MASTER'S DEGREE THESIS

Master of Science in Advanced Mathematics and Mathematical Engineering

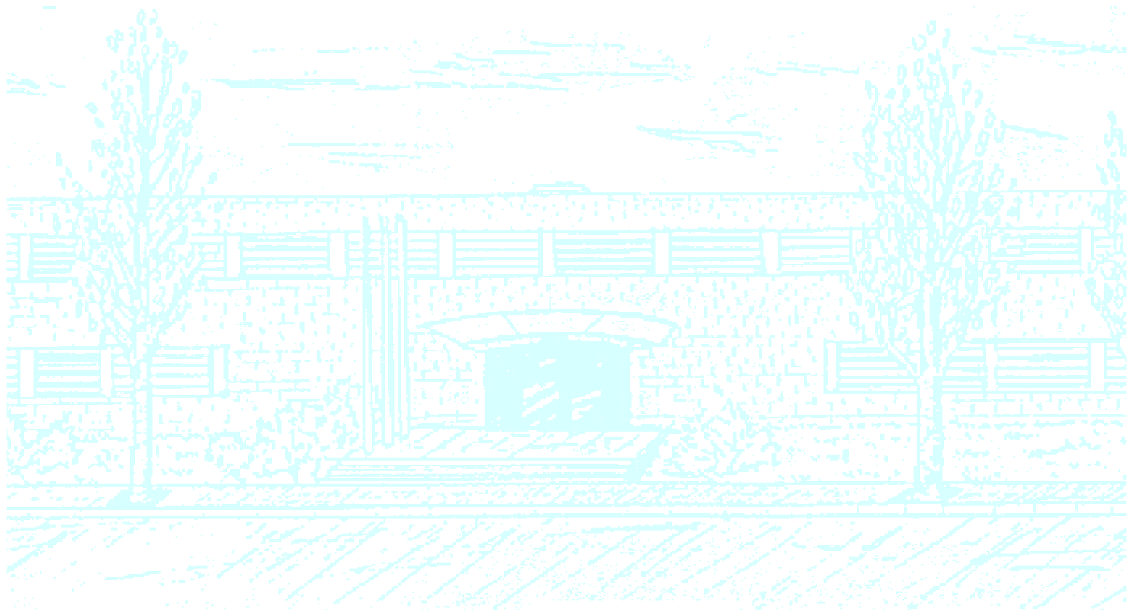
Title: Logic and proof assistants

Author: Tomás Martínez Coronado

Advisor: Enric Ventura Capell

Department: Matemàtiques

Academic year: 2015 - 2016



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

An introduction to Homotopy Type Theory

Tomás Martínez Coronado

June 26, 2016

Chapter 1

Introduction

We give a short introduction first to the philosophical motivation of Intuitionistic Type Theories such as the one presented in this text, and then we give some considerations about this text and the structure we have followed.

Philosophy

It is hard to overestimate the impact of the problem of Foundations of Mathematics in 20th century philosophy. For instance, arguably the most influential thinker in the last century, Ludwig Wittgenstein, began its philosophical career inspired by the works of Frege and Russell on the subject —although, to be fair, he quickly changed his mind and quit working about it.

Mathematics have been seen, through History, as the paradigma of absolute knowledge: Locke, in his *Essay concerning Human Understanding*, had already stated that Mathematics and Theology (*sic!*) didn't suffer the epistemological problems of empirical sciences: they were, each one in its very own way, provable knowledge. But neither Mathematics nor Theology had a good time trying to explain its own inconsistencies in the second half of the 19th century, the latter due in particular to the works of Charles Darwin.

The problem of the Foundations of Mathematics had been the elephant in the room for quite a long time: the necessity of relying on axioms and so-called "evidences" in order to state the invulnerability of the Mathematical system had already been criticized by William Frend in the late 18th century, and the incredible Mathematical works of the 19th century, the arrival of new geometries which were "as consistent as Euclidean geometry", and a bunch of well-known paradoxes derived of the misuse of the concept of infinity had shown the necessity of constructing a logical structure complete enough to sustain the entire Mathematical building. That is the beginning of half a century of interplay between Logic and Mathematics which would reach its cusp in Gödel's wonderful theorems.

The first serious attempt comes from Georg Cantor, the father of modern Set Theory in the late 19th century. Cantor's work has been enormously influential in Mathematics, thanks to the support that David Hilbert gave him, in contrast to how Poincaré excoriated his Set Theory and the influence of Logic in Mathematics. But to be honest, Cantorian naive Set Theory was full of inconsistencies, the most famous one being Russell's Paradox, which lead to his and Whitehead's masterwork *Principia Mathematica* and the first Type Theory.

In parallel to the efforts of Frege, Russell, Peano, Hilbert and many others to give a formal logical basis for Mathematics, Brouwer developed his own philosophical views on the topic: the philosophical school now called intuitionism. Loosely speaking, the fundamental idea of intuitionism is that, as mental constructs, Mathematical concepts have to be explicitly constructed inside Mathematical formalism. In this sense, we have what we call the Brouwer-Heyting-Kolmogorov interpretation of proof semantics, about which we will talk a little along this text. Under this interpretation, logical connectives such as "and", "or", "implies" "for all", etc., have a very strong meaning: to give a proof of "A and B" is to give a proof of A and to give a

proof of B; to give a proof of "A implies B" is to give an algorithm or computable function that transforms each proof of A into a proof of B, and so on.

Its constructive point of view —indeed, intuitionist Mathematics are often called constructive Mathematics— has been fundamental in Computer Science and the development of Proof Assistants. On one hand, all programming languages include types to some extent —and some of them, such a HASKELL, have a very strong embedded Type Theory. On the other, thanks to the Curry-Howard interpretation, or *propositions-as-types*, we are able to have a very computational —and hence constructive— approach to the task of proving a theorem: it suffices to identify this theorem with a type, and then construct an inhabitant of this type.

Finally, we wish to remark that, as Type Theory is strong enough to codify Arithmetic, it is also vulnerable to Gödel's Incompleteness Theorems. And whereof one cannot speak, thereof one must be silent.

Univalent Foundations

The Univalent Foundations Program belongs to the interplay between Type Theory, Set Theory and Category Theory. The Type Theory presented here is due to Martin-Löf intensional Type Theory, in which he introduced the *identity type* of a type A and two objects $a, b : A$, $\mathsf{Id}_A(a, b)$. This is the type of proofs of equality $a = b$ in A. We will regard these proofs topologically, seeing them as paths between two points a, b in a space A.

Thanks to the work of Vladimir Voevodsky and the Univalent Foundations Program, this theory is extended with the *Axiom of Univalence* (Voevodsky 2009). This aims to generalize in a uniform way the concepts of logical equivalence, set bijection and categorical equivalence of groupoids, as we will see. In particular, the Axiom of Univalence says that, given two equivalent types A and B (and we will see what does that mean along this text), there is a proof of equality between them. In other words, we already knew that two equal types are equivalent, but now we regard two equivalent types as equal. Thus, using the transport of structures along equality proofs (Bourbaki, 1957), which we will call $\mathsf{transport}$, the Univalent Foundations Program and its Homotopy Type Theory aims to give an alternative to ZF solution to the problem of Foundations of Mathematics.

Homotopy Type Theory has also proven to be very susceptible to be modeled by proof assistants. However, there are still problems in the implementation of higher inductive types that still oppose to the full development of this new theory into computer based Mathematics.

This text

This text is just an exposition of the basic facts of Homotopy Type Theory, and nothing in it is original. My own interest in the philosophy of Mathematics motivates the choice of the topic, since this seems to be a very interesting alternative to the classical approach.

The theorems, lemmas and corollaries in this text come from the Univalent Foundation Program's main work, *Homotopy Type Theory* [13], complemented with some articles I have found interesting or clarifying. A lot of them are theorem and lemmas of the book, and a few examples or exercises. The proof of the results given here might not be exactly equal, but they are surely homotopic: I have included, mainly for my own understanding, the details in those parts of the book that seemed most obscure to me. I think it is a fairly self-contained introduction to the topic, except maybe for an introduction to Category Theory which is not explicitly given, and some points left obscure when the aim of this introduction lay far enough of the necessary theory to approach them.

The basic knowledge of Category Theory needed to understand the book came mainly from two sources: the classical reference, *Categories for the working Mathematician* [5] by Saunders MacLane, and the chapter dedicated to Category Theory in *An Introduction to Homological Algebra* [4] by Peter Hilton and Urs Stammbach.

The structure of this work is also nearly the same as the structure of the Univalent Foundations Program, because of course it seems to me to be the most intuitive structure: we begin by giving an introduction to

Martyn-Löf's intuitionistic Type Theory and the propositions-as-types interpretation and Heyting's semantics, followed by the homotopical interpretation and the Axiom of Univalence which lead to the development of this new theory. Once introduced these, we proceed to model classical Set Theory and Predicates Logic into this type theory in an arguably more convenient way than the naive propositions-as-types interpretation. The final two chapters are more Mathematical, and we introduce inductive and higher-inductive types that model topological objects, in order to finally being able to prove the Seifert-Van Kampen's theorem, whose proof (using Homotopy Type Theory) was first given by Michael Shulman in [10].

Chapter 2

Type Theory

Homotopic Type Theory is a language created to codify the foundations of Mathematics. In this sense, it is an alternative point of view to classic set-theory, as the one proposed in Zermelo-Fraenkel's axioms.

We remark that set-theory in classical Mathematics has two layers: set-theory iself, and predicate logic. Predicate logic allows us to deduce theorems about sets. In opposition to it, we find Type Theory: it is its own deductive system, in which we only have one basic concept: *types*.

In Type Theory each term has a type, and an expression $a : A$ where a is a term and A is a type is called a *type judgement*, and can be read as "term a has type A." As we will see, we can identify propositions —statements susceptible of being validated, negated, etc.— with types. In particular, if a type P represents a proposition, a judgement $p : P$ corresponds to the fact that P is a *true* proposition, namely that it has a "proof", or *witness*; hence, the classical process of proving a theorem is identified with the mathematical process of *constructing an object*.

In classical set-theory, the fundamental judgement is "P has a proof" (which, as we may note, is in a different space of reference than P itself); analogously, in Type Theory the fundamental judgement is "$p : P$", the term (we will sometimes call it *element*, or *point*) p has type P. In particular, even if in set-theory "$p \in P$" is a proposition, "$p : P$" is a judgement such as "P has a proof" was: i.e., a meta-theoretical statement.

In this chapter we will give an informal introduction to Type Theory (being informal in this introduction may allow us, humans, to actually *understand* what we are doing, whereas extremely formal Type Theory seems very hard to approach). In particular, we will introduce *type-formers*: some ways to build new types by using pre-existing ones. Type Theory presented in this chapter is entirely composed by *rules* —which allow us to derive new judgements from old ones— and not a single *axiom* —judgements imposed outside the theory—, although in other chapters we will consider some of them.

Morgenstern, Abendstern

In this chapter we will present a version of Martin-Löf's intuitionistic Type Theory as it is presented in the book *Homotopy Type Theory*. It is an *intensional* theory, which means that we will distinguish between *judgemental* equalities and *propositional* equalities, which we will note as $x \equiv y$ and $x = y$, respectively.

In Frege's terminology, introduced in his *Über Sinn und Bedeutung* ("On sense and reference"), we might think about this distinction as the one existing between *sense* and *reference*: using Frege's famous example, even if expressions "morning star" and "evening star" *refer* to the same object, the planet Venus, they are defined differently and so they *mean* different things. Loosely speaking, this distinction between a "definitional" equality —an equality showed by simply developing an expression— and a propositional equality will be fundamental for the presentation of Homotopy Type Theory, where we will interpret expressions of the form "$x = y$" as paths between $x, y : A$. This idea is also found, although not implemented in the same way, in proof assistants such as Coq and Agda.

An alternative to the intensional point of view is the *extensional* one, where propositional equality and judgemental equality are treated as one. Indeed, it is usually how we treat them in classical Mathematics, but here we will treat proofs as actual Mathematical objects by using the well-known propositions-as-types interpretations, often called Curry-Howard interpretation, in which "a proposition is the type of all its proofs." A philosophical defense of this extensional point of view might be implicitly found, I think, in Barry Mazur's wonderful article "When is one thing equal to some other thing?."

In any case, an extensional Homotopy Type Theory would not be an useful approach, since it would model discrete topological spaces, where every path is trivial and it is not, hence, interesting for us: we would be left only with judgemental equalities, with no higher structure at all.

