of attaching "reversing" paths, but this would add extra higher homotopical structure, which would not work.

Nonetheless, we will still eagerly try to prove the equivalence for $\mathbb{R}\mathrm{P}^2$. The conversion from pushout to higher inductive type in this manner is proposed by the book of The Univalent Foundations Program (2013) itself, but we have not been able to find any complete proof of this style. Hence, we believe that publishing the complete version of the proof would actually contribute value to the homotopy type theory community.

One of our objectives was to use Agda as an aid for learning homotopy type theory. Using a machine turns out to be of big use: the compiler is blunt and does not forgive any mistake. The learning path with a proof assistant can feel like pushing a boulder up a hill, but the error messages at every wrong step help correct misguided intuitions. Although it did not help in producing complete results, Agda is useful for making "enquiries" about (homotopy) type theory. It is particularly good for clearing up how higher paths should be built from lower ones, as it does not allow to abuse the language, hence avoiding conceptual or syntactical mistakes.

Some people may wonder what use does a proof checker have if the job of writing the proof relies on the mathematician. After all, it might just seem like doing double work for nothing.

The immediate benefit is that verification helps catch mistakes that might be otherwise hard to notice by humans in very complex settings. Some people might have philosophical reservations about computers proving things that people cannot. We propose seeing the computer as a mathematical object: one first proves the computer to be correct (this entails everything from the processing unit up to the proof assistant), and then uses it to prove further things.

The long-term answer is that formalization of proofs is an indispensable step for further improvements, like proof assistants (programs that help the mathematician by proposing ways of filling holes in proofs), which already exist, and eventually automated proof development, which would consist of the computer being capable of deducing valid theorems from a set of hypotheses (which already shows up in logic programming).

References

Brunerie, Guillaume, Kuen-Bang Hou (Favonia), Evan Cavallo, Tim Baumann, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, and others. n.d. "Homotopy Type Theory in Agda." Accessed June 14, 2020. https://github.com/HoTT/HoTT-Agda.

Buchholtz, Ulrik, and Egbert Rijke. 2017. "The Real Projective Spaces in Homotopy Type Theory." *arXiv:1704.05770* [*Math*], April. http://arxiv.org/abs/1704.05770.

Hatcher, Allen. 2000. *Algebraic Topology*. Cambridge: Cambridge Univ. Press. https://cds.cern.ch/record/478079.

Licata, Dan. 2011. "Running Circles Around (in) Your Proof Assistant; or, Quotients That Compute." *Homotopy Type Theory*. https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/.

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