2.1 Universes

Type Theory presented here has *universes*, but there are some which do not. A universe is a type $\mathcal{U}$ such that its elements are types. After Russell, we know that it is not a very good idea to think about a universe $\mathcal{U}_\infty$ such that it contains every type and, sigh, in particular $\mathcal{U}_\infty : \mathcal{U}_\infty$. Indeed, Thierry Coquand in [1] was able to translate a similar paradox to Russell's in such a Type Theory.

Our solution is essentially due to Russell himself: we introduce a universe hierarchy as follows

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

such that if $x : \mathcal{U}_i$, then $x : \mathcal{U}_{i+1}$: ours is a cumulative hierarchy. Usually, we will not write the subindex i, and so we will only work with $\mathcal{U}$ when context is explicit enough.

In order to model a collection of types in a universe, indexated by a type A, we will write a function $A \to \mathcal{U}$ (we will introduce functions next to this section).

Remark 1. The fact that we use natural numbers in the indexes does not imply that we can build a function $\lambda(i : \mathbb{N}).\mathcal{U}_i$, since there is no universe able to be its codomain.

2.2 Functions

Suppose given types $A, B : \mathcal{U}$. We can always build the type $A \to B$ of the functions of domain A and codomain B. Unlike their analogues in Set Theory, functions in Type Theory are a primitive concept: we introduce them with rules about how can we interact with them and how can we consider the equalities they induce.

Given a function $f : A \to B$ and an element $a : A$, we can apply the function to a and obtain an element $f(a) : B$, called the *value of f in a*.

We can construct elements of $A \to B$ in two ways: by giving the expression of the function,

$$f(x) :\equiv \Phi$$

where the symbol $:\equiv$ means that we are defining $f(x)$ as the expression Φ, which might depend on $x : A$, or by giving its λ-abstraction, without naming it

$$\lambda(x : A).\Phi : A \to B$$

Conventionally, the scope of the symbol λ is everything that lies to its right if there are no parenthesis involved (as will be the case with the symbols Π and Σ). Usually, we do not write the type of the variable, and so the expression becomes

$$\lambda x.\Phi : A \to B$$

The *computation rule* of the functions type, also called *β-reduction*, is the following judgemental equality:

$$(\lambda x.\Phi)(a) \equiv \Phi'$$

where Φ' is the expression Φ where every occurrence of x has been replaced by a.

Since every function can be introduced in two different ways, we also have a *uniqueness principle*, or *η-conversion*:

$$f \equiv \lambda x.f(x)$$

which can be understood as the fact that a function is entirely determined by its values.

In order to define functions of many variables, we can use the cartesian products of types (which will be introduced very soon), but we usually use the mechanism known as *curryfication* —its name due to logician Haskell Curry[1].

Given types $A, B, C : \mathcal{U}$ and f taking variables in A and in B, we can consider the application $f : A \to B \to C$ —we will, by convention, use right-associativity, and thus $A \to B \to C$ is the same as saying $A \to (B \to C)$. Hence, f is a function of domain A and codomain $B \to C$. In this case, we write

$$f(a, b) :\equiv f(a)(b)$$

in order to avoid parentheses proliferation. We could have also used λ-abstractions: if $f(a, b) :\equiv \Phi$, where Φ is an expression which can depend on a and b, we might write

$$\lambda a.(\lambda b.\Phi)$$

or simply $\lambda a.\lambda b.\Phi$.

We might also use notation $f(-)$ or $x \mapsto f(x)$ for $\lambda x.f(x)$, and $f(-, -)$ for $\lambda x.\lambda y.f(x, y)$.

2.2.1 Dependent functions

Type theory also offers the possibility of constructing types of dependent functions, "functions" whose codomain can vary depending on which element of the domain they are applied to. Formally, given a type family $B : A \to \mathcal{U}$ indexed by A, the type

$$\prod_{x:A} B(x)$$

is the type of dependent functions of A to the family B. Each element $f : \prod_{x:A} B(x)$ is a function such that $f(x) : B(x)$ for every $x : A$. If B is a constant family, then $\prod_{x:A} B \equiv A \to B$.

A very important example of dependent function is the identity function $\mathsf{id} : \prod_{A:\mathcal{U}} A \to A$, defined as $\mathsf{id} :\equiv \lambda A : \mathcal{U}.\lambda(x : A).x$. We will usually write the arguments of such a function as subindices, and so function $\mathsf{id}(A) : A \to A$ becomes id_A.

2.3 Product type

Given $A, B : \mathcal{U}$ two types, we can consider the *cartesian product* $A \times B : \mathcal{U}$. We want the elements of $A \times B$ to be couples (a, b) where $a : A$ and $b : B$, but as we did with functions we will define the elements of $A \times B$ as primitive terms. We also introduce the type $\mathbf{1} : \mathcal{U}$, which we will want to be inhabited by a unique element, $*$.

Given $a : A, b : B$, we can always build an element $(a, b) : A \times B$. We will show later on that, indeed, any element of $A \times B$ is of this form.

To be able to define functions over a product we use currying: given $f : A \times B \to C$ we consider a function $g : A \to B \to C$ in such a way that

$$f((a, b)) \equiv g(a)(b)$$

which gives us a computation rule for product types.

[1]Who also gives its name to the functional programming language HASKELL.

Remark 2. From a categorical perspective, we can consider currying as an adjonction phenomenon between the product functor and the exponential: $\mathcal{U}(A \times B, C) \to \mathcal{U}(A, C^B)$.

We consider that a function $f : A \times B \to C$ is well defined when we have defined all the values on the couples. Notice that we have implicitly constructed a function

$$\mathsf{rec}_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to (A \times B \to C)$$

such that

$$\mathsf{rec}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b)$$

We call such a function the *recursor* of $A \times B$ —through this text we will see a lot of recursors, each one associated to a type-former. The name “recursor” is not a very good choice in this case, since there is no recursion taking place: it is inherited from other types such as $\mathbb{N}$, whose recursor will be truly recursive.

We use a recursor to define non-dependent functions. For instance, we define the *projections*:

$$\mathsf{pr}_1 :\equiv \mathsf{rec}_{A\times B}(A, \lambda a.\lambda b.a), \quad \mathsf{pr}_2 :\equiv \mathsf{rec}_{A\times B}(B, \lambda a.\lambda b.b)$$

If instead of a non-dependent function what we want is to define a dependent one, the type $C : \mathcal{U}$ must be a type family, $C : A \times B \to \mathcal{U}$. Thus, we can generalise the recursor of $A \times B$ in order to obtain the *inductor* —which we will also find for all other type formers:

$$\mathsf{ind}_{A\times B} : \prod_{C:A\times B\to\mathcal{U}} \left(\prod_{a:A} \prod_{b:B} C((a, b)) \right) \to \prod_{x:A\times B} C(x)$$

such that

$$\mathsf{ind}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b)$$

Notice that the definitional equations of the recursor and the inductor are the same. This is indeed a general pattern: the recursor is no more than a special case of the inductor, when the family C is constant.

We will see —when we introduce proof semantics in type theory—- tha the “sense” of induction is that, in order to prove a proposition over a type, it suffices to prove it for a certain subclass of “distinguished elements” of it. We may indeed be tempted to say that the type is, somehow, freely generated by them, but we will return to these questions later on. In this case, the distinguished elements are couples (a, b) where $a : A$ and $b : B$. But these are indeed the only ones we have:

Lemma 1. *Every element of $A \times B$ is a couple (a, b) where $a : A$ and $b : B$.*

Proof: We use induction over the family $D(x) :\equiv ((\mathsf{pr}_1(x), \mathsf{pr}_2(x)) =_{A\times B} x)$. (When using induction, we will usually note our instance of the family C as D.)

We must, then, show that $\prod_{a:A} \prod_{b:B} D((a, b))$, but if we fix $a : A$ and $b : B$, we have

$$\begin{aligned} D((a, b)) &\equiv (\mathsf{pr}_1((a, b)), \mathsf{pr}_2((a, b))) =_{A\times B} (a, b) \\ &\equiv (a, b) =_{A\times B} (a, b) \end{aligned}$$

by definition of the projections.

Hence, by induction, we have $\prod_{x:A}(\mathsf{pr}_1(x), \mathsf{pr}_2(x)) =_{A\times B} x$. □

Remark 3. We will reserve the word *proposition* for statements which are susceptible of being proved or refuted. A *lemma*, a *theorem* or a *corollary* are a special kind of propositions which are proven.

By using the propositions-as-types identification, while a proposition can be any type, a theorem is an inhabited one.

We might also call induction the *dependent eliminator*, in contrast to the non-dependent one, which is the recursor.

Remark 4. The recursor and the inductor of $\mathbf{1}$ are the following:

- Recursor:

$$\mathsf{rec}_{\mathbf{1}} : \prod_{C:\mathcal{U}} C \to \mathbf{1} \to C$$

where

$$\mathsf{rec}_{\mathbf{1}}(C, c, *) :\equiv c$$

- A bit more interesting, the inductor is a function

$$\mathsf{ind}_{\mathbf{1}} : \prod_{C:1\to\mathcal{U}} C(*) \to \prod_{x:\mathbf{1}} C(x)$$

such that

$$\mathsf{ind}_{\mathbf{1}}(C, c, x) :\equiv c$$

With this, we are able to show that the only element of $\mathbf{1}$ is $*$: let D defined as

$$D(x) :\equiv * = x$$

it is sufficient then to find a witness for $D(*) \equiv (* = *)$, but this is just the canonical element $\mathsf{refl}_* : * = *$; i.e., we have

$$\mathsf{ind}_*(\lambda x. * = x, \mathsf{refl}_*) : \prod_{x:\mathbf{1}} * = x$$

Remark that we have shown a propositional equality, not a judgemental one.

2.3.1 Dependent pairs

In analogy to what we did to the functions type, we introduce Σ-types, or dependent pairs. If $B : A \to \mathcal{U}$ is a type family indexed by A, we construct $\sum_{x:A} B(x)$, the type of dependent pairs where the first element is $a : A$ and the second is $b : B(a)$. Of course, if B is a constant family, then $\sum_{x:A} B$ is just $A \times B$.

Constructors for this type are generalizations of the constructors of $A \times B$. For instance, the recursor is defined as

$$\mathsf{rec}_{\sum_{x:A} B(x)} : \prod_{C:\mathcal{U}} \left(\prod_{x:A} B(x) \to C \right) \to \left(\sum_{x:A} B(x) \right) \to C$$

with

$$\mathsf{rec}_{\sum_{x:A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Induction is a function

$$\mathsf{ind}_{\sum_{x:A} B(x)\to\mathcal{U}} : \prod_{C:\sum_{x:A} B(x)\to\mathcal{U}} \left(\prod_{a:A} \prod_{b:B} C((a, b)) \right) \to \prod_{p:\sum_{x:A} B(x)} C(p)$$

with

$$\mathsf{ind}_{\sum_{x:A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Then, we can also define projections

$$\mathsf{pr}_1 : \textstyle\sum_{x:A} B(x) \to A, \quad \mathsf{pr}_2 : \prod_{p:\sum_{x:A} B(x)} B(\mathsf{pr}_1(p))$$

as

$$\mathsf{pr}_1 :\equiv \mathsf{rec}_{\sum_{x:A} B(x)}(A, \lambda a.\lambda b.a), \quad \mathsf{pr}_2 :\equiv \mathsf{ind}_{\sum_{x:A} B(x)}(\lambda p.B(\mathsf{pr}_1(p)), \lambda a.\lambda b.b)$$

We usually use the Σ-type structure in order to translate Mathematical structures. For example, a *magma* —a basic algebraic structure: a space X with a binary operation $X \times X \to X$ without any axiom— may be written as

$$\mathsf{Magma} :\equiv \sum_{A:\mathcal{U}} A \to A \to A$$

for which we use the projections to extract the space and the product.

In order to construct n-tuples, we do it by iteration: by convention, we will consider right-associativity, and so we have a judgemental equality

$$(a_1, a_2, \ldots, a_{n-1}, a_n) \equiv (a_1, (a_2, (\ldots, (a_{n-1}, a_n) \ldots)))$$

2.4 Coproduct type

Given $A : \mathcal{U}$ and $B : \mathcal{U}$, we can construct the type $A + B : \mathcal{U}$, the *coproduct* of A and B. The name, "coproduct", comes from Category Theory, since this is the dual notion of the previously introduced product type. In Set Theory, we can find an analogous construction in disjoint union $A \sqcup B$.

As before, the fact that we have not defined any kind of union of types suggests that the elements in coproducts are primitive in type theory.

Coproducts are defined by the *injections* (a dual notion of projections), functions $\mathsf{in}_1 : A \to A + B$ and $\mathsf{in}_2 : B \to A + B$ such that, in order to define a function $f : A + B \to B$ we have to give functions $g_1 : A \to B$ and $g_2 : B \to C$ such that

$$f(\mathsf{in}_1(a)) :\equiv g_1(a), \quad f(\mathsf{in}_2(b)) :\equiv g_2(b)$$

i.e., such that the following diagrams

$$\begin{array}{ccc} A + B & \xrightarrow{f} & C \\ \uparrow{\scriptstyle \mathsf{in}_1} & \nearrow{\scriptstyle g_1} & \\ A & & \end{array} \qquad \begin{array}{ccc} A + B & \xrightarrow{f} & C \\ \uparrow{\scriptstyle \mathsf{in}_2} & \nearrow{\scriptstyle g_2} & \\ B, & & \end{array}$$

commute.

This idea is encapsulated in the recursor, and we will often refer to it just as case analysis:

$$\mathsf{rec}_{A+B} : \prod_{C:\mathcal{U}} (A \to C) \to (B \to C) \to (A + B \to C)$$

where

$$\mathsf{rec}_{A+B}(C, g_1, g_2, \mathsf{in}_1(a)) :\equiv g_1(a), \quad \mathsf{rec}_{A+B}(C, g_1, g_2, \mathsf{in}_2(b)) :\equiv g_1(b)$$

Induction principle is an obvious generalization of the recursor: if we want $f : \prod_{x:A+B} C(x)$, we introduce

$$\mathsf{ind}_{A+B} : \prod_{C:A+B \to U} \left(\prod_{a:A} C(\mathsf{in}_1(a)) \right) \to \left(\prod_{b:B} C(\mathsf{in}_2(b)) \right) \to \prod_{x:A+B} C(x)$$

such that

$$\mathsf{ind}_{A+B}(C, g_1, g_2, \mathsf{in}_1(a)) :\equiv g_1(a), \quad \mathsf{ind}_{A+B}(C, g_1, g_2, \mathsf{in}_2(b)) :\equiv g_1(b)$$

Example 1 (Booleans)**.** The type of booleans is defined as $\mathbf{2} :\equiv \mathbf{1} + \mathbf{1}$. Here, the recursor corresponds to a well-known sentence in fonctional programming: *if-then-else.*

We can also define the *nullary object*, $\mathbf{0}$. The idea is that, as we intuitively think that $A \times \mathbf{1}$ is equivalent to A (and we will see what does that mean), $A + \mathbf{0}$ is equivalent to A. We can consider the recursor and the inductor of $\mathbf{0}$:

$$\mathsf{rec}_{\mathbf{0}} : \prod_{C:\mathcal{U}} \mathbf{0} \to C$$

which, once we introduce proof semantics, and we identify $\mathbf{0}$ with "false", will correspond to the explosion principle, or *ex falso quodlibet*. Or, as Umberto Eco explained,

> Quel giorno avrei potuto essere altrove. Se quel giorno non fossi stato nell'ufficio di Belbo ora sarei... a Samarcanda a vendere semi di sesamo, a fare l'editor di una collana in Braille, a dirigere la First National Bank nella Terra di Francesco Giuseppe? I condizionali controfattuali sono sempre veri perché la premessa è falsa.[2]

Finally we introduce the induction principle for $\mathbf{0}$

$$\mathsf{ind}_{\mathbf{0}} : \prod_{C:\mathbf{0}\to\mathcal{U}} \prod_{x:\mathbf{0}} C(x).$$

2.5 Natural numbers

Until now we have just introduced finite types. Historically, the first infinite concept is that of natural numbers: their elements can be constructed *à la* Peano, by specifying a natural number $0 : \mathbb{N}$ and a function $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. We will of course use the classical notation $1 :\equiv \mathsf{succ}(0), 2 :\equiv \mathsf{succ}(1), 3 :\equiv \mathsf{succ}(2), \ldots$

A fundamental property that natural numbers satisfy is that we can define functions by recursion —in the usual sense— and write proofs by induction —in the usual sense.

In order to define a non-dependent function $f : \mathbb{N} \to C$ by recursion it suffices to give an initial value $c_0 : C$ and a function $c_{\mathsf{succ}} : \mathbb{N} \to C \to C$ in such a way that the function applied to a non-zero natural number is defined by the equation

$$f(\mathsf{succ}(n)) :\equiv c_{\mathsf{succ}}(n, f(n))$$

If we merge both c_0 and c_{succ} in the same expression we obtain the recursion principle of natural numbers:

$$\mathsf{rec}_{\mathbb{N}} : \prod_{C:\mathcal{U}} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$$

where

$$\mathsf{rec}_{\mathbb{N}}(C, c_0, c_{\mathsf{succ}}, 0) :\equiv c_0, \quad \mathsf{rec}_{\mathbb{N}}(C, c_0, c_{\mathsf{succ}}, \mathsf{succ}(n)) :\equiv c_{\mathsf{succ}}(n, \mathsf{rec}_{\mathbb{N}}(C, c_0, c_{\mathsf{succ}}, n))$$

Example 2. Function $\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ performs the addition of two natural numbers, and thus here we will call $C \equiv \mathbb{N} \to \mathbb{N}$. Adding 0 to something is the identity,

$$c_0 :\equiv \lambda n.n : \mathbb{N} \to \mathbb{N}$$

and this will be our base case for recursion. Otherwise, we have

$$c_{\mathsf{succ}} : \lambda n.\lambda g.\lambda m.\mathsf{succ}(g(m))$$

where $c_{\mathsf{succ}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$. This might seem a bit convoluted, but we will see that it actually works. We have

$$\mathsf{add}(\mathsf{succ}(n), m) \equiv c_{\mathsf{succ}}(n, \mathsf{add}(n))(m) \equiv \mathsf{succ}\left(\mathsf{add}(n, m)\right)$$

since here $\mathsf{add}(n) \equiv \lambda m.\mathsf{add}(n, m)$ is our $g : \mathbb{N} \to \mathbb{N}$. Thus, we can very well define add as

$$\mathsf{add} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \lambda m.m, \lambda n.\lambda g.\lambda m.\mathsf{succ}(g(m)))$$

[2] *Il pendolo di Foucault*, Umberto Eco, chap. 19.

2.5.1 Natural induction

Induction principle for natural numbers is the fifth of the Peano's axioms. If we read "$C : \mathbb{N} \to \mathcal{U}$" as a "property concerning natural numbers", arrows as implications and the symbol "$\prod_{x:A}$" as "for all $x : A$" —which will make sense in a few pages—, the paraphrase of the following induction principle

> For any property concerning natural numbers, if 0 has this property and, for any n which satisfies it, $\mathsf{succ}(n)$ satisfies it too, then this property is true for any natural number.

can be written intuitively:

$$\mathsf{ind}_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}} C(0) \to \left(\prod_{n:\mathbb{N}} C(n) \to C(\mathsf{succ}(n))\right) \to \prod_{n:\mathbb{N}} C(n)$$

with equations

$$\mathsf{ind}_{\mathbb{N}}(C, c_0, c_{\mathsf{succ}}, 0) :\equiv c_0, \quad \mathsf{ind}_{\mathbb{N}}(C, c_0, c_{\mathsf{succ}}, \mathsf{succ}(n)) :\equiv c_{\mathsf{succ}}(n, \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, n))$$

Lemma 2. *Addition of natural numbers is commutative.*

Proof: We want to prove the statement

$$\prod_{i,j:\mathbb{N}} \mathsf{add}(i, j) =_{\mathbb{N}} \mathsf{add}(j, i).$$

As before, it suffices to exhibit a witness of this proposition, an element of this type. We consider the previous induction principle with $D : \mathbb{N} \to \mathcal{U}$ defined as

$$D :\equiv \lambda i. \prod_{j:\mathbb{N}} \mathsf{add}(i, j) =_{\mathbb{N}} \mathsf{add}(j, i).$$

We have that c_0 is very well in $C(0) \equiv \prod_{j:\mathbb{N}} \mathsf{add}(0, j) =_{\mathbb{N}} \mathsf{add}(j, 0)$, since by definition $\mathsf{add}(0, j) \equiv j$ and $\mathsf{add}(j, 0) \equiv j$ and it suffices to consider $c_0 :\equiv \lambda n.\mathsf{refl}_n$.

On the other hand, if we have $p : \mathsf{add}(i, j) =_{\mathbb{N}} \mathsf{add}(j, i)$, we want to show that $\mathsf{add}(\mathsf{succ}(i), j) =_{\mathbb{N}} \mathsf{add}(j, \mathsf{succ}(i))$. By definition, this is judgementally equal to the type $\mathsf{succ}\,(\mathsf{add}(i, j)) =_{\mathbb{N}} \mathsf{succ}\,(\mathsf{add}(j, i))$. But this will be true by lemma 12, which says that if two elements $x, y : A$ are such that $p : x =_A y$, then there exists a proof $\mathsf{ap}_f(p) : f(x) =_B f(y)$ for any function $f : A \to B$. □

Notice that we have shown that $\mathsf{add}(i, j) =_{\mathbb{N}} \mathsf{add}(j, i)$ but *not* that $\mathsf{add}(i, j) \equiv \mathsf{add}(j, i)$, since both expressions are defined in an essentially different way.

The associativity of addition is an analogous, a bit more convoluted, proof.

2.6 Proof semantics

We will now introduce a semantic interpretation due to Brouwer, Heyting and Kolmogorov, sometimes called BHK interpretation, or just Heyting semantics, which finds a model in Type Theory for Proof Theory.

We have already stated that the proof of a proposition expressed as a type is an element of this type; actually, we might have reasons to consider a particular proof and not another, since we are doing proof-relevant Mathematics: proofs are just another type of Mathematical object. A proposition becomes, after Curry-Howard interpretation, "the type of all its proofs." We will now give this interpretation for propositional logic and predicate logic.

2.6.1 Propositional logic

We can give the following table, with correspondence in English and in Type Theory:

English	Type Theory
True	$\mathbf{1}$
False	$\mathbf{0}$
A and B	$A \times B$
A or B	$A + B$
If A, then B	$A \to B$
A if, and only if, B	$(A \to B) \times (B \to A)$
Not A	$A \to \mathbf{0}$

The basic idea after this correspondence is truly intuitive:

- To construct un element (or proof) of $A \times B$ is to give a couple of elements (or proofs) in A and in B;
- to give an element of $A + B$ is to give an element of A or an element of B, and then to inject them;
- an element of $A \to B$ is a rule f which maps each element of A to an element of B, hence a proof of A is transformed into a proof of B.

Negation $A \to \mathbf{0}$ comes from the fact that $\mathbf{0}$ is identified with a false proposition since it has no elements. Thus, a proof of $A \to \mathbf{0}$ is a proof that A implies $\mathbf{0}$, and therefore A is also false (if $a : A$ and $f : A \to \mathbf{0}$, then $f(a) : \mathbf{0}$). In other words, we assume A and derive a contradiction. Hence, saying that a type A is not inhabited is saying that $\neg A$ is, and in particular $\neg \mathbf{0}$ is always inhabited, at least by $\mathsf{id}_{\mathbf{0}} : \mathbf{0} \to \mathbf{0}$.

Remark 5. Notice that we are allowed to some extend to use indirect reasoning. For instance, we are allowed to perform the following proof by contradiction: suppose A, derive $\mathbf{0}$, conclude $\neg A$.

The kind of contradiction reasoning that we are *not* allowed to do, at least *a priori*, is to suppose $\neg A$ and, by deriving a contradiction, deduce A. Logically, that would only imply $\neg(\neg A)$, which does not, in principle, imply A.

However, we could consider an *axiom*, by imposing an element of the form

$$\mathsf{lem} : \prod_{A:\mathcal{U}} \neg\neg A \to A$$

by reading, again, "$\prod_{A:\mathcal{U}}$" as "for all $A : \mathcal{U}$". In this sense, constructive Logic generalizes, rather than constraints, classical Logic.

Nevertheless, some more subtle versions of the law of excluded middle can be considered, and we will talk about them in Chapter 4. In particular, we will see that this axiom (or, rather, the one which states that for any $A : \mathcal{U}$, $A + \neg A$) is incompatible with some other axioms we will consider in next chapter.

However, we do have the following lemma:

Lemma 3. $\prod_{A:\mathcal{U}} A \to \neg\neg A$

Proof: Let $A : \mathcal{U}$; we want to show that $A \to \neg\neg A$. We can write it as type

$$A \to (A \to \mathbf{0}) \to \mathbf{0}$$

by using right-associativity. Hence, we find a witness w

$$w_A :\equiv \lambda x.\lambda y.y(x)$$

i.e., $w : \lambda(A : \mathcal{U}).\lambda(x : A).\lambda(y : A \to \mathbf{0}).(y(x) : \mathbf{0})$. □

Corollary 4 (De Morgan laws)**.** *The following rules hold:*

- $\prod_{A,B:\mathcal{U}}((A \to \mathbf{0}) \times (B \to \mathbf{0})) \to ((A + B) \to \mathbf{0})$*;*
- $\prod_{A,B:\mathcal{U}}((A \to \mathbf{0}) + (B \to \mathbf{0})) \to ((A \times B) \to \mathbf{0})$

Proof: Suppose given A and B.

For the first law, let $(x, y) : (A \to \mathbf{0}) \times (B \to \mathbf{0})$ be our hypotheses (we already know it is a couple). We want to build a function f such that $f(x, y) : (A + B) \to \mathbf{0}$. But this is just done by case analysis (or the coproduct's recursion):

$$\begin{aligned} f(x,y)(\mathsf{in}_1(a)) &:\equiv x(a) : \mathbf{0} \\ f(x,y)(\mathsf{in}_2(b)) &:\equiv y(b) : \mathbf{0} \end{aligned}$$

For the other direction, let $x : (A \to \mathbf{0}) + (B \to \mathbf{0})$. We need just to build a function by recursion. Let $a : A \to \mathbf{0}$ and $b : B \to \mathbf{0}$. We define the function by the equations

$$\begin{aligned} f(\mathsf{in}_1(a)) &:\equiv \lambda(c : A \times B).a(\mathsf{pr}_1(c)) \\ f(\mathsf{in}_2(b)) &:\equiv \lambda(c : A \times B).b(\mathsf{pr}_2(c)) \end{aligned}$$

and thus we build a function in $A \times B \to \mathbf{0}$. □

Remark 6. The other direction of De Morgan laws can not be proved with this version of Type Theory.

2.6.2 Predicate logic

To be able to talk about predicate logic we need an equivalent in Type Theory of universal and existential quantificators, $\forall$ and $\exists$, in classical Logic.

We have already stated one: the type of dependent functions as the universal quantificator. Actually, given a property $P : A \to \mathcal{U}$, we have the table

English	Type Theory
For all $x : A$, we have $P(x)$	$\prod_{x:A} P(x)$
There is a $x : A$ such that $P(x)$	$\sum_{x:A} P(x)$

- To construct, for every $x : A$, a function $f : \prod_{x:A} P(x)$ such that $f(x) : P(x)$ is to construct a device that gives a proof $f(x)$ of $P(x)$ for any $x : A$;
- similarly, to give a couple $(x, w) : \sum_{x:A} P(x)$ is to give an element of A (which we will be able to recover by applying pr_1) together with a proof that $P(x)$ holds.

Remark 7. Here types play a double role: A is a domain, a space over which we quantify, whereas $P(x)$ is a proposition which may be true or false depending on $x : A$.

Thus, another way to use Σ-types is to model *subtypes*: if $P : A \to \mathcal{U}$ is a property over A, then $\sum_{x:A} P(x)$ can be thought as the type of elements $x : A$ such that $P(x)$, the proof of this statement being the second component of the couple. We will return to this issue later.

Example 3. According to this interpretation, we define

$$n \leq m :\equiv \sum_{k:\mathbb{N}} \mathsf{add}(n, k) =_{\mathbb{N}} m$$

and

$$n < m :\equiv \sum_{k:\mathbb{N}} \mathsf{add}(n, \mathsf{succ}(k)) =_{\mathbb{N}} m$$

Finally, we can also prove that $\mathbb{N} \leftrightarrow \mathbf{1}$, which does not have, *a priori*, any logical sense. This is due to the fact that, even if we *can* view types as logical propositions, it might not be useful at all. The *logical* equivalence of $\mathbb{N}$ and $\mathbf{1}$ is not a *type* equivalence ($\mathbb{N}$ is not equivalent to $\mathbf{1}$ under any reasonable point of view). The definition of an equivalence of types will be given in next chapter.

We will also introduce the type of *mere propositions*, types for which a logical equivalence is an equivalence as types.

The "axiom of choice" is a theorem, I

By means of type theory introduced here, we can prove the following theorem

Theorem 5. *Let A and B be types, and $R : A \to B \to \mathcal{U}$ a type family. Then we have*

$$\mathsf{ac} : \left(\prod_{x:A}\sum_{y:B} R(x,y)\right) \to \left(\sum_{f:A\to B}\prod_{x:A} R(x,f(x))\right).$$

Before giving the proof, notice that we could read it as the statement

> If for every $x : A$ there exists a $y : B$ such that $R(x,y)$, then there exists a choice function $f : A \to B$ such that for every $x : A$, $R(x, f(x))$.

which seems to be a type-theoretic version of the axiom of choice. We notice, however, that in the future we will not be as comfortable with the propositions-as-types interpretation as we are now, and hence this theorem will not be regarded, in general, as the "axiom of choice." In Chapter 4 we will give a version closer to the classical one.

Proof: Intuitively, we want to show that *if* we have a dependent function g that assigns to each $x : A$ a pair (b, r) where $b : B$ and $r : R(a, b)$, we can derive a function $f : A \to B$ such that for every $a : A$ we have $R(a, f(a))$. We might then define

$$\mathsf{ac}(g) :\equiv (\lambda x.\mathsf{pr}_1(g(x)), \lambda x.\mathsf{pr}_2(g(x)))$$

And this is it. It is well-typed: the first component is, as $\lambda x.\mathsf{pr}_1(g(x)) : A \to B$, because $g(x) : \sum_{y:B} R(x,y)$. The second component also is, since we have $\lambda x.\mathsf{pr}_2(g(x)) : \prod_{x:A} R(x, \mathsf{pr}_1(g(x)))$ by definition. But, in fact, we have

$$\prod_{x:A} R(x, \mathsf{pr}_1(g(x))) \equiv \left(\lambda f. \prod_{x:A} R(x, f(x))\right)(\lambda x.\mathsf{pr}_1(g(x)))$$

so we actually have

$$(\lambda x.\mathsf{pr}_1(g(x)), \lambda x.\mathsf{pr}_2(g(x))) : \sum_{f:A\to B}\prod_{x:A} R(x, f(x))$$

just as we wanted to show. □

Notice that there is actually no choice involved: we just use the input $g : \prod_{x:A}\sum_{y:B} R(x,y)$, so it may sound a bit uneasy to classical Mathematicians. We will return to this issue in the following chapters.

2.7 Identity types

We finish this chapter by introducing identity types, which will give us the starting point for the homotopic interpretation.

For every type $A : \mathcal{U}$ we can form a type $\mathsf{Id}_A : A \to A \to \mathcal{U}$, where $\mathsf{Id}_A(a,b)$ is the proposition stating that a and b are *propositionally* equal in A: $a =_A b$. Indeed, we will usually use this latter notation rather than the former; moreover, when context is explicit enough we will write simply $a = b$. Remark again that this is *not* a judgemental equality: it is an equality we have to show, it is not defined to be like that.

The formation rule for identity types says that, given $A : \mathcal{U}$ and two elements $a, b : A$, we can consider the type $a =_A b$ in *the same universe* $\mathcal{U}$. An introduction rule says that every element is propositionally equal to itself by introducing a canonical element that we have already used:

$$\mathsf{refl} : \prod_{a:A} a =_A a$$

the proof that equality is reflexive. If $a \equiv b$, then $\mathsf{refl}_a : a =_A b$ since $a =_A b \equiv a =_A a$.

Recursor can be thought as Leibniz's indiscernability of identical principle:

$$\mathsf{rec}_{=_A} : \prod_{C:\mathcal{U}}\prod_{x,y:A}\prod_{p:x=_A y} C(x) \to C(y)$$

such that

$$\mathsf{rec}_{=_A}(C, x, x, \mathsf{refl}_x, c) :\equiv c$$

which says that for every property C over A, if x is such that $C(x)$, then every element y equal to x has the property $C(y)$.

2.7.1 Path induction

We call induction principle for identity types *path induction*, which has strong topologic connotations. This is due to the fact that, in the following interpretation, we consider proofs of the form $p : x =_A y$ as *paths* p in A between $x : A$ and $y : A$, and refl_x to be the constant loop at x.

Again, induction principle gives us a "base" over which it is sufficient to show a proposition in order to prove it for the whole space:

$$\mathsf{ind}_{=_A} : \prod_{C : \prod_{x,y:A} x =_A y \to \mathcal{U}} \left(\prod_{x:A} C(x, x, \mathsf{refl}_x) \right) \to \prod_{x,y:A} \prod_{p : x =_A y} C(x, y, p)$$

with

$$\mathsf{ind}_{=_A}(C, c, x, x, \mathsf{refl}_x) :\equiv c(x)$$

Thus, if we want to show a property over two elements $x, y : A$ and a proof $p : x =_A y$, it will suffice to prove it for every $x : A$ and the canonical proof of reflexivity, $\mathsf{refl}_x : x =_A x$. Topologically, we express the fact that in the space of paths in A without restrictions (we can choose any couple of elements in A and any path between them), $\sum_{x,y:A} x =_A y$, any path is *homotope* to the constant path in one of the endpoints, refl_x.

However, in the daily practice of Mathematics, whenever we want to prove a proposition for two elements, we fix the first of them and quantify over the second. This is, topologically speaking, analogue to work with the space of points beginning in a particular point, x. This gives us a philosophical —one such proof should be valid— and topological —this space is contractible— motivation to introduced a *based* induction principle:

$$\mathsf{ind}_{=_A}{}' : \prod_{x:A} \prod_{C : \prod_{y:A} x =_A y \to \mathcal{U}} C(x, \mathsf{refl}_x) \to \prod_{y:A} \prod_{p : x =_A y} C(y, p)$$

with

$$\mathsf{ind}_{=_A}(x, C, c, x, \mathsf{refl}_x) :\equiv c$$

These two principles are, of course, equivalent:

Lemma 6. *Path induction and based path induction are equivalent.*

Proof: We begin by proving that based path induction implies path induction. Thus, we may assume that based path induction holds. Suppose given

$$C : \prod_{x,y:A} (x =_A y) \to \mathcal{U}$$

and

$$c : \prod_{x:A} C(x, x, \mathsf{refl}_x)$$

we want to show that build an element $f : \prod_{x,y:A} \prod_{p:x=_a y} C(x, y, p)$ out of C and c such that $f(\mathsf{refl}_x) \equiv c(x)$. Let $x : A$ be a point, consider

$$C' : \prod_{y:A} (x =_A y) \to \mathcal{U}$$

such that $C' :\equiv C(x)$, and $c' : C'(x, \mathsf{refl}_x)$ such that $c' :\equiv c(x)$. As C' and c' match the premises of based path induction, we can construct a function $g : \prod_{y:A} \prod_{p:x=_A y} C'(y, p)$ such that

$$g(x, \mathsf{refl}_x) :\equiv c'.$$

But $C'(y,p)$ is actually, by definition, $C(x,y,p)$. Hence, we can define

$$f : \prod_{x,y:A} \prod_{p:x=_A y} C(x,y,p)$$

by $f(x,x,\mathsf{refl}_x) \equiv g(x,\mathsf{refl}_x)$. But this is equal to $c' \equiv c(x)$, by definition.

On the other hand, assume that path induction holds. Let $D : \prod_{x,y}(x =_A y) \to \mathcal{U}$ be a type family such that

$$D(x,y,p) :\equiv \prod_{C:\prod_{z:A}(x=_A z)\to\mathcal{U}} C(x,\mathsf{refl}_x) \to C(y,p)$$

and let $d : \prod_{x:A} D(x,x,\mathsf{refl}_x)$ be defined as

$$d :\equiv \lambda x.\lambda\left(C : \prod_{y:A}\prod_{p:x=_A y} C(y,p)\right).\lambda(c : C(x,\mathsf{refl}_x)).c$$

By using path induction over family D with d as defined above, we can define a function

$$f : \prod_{x,y:A} \prod_{p:x=_A y} D(x,y,p)$$

such that $f(x,x,\mathsf{refl}_x) :\equiv d(x)$.

Actually, f has type

$$f : \prod_{x:A}\prod_{y:A}\prod_{p:x=_A y}\prod_{C:\prod_{z:A}(x=_A z)\to\mathcal{U}} C(x,\mathsf{refl}_x) \to C(y,p)$$

but, loosely speaking, we can swap the products to define a function

$$f' : \prod_{x:A}\prod_{C:\prod_{z:A}(x=_A z)\to\mathcal{U}} C(x,\mathsf{refl}_x) \to \prod_{y:A}\prod_{p:x=_A y} C(y,p)$$

such that $f'(x,C,c,y,p) :\equiv f(x,y,p,C,c)$. Thus, we recover based path induction by means of the definitional equalities of f. $\square$

Chapter 3

Homotopy Type Theory

The basic idea of Homotopy Type Theory is that we can treat types $A : \mathcal{U}$ as topological spaces, and hence every element $x : A$ as a point in such space. We will see that fonctions $f : A \to B$ will respect fundamental aspects of this homotopic point of view: they are, under this point of view, “continue.”

In this interpretation, we will look at proofs $p : x =_A y$ as paths in A between points x and y, and thus we treat paths $l : x =_A x$ as loops based in $x : A$. Analogously to Homotopy Theory, where we have homotopies between paths —which can be thought as paths between paths—, we can ask if two paths $p, q : x =_A y$ are propositionally equal; that is, whether the type $p =_{x=_A y} q$ is inhabited or not. Thus, we have a natural notion of homotopy, or higher path, or 2-path. But it will also make sense to consider paths between paths between paths, and so on. In general, we can talk about k-paths.

As in the topological case, we can proof that there is a structure of groupoid for each level of morphisms if we treat proofs of equality as categorical morphisms —which we will prove them to be— since they will always have inverses. Thus, we will give types a structure of ∞-groupoids: a ∞-category (a category at each level of morphisms) such that every k-morphism is an isomorphism.

We will, then have the following correspondences between points of view:

Equality	Homotopies	Categories
Reflexivity	Constant path	Identity morphism
Symetry	Inverse path	Inverse morphism
Transitivity	Concatenation of paths	Morphism composition

As a disclaimer, we say that this is potentially the longest section of this text, and it contains most of the results we will use. A lot of them have very similar proofs, so we will not show all of them, but only the ones which have the most interesting or useful proofs.

3.1 Types as higher groupoids

We will not prove here that types are ∞-groupoids, since the needed conceptual structure exceeds the aim of this text. We will, however, exhibit the proof for the first level, but we will just loosely justify the others.

To begin, we have to show that elements $p : x =_A y$ are categorical morphisms for every two $x, y : A$; that is, we will regard elements of A as objects, and we want an internal law of composition such that for every $p : x =_A y$, $q : y =_A z$ we can construct a morphism $p \cdot q : x =_A z$, and that there exists for every $x : A$ a path $\mathsf{id}_x : x =_A x$ such that $\mathsf{id}_x \cdot q = q$ and $p \cdot \mathsf{id}_y = y$, by using the above notations. In order to show that a type is not only a category but a groupoid, we will also have to show that for every $p : x =_A y$ there exists a morphism $p^{-1} : y =_A x$ such that $p^{-1} \cdot p = \mathsf{id}_y$ and $p \cdot p^{-1} = \mathsf{id}_x$.

We begin by defining path concatenation:

Lemma 7. *For every type A and any $x, y, z : A$ there exists a function*

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

written as $\lambda p.\lambda q.p \cdot q$ such that $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$ for any $x : A$.

Proof: We will give three proofs of the same result. They turn out to be homotopic between them:

1. Recall path induction. Consider type family $D : \prod_{x,y:A} x =_A y \to \mathcal{U}$ defined as

   $$D(x, y, p) :\equiv \prod_{z:A} \prod_{q:y=_A z} x =_A z$$

   Hence, it suffices to consider $D(x, x, \mathsf{refl}_x) \equiv \prod_{z:A} \prod_{q:x=_A z} x =_A z$, but as an element here we can simply consider $d :\equiv \lambda z.\lambda q.q$.
   Moreover, we have
   $$\mathsf{ind}_{=_A}(D, d, x, x, \mathsf{refl}_x) :\equiv d(x)$$
   which gives an elimination rule: $\mathsf{refl}_x \cdot q \equiv q$ for any $q : x =_A z$.
2. Consider the family $E : \prod_{y,z:A} y =_A z \to \mathcal{U}$ defined as

   $$E(y, z, q) :\equiv \prod_{x:A} \prod_{p:x=_A y} x =_A z$$

   By applying path induction, we consider $E(y, y, \mathsf{refl}_y) \equiv \prod_{x:A} \prod_{p:x=_A y} x =_A y$, and also a dependent function $e :\equiv \lambda x.\lambda p.p$.
   As before, we derive $p \cdot \mathsf{refl}_y = p$.
3. It is the same proof as in 1. —an induction over p—, but doing also an induction over q. We deduce that $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$.

□

We have then shown that we have a composition law. Even more, the path that we had noted as $\mathsf{id}_x : x =_A x$ actually is $\mathsf{refl}_x : x =_A x$.

Similarly we can proof the following propositions:

Lemma 8. *For every type $A : \mathcal{U}$ and element $x, y : A$, there is a function*

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

noted as $\lambda p.p^{-1}$, such that $\mathsf{refl}_x^{-1} \equiv \mathsf{refl}_x$.

We call such a path p^{-1} the *inverse path* of p.

Theorem 9. *Suppose given $A : \mathcal{U}$, $x, y, z, w : A$ and $p : x =_A y$, $q : y =_A z$, $r : z =_A w$. We have*

i) $p = p \cdot \mathsf{refl}_y$ *and* $p = \mathsf{refl}_x \cdot p$;

ii) $p \cdot p^{-1} = \mathsf{refl}_x$ *and* $p^{-1} \cdot p = \mathsf{refl}_y$;

iii) $(p^{-1})^{-1} = p$;

iv) $(p \cdot q) \cdot r = p \cdot (q \cdot r)$.

Corollary 10. *Types have a 1-groupoid structure.*

Paths $p : x =_A y$ are 1-morphisms between objects $x, y : A$; hence, paths between p and q will be 2-morphisms between objects p and q, which will have a 2-groupoid structure with its own coherence laws; after that, we will have to worry about the 3-groupoid structure and its coherence laws, and so it goes.

We will now focus on propositions of the form $x =_A x$, which are hardly interesting in Set Theory. However, by means of this homotopic interpretation, we can think them as loops in A. We write $\Omega(A, x)$ for

the space of loops $p : x =_A x$, where we can always consider concatenation of paths and inverses. We then have an opperation

$$\Omega(A, x) \times \Omega(A, x) \to \Omega(A, x)$$

which, by the former results, is a group operation.

Formally, we define

Definition 1. A *pointed type* (A, x) is a type $A : \mathcal{U}$ together with a point $x : A$, called *base*. We write $\mathcal{U}_\bullet :\equiv \sum_{A:\mathcal{U}} A$ for the type of of every pointed type in universe $\mathcal{U}$.

Definition 2. Given a pointed type (A, x), we can consider the *space of loops* $\Omega(A, x)$, defined as the pointed type

$$\Omega(A, x) :\equiv (x =_A x, \mathsf{refl}_x)$$

whose elements are called *loops*. Moreover, for $n : \mathbb{N}$, we consider

$$\begin{aligned} \Omega^0(A, x) &:\equiv (A, x) \\ \Omega^{\mathsf{succ}(x)}(A, x) &:\equiv \Omega\left(\Omega^k(A, x)\right) \end{aligned}$$

whose elements are called k-loops.

Consider now the type $\Omega^2(A, x)$ of 2-types between loops of $\Omega(A, x)$ (some kind of analogue to $\pi_2(A, x)$, but we will not define the fundamental group exactly like that). As in Homotopy Theory, we have an Eckmann-Hilton theorem about the commutativity of higher-homotopy groups.

Theorem 11 (Eckmann-Hilton)**.** *Composition in $\Omega^2(A, x)$ is commutative: we have $\alpha \cdot \beta = \beta \cdot \alpha$ for any $\alpha, \beta : \Omega^2(A, x)$.*

Remark 8. Notice that, as before, this equality is a path satisfying its own coherence laws.

3.1.1 Functions

Since types are thought as groupoids (and, in particular, as categories) we may want functions to be *functors*. We recall the definition of a functor:

Definition 3. A *functor* $F : \mathscr{C} \to \mathscr{D}$ is a rule which maps to each object A in $\mathscr{C}$ an object FA in $\mathscr{D}$, and to each morphism $f : A \to B$ a morphism $F(f) : FA \to FB$ such that

$$F(f \circ g) = F(f) \circ F(g)$$

and

$$F(\mathsf{id}_A) = \mathsf{id}_{FA}.$$

Type-theoretically, the second condition may be expressed as the fact that functions preserve equality; topologically, we will say that they are "continuous": they respect paths. And this is indeed the case:

Lemma 12. *Let $f : A \to B$ be a function. Then, for any $x, y : A$, we have*

$$\mathsf{ap}_f : (x =_A y) \to (f(a) =_B f(b))$$

such that for every $x : A$, $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$.

Proof: Path induction over $D(x, y, p) :\equiv (f(x) =_B f(y))$. □

We will sometimes write $f(p)$ when there is no possibility of congusion. Recall that this lemma was used in the proof of commutativity of natural numbers, where the function was $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Similarly we can proof the following properties:

Lemma 13. *Let $f : A \to B$ and $f : B \to C$ be two functions, and $p : x =_A y$, $q : y =_A z$ be two paths. Then*

i) $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$;

ii) $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$;

iii) $\mathsf{ap}_g\left(\mathsf{ap}_f(p)\right) = \mathsf{ap}_{g\circ f}(p)$;

iv) $\mathsf{ap}_{\mathsf{id}_A}(p) = p$.

Which are, again, 2-paths with their own structure. We could, of course, define analogous paths for higher dimensions. For example, we have the following result:

Lemma 14. *Let $f : A \to B$, $x, y : A$, $p, q : x = y$, and $r : p = q$. Then we have a path $\mathsf{ap}^2_f(r) : \mathsf{ap}_f(p) = \mathsf{ap}_f(q)$.*

Proof: By path induction over the path r, with $D(r) :\equiv \mathsf{ap}_f(r) = \mathsf{ap}_g(r)$. □

3.1.2 Dependent functions: transport

We wish to have a "continuity" property for non-dependent functions: we also want them to have functorial behaviour. However, *a priori*, if $f : \prod_{x:A} B(x)$ and $p : x =_A y$, the "type" $f(x) = f(y)$ does not make any sense, since the types of $f(x)$ and $f(y)$ are, in general, different.

We recall Leibniz's Indiscernability of Identicals principle, which we have already stated as the recursor principle of identity types:

$$\mathsf{rec}_{=_A} : \prod_{C:A\to\mathcal{U}} \prod_{x,y:A} \prod_{p:x=_A y} C(x) \to C(y)$$

This can be rewritten in a topological way, which will be fundamental for us:

Lemma 15 (Transport)**.** *Let $P : A \to \mathcal{U}$ be a property over A and $p : x =_A y$. Then, we have a function $\mathsf{transport}^A(p, -) : P(x) \to P(y)$ such that $\mathsf{transport}^A(\mathsf{refl}_x, -) \equiv \mathsf{id}_{C(x)}$.*

When the context is clear, we allow us to write p_* instead of $\mathsf{transport}^A(p, -)$.

Remark 9. We will see later on that, if $p : x =_A y$, then the types $P(x)$ and $P(y)$ are equivalent. Under the categorical point of view, this should not surprise us: two categories are said to be equivalent if there exist functors $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ such that their composition is $F \circ G = \mathrm{Id}_{\mathscr{D}}$ and $G \circ F = \mathrm{Id}_{\mathscr{C}}$.

If we have $p : x =_A\to y$ and $p_* : P(x) \to P(y)$, we have already shown that $p^{-1} : y =_A x$ and thus $(p^{-1})_* : P(y) \to P(x)$. But we are not done yet: we still have to show that $(p^{-1})_* = (p_*)^{-1}$.

Topologically, one might think about properties $P(x)$ as *fibers* over A: each point $x : A$ might have a different fiber. This lemma says, then, that if we have a path between two points of A (the base), then we have a morphism between fibers. Nevertheless, we would like to find a path between points of different fibers, even if we have to find another type to contain it: we consider, then, the total space of fibration: $\sum_{x:A} P(x)$. There it is, at least, possible to consider the type $(x, u) =_{\sum_{x:A} P(x)} (y, v)$ of paths between $(x, u), (y, v) : \sum_{x:A} P(x)$. And of course, as we expected, it will be inhabited: we have a path lifting property here as we have it in the theory of covering spaces:

Theorem 16 (Path lifting)**.** *Suppose given $P : A \to \mathcal{U}$ a type family (or property) over A, and $p : x =_A y$. Then, for any $u : P(x)$ there exists a path*

$$\mathsf{lift}(u, p) : (x, u) =_{\sum_{x:A} P(x)} (y, p_*(u))$$

such that $\mathsf{pr}_1(\mathsf{lift}(u, p)) \equiv p$.

Proof: Path induction over the family $D(x, y, p) :\equiv (x, u) = (y, p_*(u))$ by using the previous lemma. $\square$

Hence, following the constructive philosophy, we do not only prove that ther exists a lifting, but we give a canonical one: lift.

Using topologic nomenclature, a function $f : \prod_{x:A} P(x)$ is called a *section*. We can finally state a generalization of lemma 12 over the functoriality ("continuity") of dependent functions: given a path p between x and y in A, and a section $f : \prod_{x:A} P(x)$, we want to be capable of constructing a path between $(x, f(x))$ and $(y, f(y))$, and we want it to be "over" p.

Lemma 17. *Let $x, y : A$ be two points and $f : \prod_{x:A} P(x)$ a section. We thus have a dependent function*

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

Proof: We proceed by path induction over $D(x, y, p) :\equiv p_*(f(x)) = f(y)$. $\square$ Hence, to find a path "$f(p)$" over p between $f(x)$ and $f(y)$ we have to concatenate paths $\mathsf{lift}(p, f(x))$ and $(\mathsf{refl}_y, \mathsf{apd}_f(p))$. We can, thus, think about $p_*(f(x)) =_{P(y)} f(y)$ as the type of paths between $f(x)$ and $f(y)$ lying over p, since all that is left is to compose with the canonical path $\mathsf{lift}(p, f(x))$. We will refer to this type as the type of *dependent paths*.

If $P : A \to \mathcal{U}$ is a constant family B, and $f : \prod_{x:A} B$, we will intuitively say that functions apd_f and ap_f coincide. Indeed, we have the following result:

Lemma 18. *If $P : A \to \mathcal{U}$ is a constant family B, for any $x, y : A$, $p : x =_A y$ and $b : B$ we have*

$$\mathsf{transportconst}^B(p, b) : \mathsf{transport}^P(p, b) = b.$$

Moreover,

$$\mathsf{apd}_f = \mathsf{transportconst}^B(p, f(x)) \cdot \mathsf{ap}_f.$$

Proof: Again, by path induction. $\square$

But in general types of ap_f and apd_f are different. Finally, we give the following results without proof (the proofs are, over and over again, a result of path induction). They show the good behaviour of transport over fibers.

Lemma 19. *Let $P : A \to \mathcal{U}$ be a type family, $p : x =_A y$, $q : y =_A z$ two paths and $u : P(x)$ a point in the fiber of x. In this situation, we have*

$$p_*(q_*(u)) = (p \cdot q)_*(u).$$

Thus, as we said, types $P(x)$ and $P(y)$ are categorically equivalents if $p : x =_A y$. In the next section we will give a type-theoretic definition of equivalence of types, and thus this result will have more sense.

Lemma 20. *Given a function $f : A \to B$, a family of types $P : B \to \mathcal{U}$ and a path $p : x =_A y$, we have*

$$\mathsf{transport}^{P \circ f}(p, -) = \mathsf{transport}^P(\mathsf{ap}_f(p), -).$$

Lemma 21. *Let $P, Q : A \to \mathcal{U}$ be two families of types, and $f : \prod_{x:A} P(x) \to Q(x)$ a family of functions. Let $p : x =_A y$ be a path. If $u : P(x)$, we have*

$$\mathsf{transport}^Q(p, f_x(u)) = f_y\left(\mathsf{transport}^P(p, u)\right).$$

3.2 Homotopies and equivalences

We have already suggested a concept of equivalence between types in a categorical way. Here we will introduce a definition in Type Theory.

First we introduce another analogy to Category Theory: natural transformations.

3.2.1 Homotopies

Recall η-reduction: a function is defined by its images. Thus, it might seem intuitive to think that two functions are equal if, and only if, their images are for every x in their domain. I.e., if the type

$$\prod_{x:A}(f(x) =_{B(x)} g(x))$$

is inhabited for two functions $f, g : A \to B$.

Definition 4. Let $f, g : \prod_{x:A} B(x)$. We say that f and g are *homotopes* if the type

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

is inhabited by a dependent function, called *homotopy.*

In principle, even if $(f = g) \to (f \sim g)$, the other implication is not true in general. We will introduce an axiom of *extensionality of functions*: an element $\mathsf{funext} : (f \sim g) \to (f = g)$.

Lemma 22. *Being homotopic is an equivalence relation in the space of functions.*

In the same way that functions have a functorial behaviour, we want homotopies to behave as *natural transformations* between functors. We recall the definition of a natural transformation:

Definition 5. Let $F, G : \mathscr{C} \to \mathscr{D}$ be two functors. A *natural transformation* between F and G is a rule τ such that to each coupe of objects A, B in $\mathscr{C}$ associates a coupe of morphisms $\tau_A : FA \to GA$ and $\tau_B : FB \to GB$ in $\mathscr{D}$ in such a way that the diagram

$$\begin{array}{ccc} FA & \xrightarrow{\tau_A} & GA \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} \\ FB & \xrightarrow{\tau_B} & GB \end{array}$$

commutes.

Remark 10. If we think about F as a portrait in $\mathscr{D}$ of the objects and morphisms of $\mathscr{C}$, then a natural transformation τ is a translation of the portrait generated by F to the portrait generated by G, in such a way that if the diagram

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow^{h} & \downarrow{\scriptstyle g} \\ & & C \end{array}$$

commutes in $\mathscr{C}$, then diagram

$$\begin{array}{ccccc} FA & & \xrightarrow{\tau_A} & & GA & & \\ & \searrow^{F(f)} & & & & \searrow^{Gf} & \\ \downarrow{\scriptstyle F(h)} & & FB & \xrightarrow{\tau_B} & \downarrow{\scriptstyle G(h)} & & GB \\ & \swarrow_{F(g)} & & & & \swarrow_{G(g)} & \\ FC & & \xrightarrow{\tau_C} & & GC & & \end{array}$$

commutes in $\mathscr{D}$.

If we write $\mathsf{ap}_f(p)$ as $f(p)$, then

Lemma 23. *Let $H : f \sim g$ a homotopy between $f, g : A \to B$ and $p : x =_A y$. Then the following diagram commutes:*

$$\begin{array}{ccc} f(x) & \overset{H(x)}{=\!=\!=} & g(x) \\ \Big\| {\scriptstyle f(p)} & & \Big\| {\scriptstyle g(p)} \\ f(y) & \underset{}{\overset{H(y)}{=\!=\!=}} & g(y) \end{array}$$

Proof: Path induction over $D(x, y, p) :\equiv H(x) \cdot g(p) = f(p) \cdot H(y)$. □

Thus, homotopies are natural transformations between functions. In particular, if $H : f \sim \mathsf{id}_A$, by considering the diagram

$$\begin{array}{ccc} f\left(f(x)\right) & \overset{H(f(x))}{=\!=\!=} & f(x) \\ \Big\| {\scriptstyle f(H(x))} & & \Big\| {\scriptstyle H(x)} \\ f(x) & \overset{H(x)}{=\!=\!=} & x \end{array}$$

we get that

$$H\left(f(x)\right) = f\left(H(x)\right)$$

where, again, the equivalence of paths is a path in a particular level of the groupoid.

3.2.2 Equivalence of types

Traditionally, in Type Theory we say that two types A and B are equivalents if there exist two functions $f : A \to B$ and $g : B \to A$ such that $g \circ f \sim \mathsf{id}_A$ and $g \circ f \sim \mathsf{id}_B$; that is, functions whose composition is equal to the identity at each point.

Thus, the type of "inverses" would be

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

Definition 6. Let $f : A \to B$ a function. A *quasi-inverse* of f is an element

$$(g, \alpha, \beta) : \sum_{g:B\to A} (g \circ f \sim \mathsf{id}_A) \times (g \circ f \sim \mathsf{id}_B)\,.$$

We may write $\mathsf{qinv}(f)$ to denote its type.

Example 4. Identity $\mathsf{id}_A : A \to A$ is its own quasi-inverse, with homotopies defined by $\lambda x.\mathsf{refl}_x$.

A stronger notion is that of *equivalence* of types

Definition 7. Let $f : A \to B$ be a function. We say that f is an *equivalence* if the type $\mathsf{isequiv}(f)$ is inhabited.

We will return to this topic in the next chapter. Let us just say that a type $\mathsf{isequiv}(f)$ must satisfy the following rules:

1. $\mathsf{isequiv}(f) \leftrightarrow \mathsf{qinv}(f)$; that is, both are logically equivalent, and

2. for any $e_1, e_2 : \mathsf{isequiv}(f)$, $e_1 = e_2$.

We will then say that A and B are *equivalent* (as types), and we will write $A \simeq B$.

For instance, we state that the type

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

is a candidate to be $\mathsf{isequiv}(f)$.

As expected, we have:

Lemma 24. *To be equivalent as types is an equivalence relation. In particular,*

i) For any $A : \mathcal{U}$, $\mathsf{id}_A : A \to A$ is an equivalence;

ii) for any equivalence $f : A \to B$, $f^{-1} : B \to A$ is also an equivalence;

iii) for any two equivalences $f : A \to B$ and $g : B \to C$, $g \circ f : A \to C$ is an equivalence.

3.3 The homotopic structure of type-formers

In this section we will study how the homotopical interpretation of Type Theory affects the type-formers studied in the previous chapter.

3.3.1 Product type

Suppose given $x, y : A \times B$ such that $p : x =_{A\times B} y$. By functoriality, we have $\mathsf{ap}_{\mathsf{pr}_1}(p) : \mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)$ and $\mathsf{ap}_{\mathsf{pr}_2}(p) : \mathsf{pr}_2(x) =_B \mathsf{pr}_2(y)$.

Lemma 25. *For any x and y in $A \times B$, $f : (x =_{A\times B} y) \to ((pr_1(x) =_A pr_1(y)) \times (pr_2(x) =_B pr_2(y)))$ as defined above is an equivalence.*

Proof: We consider as inverse function

$$g : ((\mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)) \times (\mathsf{pr}_2(x) =_A \mathsf{pr}_2(y))) \to (x =_{A\times B} y)$$

By the induction principle of product type, we can assume that x and y are couples (a, b) and (a', b'). Thus, we have

$$g : ((a =_A a') \times (b =_B b')) \to ((a, b) =_{A\times B} (a', b'))$$

Let $p : a =_A a'$ and $q : b =_B b'$ be two proofs of equality. We apply path induction twice, in such a way that we can suppose $a \equiv a'$ and $b \equiv b'$, hence $p \equiv \mathsf{refl}_a$ and $q \equiv \mathsf{refl}_b$. But then $(a, b) \equiv (a', b')$, and we can define $\mathsf{refl}_(a, b)$ as the image by g.

We have defined g; let us show that it is a quasi-inverse of f. We will show that $g\circ f \sim \mathsf{id}_{x=_{A\times B}y}$. Let $r : x =_{A\times B} y$ be a path. By path induction over r, we suppose $x \equiv y$ and $r \equiv \mathsf{refl}_x$. By induction on the product type over x, we suppose $x \equiv (a, b)$, and so f sends refl_x to refl_a and refl_b. But by definition of g, $g(\mathsf{refl}_a, \mathsf{refl}_b) \equiv \mathsf{refl}_{(a,b)} \equiv r$.

In the other direction, if we begin with $s : (\mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)) \times (\mathsf{pr}_2(x) =_B \mathsf{pr}_2(y)$, we can use the induction principle of the product type and so consider couples $(a, b), (a', b')$, hence $s : (a = a') \times (b = b')$. Thus, by path induction over p and q we suppose they are reflexivity paths, and so $g(\mathsf{refl}_a, \mathsf{refl}_b) = \mathsf{refl}_{(a,b)}$. Finally, f yields $(\mathsf{refl}_a, \mathsf{refl}_b)$, as we wanted. □

This is of course a very intuitive result. We note the inverse application, our g, as $\mathsf{pair}^=$. We think about it as an *introduction rule* for the type $x =_{A\times B} y$, whereas $\mathsf{ap}_{\mathsf{pr}_1}$ and $\mathsf{ap}_{\mathsf{pr}_2}$ are *elimination rules*, in such a way that the equation

$$r = \mathsf{pair}^=(\mathsf{ap}_{\mathsf{pr}_1}(r), \mathsf{ap}_{\mathsf{pr}_2}(r))$$

holds for every $r : x =_{A\times B} y$.

We will also prove that ap acts as expected over the cartesian product: if $g : A \to A'$ and $h : B \to B'$, we can define $f : A \times B \to A' \times B'$ as $f(x) :\equiv (g(\mathsf{pr}_1(x)), h(\mathsf{pr}_2(x)))$.

Theorem 26. *In the above situation, if $x, y : A \times B$ and $p : \mathsf{pr}_1(x) = \mathsf{pr}_1(y)$, $q : \mathsf{pr}_2(x) = \mathsf{pr}_2(y)$, we have*

$$\mathsf{ap}_f(\mathsf{pair}^=(p,q)) =_{f(x)=f(y)} \mathsf{pair}^=(\mathsf{ap}_g(p), \mathsf{ap}_h(q)).$$

Proof: Product induction over x and y, followed by path induction over p and q. □

Finally we study the action of transport over products. Let $A \times B : Z \to \mathcal{U}$ defined as $(A \times B)(z) :\equiv A(z) \times B(z)$. Again by path induction and theorem 25, we have

Theorem 27. *Let $A \times B : Z \to \mathcal{U}$ be a type family, and $p : z =_Z w$ a path. Let $x : A(z) \times B(z)$ be a point. Then*

$$\mathsf{transport}^{A\times B}(p, x) =_{A(w)\times B(w)} (\mathsf{transport}^A(p, \mathsf{pr}_1(x)), \mathsf{transport}^B(p, \mathsf{pr}_2(x))).$$

Remark 11. Type **1** is worth mentioning. We can prove (both by induction over **1** and by path induction) that for every $x, y : \mathbf{1}$, $(x = y) \simeq \mathbf{1}$

In particular, all elements in **1** are equal. Moreover, we only have one equality proof (up to homotopy), and hence all the ∞-groupoid structure collapses here.

Topologically, we might feel tempterd to say that **1** is "path-connected", but this is not really enough to describe it (every point in **1** is connected by a unique path (again, up to homotopy). Actually the property **1** has is even stronger: **1** is contractible in a sense yet to be defined.

Dependent pairs

Since the type of dependent pairs is a generalization of the product type, we presume that the homotopic structure of Σ-types will be a generalization of the homotopic structure of

In analogy to the dependent functions' situation, if $p : w =_{\sum_{x:A} P(x)} w'$, we can very well consider the type $\mathsf{pr}_1(p) : \mathsf{pr}_1(w) =_A \mathsf{pr}_1(w')$, but it is not possible in general to consider type $\mathsf{pr}_2(p) : \mathsf{pr}_2(w) = \mathsf{pr}_2(w')$ because second projections might very well have different codomains. By lemma 17, we know we can at last consider the type

$$\mathsf{transport}^P(\mathsf{pr}_1(p), \mathsf{pr}_2(w)) =_{P(\mathsf{pr}_1(w'))} \mathsf{pr}_2(w')$$

or $(\mathsf{pr}_1(p))_*(\mathsf{pr}_2(w)) =_{P(\mathsf{pr}_1(w'))} \mathsf{pr}_2(w')$, the type of dependent paths between $\mathsf{pr}_2(w)$ and $\mathsf{pr}_2(w')$: we will determine paths in $\sum_{x:A} P(x)$ by paths on the first projection and dependent paths over them.

Remark 12. Even if $(x, u) =_{\sum_{x:A} P(x)} (x, v)$, this might not entail that $u =_{P(x)} v$. Topological intuition helps here: a path between (x, u) and (x, v) over the fibers' space does not imply that the path between u and v lies entirely *in* the fiber $P(x)$.

We can reverse this process by generalizing theorem 25:

Theorem 28. *Let $P : A \to \mathcal{U}$ be a type family over A, and $w, w' : \sum_{x:A} P(x)$. In this situation, we have:*

$$(w = w') \simeq \sum_{p:\mathsf{pr}_1(w) =_A \mathsf{pr}_1(w')} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w).$$

Proof: We first define a function

$$f : \prod_{w,w':\sum w=w'} (w = w') \to \sum_{p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w')} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')$$

by path induction over p; that is, such that $f(w, w, \mathsf{refl}_w) :\equiv (\mathsf{refl}_{\mathsf{pr}_2(w)}, \mathsf{refl}_{\mathsf{pr}_2(w)})$.

We want, of course, to show that f is an equivalence. So we define its quasi-inverse

$$g : \prod_{w,w':\sum w=w'} \left(\sum_{p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w')} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w') \right) \to w = w'$$

by first using induction over w and w': we split them into (w_1, w_2) and (w'_1, w'_2), so it suffices to show define g over

$$\sum_{p:w_1=w'_1} (p_*(w_2) = w'_2) \to ((w_1, w_2) = (w'_1, w'_2))$$

Then we use Σ-induction and recover $p : w_1 = w'_1$ and $q : p_*(w_2) = w'_2$, and by using path induction on p we have $q : (\mathsf{refl}_{w_1})_*(w_2) = w'_2$, and it suffices to show that $(w_1, w_2) = (w_1, w'_2)$. But this is done by path induction over q.

Now we have to show that this is, effectively, its quasi-inverse. First we want to show that $f(g(r)) = r$ for all w, w' and $r : \sum_{p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w')} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')$. Using induction over r, it will suffice to show that

$$f(g(\mathsf{refl}_{w_1}, \mathsf{refl}_{w_2})) = (\mathsf{refl}_{w_1}, \mathsf{refl}_{w_2})$$

but this is true by definition.

We proceed analogously to show that $g(f(p)) = p$. □

As we did before, we call the right-to-left application $\mathsf{pair}^=$, and we can think about it as an introduction rule in contrast to elimination rules $\mathsf{ap}_{\mathsf{pr}_1}$ and $\mathsf{ap}_{\mathsf{pr}_2}$. As a corollary, we have a propositional uniqueness principle:

Corollary 29. *We have $z = (\mathsf{pr}_1(z), \mathsf{pr}_2(z))$ for any $z : \sum_{x:A} P(x)$.*

We can generalize path lifting $\mathsf{lift}(p, u) : u =_{\sum_{x:A} P(x)} p_*(u)$, which here can be expressed as

$$\mathsf{pair}^=(p, \mathsf{refl}_{p_*(u)}) : (x, u) = (y, p_*(u))$$

In general, by path induction we can show

Theorem 30. *Let $P : A \to \mathcal{U}$ and $Q : \sum_{x:A} P(x) \to \mathcal{U}$ be two type families. We can define a type family over A defined by*

$$\lambda x. \sum_{u:P(x)} Q(x, u)$$

such that for any path $p : x =_A y$ and $(u, z) : \sum_{x:A} Q(x, u)$ we have

$$\mathsf{transport}^{\Sigma Q}(p, (u, z)) = \big(\mathsf{transport}^P(p, u), \mathsf{transport}^Q(\mathsf{pair}^=(p, \mathsf{refl}_{\mathsf{transport}^P(p,u)}), z)\big)$$

where ΣQ is the name of such a type family.

Remark 13. Notice that this is indeed what we were expecting: the application $\mathsf{transport}^{\Sigma Q}(p, -)$ has type $\Big(\sum_{u:P(x)} Q(x, u)\Big) \to \Big(\sum_{u:P(y)} Q(y, u)\Big)$. The elements of both domain and codomain are couples, and if we write the function componentwise, the first one has type $P(x) \to P(y)$, suggesting that this function may be $\mathsf{transport}^P(p, -)$.

On the other hand, the path $\mathsf{pair}^=(p, \mathsf{refl}_{\mathsf{transport}^P(p,u)})$ is a proof of $(x, u) =_{\sum_{x:A} P(x)} (y, \mathsf{transport}^P(p, u))$, so

$$\mathsf{transport}^Q(\mathsf{pair}^=(p, \mathsf{refl}_{\mathsf{transport}^P(p,u)}), -) : Q(x, u) \to Q(y, \mathsf{transport}^P(p, u))$$

which applied to $z : Q(x, u)$ yields the desired equation.

3.3.2 Function extensionality

We have already stated that $(f = g) \to (f \sim g)$, but the right-to-left implication is not a theorem in this version of Type Theory. In general, if $f, g : \prod_{x:A} B(x)$, we have

$$\mathsf{happly} : (f = g) \to \prod_{x:A} \big(f(x) =_{B(x)} g(x)\big)$$

We introduce the *function extensionality axiom*: $\mathsf{funext} : (f \sim g) \to (f = g)$:

Axiom 1 (Function extensionality)**.** *happly is an equivalence. In particular, we define its quasi-inverse as*

$$\mathsf{funext} : \prod_{x:A} \big(f(x) =_{B(x)} g(X)\big) \to (f = g)$$

Remark 14. This is an axiom *now*, but it will not be in a few pages: we will introduce Voevodsky's univalence axiom, from which we will be able to prove, later on, function extensionality as a theorem.

Particularly, this is of course true for non-dependent functions. Topologically, this is just the identification of paths in function spaces with homotopies between functions; categorically, it is to say that two functions are propositionally equal if there is a family of natural equivalence between them.

As we did with $\mathsf{pair}^=$, funext can be seen as an introduction rule for the type $f = g$, and so happly as its elimination rule.

Under this perspective, the homotopies that show that both functions are quasi-inverses are thought as propositional computation rules: for all $h : \prod_{x:A} f(x) =_{B(x)} g(x)$,

$$\mathsf{happly}(\mathsf{funext}(h), x) =_{\prod_{x:A} f(x) =_{B(x)} g(x)} h(x)$$

and propositional uniqueness: for all $p : f = g$,

$$\mathsf{funext}\,(\lambda x.(\mathsf{happly}(p, x))) =_{f=g} p$$

Finally, we also have

$$\begin{array}{rcl} \mathsf{refl}_f & = & \mathsf{funext}(\lambda x.\mathsf{refl}_{f(x)}) \\ \alpha^{-1} & = & \mathsf{funext}(\lambda x.\mathsf{happly}(\alpha, x)^{-1}) \\ \alpha \cdot \beta & = & \mathsf{funext}(\lambda x.\mathsf{happly}(\alpha, x) \cdot \mathsf{happly}(\beta, x)). \end{array}$$

Transport equations

The non-dependent case is a particular case of functions. For them, given a type X, $p : x_1 =_X x_2$, $A, B : X \to \mathcal{U}$ and $f : A(x_1) \to B(x_1)$, we have

$$\mathsf{transport}^{A \to B}(p, f) = \big(\lambda x.\mathsf{transport}^B(p, f(\mathsf{transport}^A(p^{-1}, x)))\big) \tag{3.1}$$

where $A \to B :\equiv \lambda x.A(x) \to B(x)$. This equality follows by path induction. It identifies transport over the function fiber with the composition

$$A(x_2) \xrightarrow{\mathsf{transport}^A(p^{-1}, -)} A(x_1) \xrightarrow{\quad f \quad} B(x_1) \xrightarrow{\mathsf{transport}^B(p, -)} B(x_2)$$

The equation of dependent functions is analogue, but more convoluted. We will not use it, and thus we will not write it. It can be found in [13].

Moreover, we had suggested that for a type family $P : X \to \mathcal{U}$ of types, $p_*(u) =_{P(y)} v$ was seen as the type of paths between u and v over $p : x =_X y$. If P is a type of functions, we have

Lemma 31. *Let $A, B : X \to \mathcal{U}$ be two type families and $p : x =_X y$. Suppose given $f : A(x) \to B(x)$ and $g : A(y) \to B(y)$, we have then an equivalence*

$$(\mathsf{transport}^{A \to B}(p, f) = g) \simeq \prod_{a:A(x)} (\mathsf{transport}^B(p, f(a)) = g(\mathsf{transport}^A(p, x)))$$

Moreover, after (3.1), if $q : \mathsf{transport}^{A \to B}(p, f) = g$ is mapped to $\hat{q}$, then for any $a : A(x)$ the path

$$\mathsf{happly}(q, \mathsf{transport}^A(p, a)) : (\mathsf{transport}^{A \to B}(p, f))(\mathsf{transport}^A(p, a)) = g(\mathsf{transport}^A(p, a))$$

is equal to the composed path

$$\begin{array}{rcl} (\mathsf{transport}^{A \to B}(p, f))(\mathsf{transport}^A(p, a)) & = & \mathsf{transport}^B\left(f\left(\mathsf{transport}^A\left(p^{-1}, \mathsf{transport}^A(p, a)\right)\right)\right) \\ & = & \mathsf{transport}^B(f(a)) \\ & = & g(\mathsf{transport}^A(p, a)) \end{array}$$

Proof: Essentially by path induction over p and the funext properties above. □

As before, the case for P being a family of dependent function is analogous, but more convoluted. We will not need it.

3.3.3 Univalence axiom

Given two types $A, B : \mathcal{U}$ within the same universe, we can ask ourselves whether the identity type $A =_{\mathcal{U}} B$ is inhabited or not. The *univalence axiom*, first introduced by Voevodsky, is the identification of the type of paths between A and B and the type of equivalences between A and B.

Notice that the function $\mathsf{id}_{\mathcal{U}} : \mathcal{U} \to \mathcal{U}$ can be thought as a type family that associates to each $X : \mathcal{U}$ the same X; we note this family $X \mapsto X$.

Remark 15. If we think as $X \mapsto X$ as a fibration, the total space of fibration is $\sum_{A:\mathcal{U}}$; i.e., the space of pointed types.

Given a path $p : A =_{\mathcal{U}} B$ we have a fucntion $\mathsf{transport}^{X \mapsto X}(p, -) : A \to B$; this function is an equivalence. Indeed: by path induction, it suffices to consider $A \equiv B$ and $p \equiv \mathsf{refl}_A$. But then, $\mathsf{transport}^{X \mapsto X}(\mathsf{refl}_A, -)$ is the identity function $\mathsf{id}_A : A \to A$, which is an equivalence.

Thus, we have proved

Lemma 32. *Let $A, B : \mathcal{U}$ be two types. Then there exists a function*

$$\mathsf{idtoeq} : (A =_{\mathcal{U}} B) \to (A \simeq B)$$

defined as the transport function $\lambda p.\mathsf{transport}^{X \mapsto X}(p, -)$.

But as it was the case for the function extensionality axiom, our version (and *any* version thus far, [13]) of Type Theory is not sufficient to show that idtoeq is an equivalence. Hence, we introduce Voevodsky's univalence axiom.

Axiom 2 (Univalence)**.** *idtoeq is an equivalence. In particular, we define its quasi-inverse as*

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

.

Remark 16. Actually, the univalence axiom is a statement over a particular universe $\mathcal{U}$. From now on, we will suppose that all our universes are univalent.

Remark 17. As we have already said, the univalence axiom and the function extensionality axiom are not independent: the former implies the latter.

As we had suggested when introducing fibers and transport, for a type family over A, P, and a path $p : x =_A y$, we have now that $P(x) = P(y)$ by univalence.

In the same way we did with other types, we can think about ua as an introduction rule for $A =_{\mathcal{U}} B$, whereas idtoeq may be an elimination one. We have also a propositional computation rule

$$\mathsf{idtoeq}(\mathsf{ua}(f), x) = f(x)$$

and a propositional uniqueness rule: for any $p : A =_{\mathcal{U}} B$,

$$p = \mathsf{ua}(\mathsf{idtoeq}(p)).$$

Lemma 33. *The following equations hold:*

$$\begin{aligned} \mathsf{refl}_A &= \mathsf{ua}(\mathsf{id}_A) \\ \mathsf{ua}(f)^{-1} &= \mathsf{ua}(f^{-1}) \\ \mathsf{ua}(f) \cdot \mathsf{ua}(g) &= \mathsf{ua}(g \circ f). \end{aligned}$$

Finally, as a special case of lemma 20, we deduce that given $B : A \to \mathcal{U}$, $x, y : A$, $p : x =_A y$ and $u : B(x)$ we have

$$\begin{array}{rcl} \mathsf{transport}^B(p, u) & = & \mathsf{transport}^{X \mapsto X}(\mathsf{ap}_B(p), u) \\ & = & \mathsf{idtoeq}(\mathsf{ap}_B(p))(u) \end{array}$$

where $\mathsf{ap}_B(p) : B(x) =_{\mathcal{U}} B(y)$.

3.3.4 Identity type

We begin this section by giving an important property of ap_f: we presume that, if f is an equivalence, then ap_f is also one.

Theorem 34. *If $f : A \to B$ is an equivalence, then for any $a, b : A$*

$$\mathsf{ap}_f : (a =_A b) \to (f(a) =_B f(b))$$

is also an equivalence.

Proof: We already know that ap_f exists; we want to know whether it has or not a quasi-inverse. Let f^{-1} be a quasi-inverse of f; we have then homotopies

$$\alpha : \prod_{b:B} \left(f(f^{-1}(b)) =_B b\right), \quad \beta : \prod_{a:A} \left(f^{-1}(f(a)) =_A a\right),$$

We will show that a quasi-inverse of ap_f is, as we expect it to be, $\mathsf{ap}_{f^{-1}} : (f(a) =_A f(b)) \to \left(f^{-1}(f(a)) =_A f^{-1}(f(b))\right)$.

Consider the following commutative diagram:

$$\begin{array}{ccc} f^{-1}(f(a)) & \overset{\beta_a}{=\!=\!=} & a \\ \Big\| \scriptstyle{\mathsf{ap}_{f^{-1}}(\mathsf{ap}_f(p))} & & \Big\| \scriptstyle{p} \\ f^{-1}(f(b)) & \overset{\beta_b}{=\!=\!=} & b \end{array}$$

hence, we have

$$\beta_a^{-1} \cdot \mathsf{ap}_{f^{-1}}(\mathsf{ap}_f(p)) \cdot \beta_b =_{a=b} p.$$

Reciprocally, we will prove that for any $q : f(a) =_B f(b)$, we have $\mathsf{ap}_f\left(\beta_a^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \beta_b\right) = q$. By applying f to the previous diagram, we get

$$\begin{array}{ccc} f\left(f^{-1}(f(a))\right) & \overset{\alpha_{f(a)}}{=\!=\!=} & f(a) \\ \Big\| \scriptstyle{\varphi} & & \Big\| \scriptstyle{\mathsf{ap}_f\left(\beta_a^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \beta_b\right)} \\ f\left(f^{-1}(f(b))\right) & \overset{\alpha_{f(b)}}{=\!=\!=} & f(b) \end{array}$$

where $\varphi :\equiv \mathsf{ap}_f\left(\mathsf{ap}_{f^{-1}}\left(\mathsf{ap}_f(\beta_a^{-1} \cdot \mathsf{ap}_{f^{-1}}(1) \cdot \beta_b)\right)\right)$. Finally,

$$\begin{array}{rcl} \mathsf{ap}_f\left(\beta_a^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \beta_b\right) & = & \alpha_{f(a)}^{-1} \cdot \mathsf{ap}_f\left(\mathsf{ap}_{f^{-1}}\left(\mathsf{ap}_f(\beta_a^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \beta_b)\right)\right) \cdot \alpha_{f(b)} \\ & = & \alpha_{f(a)}^{-1} \cdot \mathsf{ap}_f\left(\beta_a \cdot \beta_a^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \beta_b \cdot \beta_b^{-1}\right) \cdot \alpha_{f(b)} \\ & = & \alpha_{f(a)}^{-1} \cdot \alpha_{f(a)} \cdot q \cdot \alpha_{f(b)}^{-1} \cdot \alpha_{f(b)} \\ & = & q \end{array}$$

by using 17 twice. □

Transporting paths over paths

Let $C : A \to \mathcal{U}$ a family of types such that $C(x)$ is the identity type for each $x : A$. The simplest case is when $C(x)$ is a path type in the same A; we have the following lemma

Lemma 35. *For each $A : \mathcal{U}$ and $a : A$, with $p : x_1 =_A x_2$, we have*

$$\begin{array}{rcll} \mathsf{transport}^{x\mapsto(a=x)}(p,q) & = & q \cdot p & \text{for } q : a = x_1, \\ \mathsf{transport}^{x\mapsto(x=a)}(p,q) & = & p^{-1} \cdot q & \text{for } q : x_1 = a, \\ \mathsf{transport}^{x\mapsto(x=x)}(p,q) & = & p^{-1} \cdot q \cdot p & \text{for } q : x_1 = x_1. \end{array}$$

Proof: Path induction over p. □

Corollary 36. *Let $p : a =_A a'$, $q : a = a$ and $r : a' = a'$ be paths. Then*

$$(\mathsf{transport}^{x\mapsto x=x}(p,q) = r) \simeq (q \cdot p = p \cdot r).$$

Remark 18. Functions $\mathsf{transport}^{x\mapsto(a=x)}(p,-)$ and $\mathsf{transport}^{x\mapsto(x=a)}(p,-)$, with $p : x_1 = x_2$, can be identified with functors $\mathrm{Hom}(a,-)$ and $\mathrm{Hom}(-,a)$, respectively.

Indeed: $\mathsf{transport}^{x\mapsto(a=x)}(p,-) : (a = x) \to (a = y)$. If we have $q : a = x$, then $\mathsf{transport}^{x\mapsto(a=x)}(p,q) = q \cdot p : a = y$. If we think about $x = y$ as the object $\mathrm{Hom}(x,y)$ of morphisms x and y in the type A —and analogously with $a = x$ and $a = y$. Thus, we can re-write $\mathsf{transport}^{x\mapsto(a=x)}$ as $\mathrm{Hom}(a,-)$, we have

$$\begin{array}{rccc} \mathrm{Hom}(a,p): & \mathrm{Hom}(a,x) & \to & \mathrm{Hom}(a,y) \\ & q & \mapsto & q \cdot p \end{array}$$

Similarly, $\mathsf{transport}^{x\mapsto(x=a)}(p,-)$ can be thought as

$$\begin{array}{rccc} \mathrm{Hom}(p^{-1},a): & \mathrm{Hom}(x,a) & \to & \mathrm{Hom}(y,a) \\ & q & \mapsto & p^{-1} \cdot q \end{array}$$

where $p^{-1} : y = x$.

In general, we have

Theorem 37. *Let $f, g : A \to B$ be two functions, and $p : a = Aa'$ and $q : f(a) =_B g(a)$ be paths. We have*

$$\mathsf{transport}^{x\mapsto f(x)=_B g(x)}(p,q) =_{f(a')=_B g(a')} (\mathsf{ap}_f p)^{-1} \cdot q \cdot \mathsf{ap}_g p.$$

Or, for dependent functions,

Theorem 38. *Let $B : A \to \mathcal{U}$ be a type family, $f, g : \prod_{x:A} B(x)$ be two dependent functions, and $p : a =_A a'$ and $q : f(a) =_{B(a)} g(a)$ be paths. Then,*

$$\mathsf{transport}^{x\mapsto f(x)=_{B(x)} g(x)}(p,q) = (\mathsf{apd}_f(p))^{-1} \cdot \mathsf{ap}_{\mathsf{transport}^B(p,-)}(q) \cdot \mathsf{apd}_g(p).$$

3.3.5 Coproduct type: encode/decode

It is harder to characterize the identity type for coproducts than it has been for the previous ones. We will thus introduce the encode/decode technique, which we will use extensively at the end of this text when proving Seifert-Van Kampen's theorem and computing the fundamental group of S^1. Loosely speaking, coproducts (as natural numbers) are a *positive* type: a type whose constructors are primitive notions. In $A + B$, $a : A$ is a primitive notion and the element in $A + B$ is not, since it is $\mathsf{in}_1(a)$. In contrast, product types such as $A \times B$ are *negative* types: the eliminators are primitive. If $x : A \times B$, then $\mathsf{pr}_1(x) : A$.

Coproduct type $A+B$ is given by the injections $\mathsf{in}_1 : A \to A+B$ and $\mathsf{in}_2 : B \to A+B$. We expect the following results:

$$\begin{aligned}(\mathsf{in}_1(a_1) = \mathsf{in}_1(a_2)) &\simeq (a_1 = a_2)\\ (\mathsf{in}_2(b_1) = \mathsf{in}_2(b_2)) &\simeq (b_1 = b_2)\\ (\mathsf{in}_1(a) = \mathsf{in}_2(b) &\simeq \mathbf{0}\end{aligned}$$

In order to prove them, we define a type family $\mathsf{code} : A+B \to \mathcal{U}$, with $\mathsf{code}(\mathsf{in}_1(a)) :\simeq (a_1 = a)$ in such a way that we can prove $\prod_{x:A+B}((\mathsf{in}_1(a_2) = x) \simeq \mathsf{code}(x))$. Similarly, we want $\mathsf{code}(\in 2(b)) :\equiv \mathbf{0}$. We define code as

$$\mathsf{code} :\equiv \mathsf{rec}_{A+B}(\mathcal{U}, \lambda a.(a_1 = a), \lambda b.\mathbf{0})$$

Theorem 39. *For any $x : A+B$, $(\mathsf{in}_1(a_1) = x) \simeq \mathsf{code}(x)$.*

Proof: We define the dependent function encode :

$$\mathsf{encode} : \prod_{x:A+B} \prod_{p:\mathsf{in}_1(a_1)=x} \mathsf{code}(x)$$

in such a way that $\mathsf{encode}(x,p) :\equiv \mathsf{transport}^{\mathsf{code}}(p, \mathsf{refl}_{a_1})$ —it is well defined, because $\mathsf{transport}^{\mathsf{code}}(p, -) : \mathsf{code}(\mathsf{in}_1(a_1)) \to \mathsf{code}(x)$.

We also define

$$\mathsf{decode} : \prod_{x:A+b} \prod_{c:\mathsf{code}(x)} (\mathsf{in}_1(a_1) = x)$$

as follows:

- If $x \equiv \mathsf{in}_1(a)$ for any $a : A$, then $\mathsf{code}(x) \equiv (a_1 = a)$, hence $c : a_1 = a$. Then, $\mathsf{ap}_{\mathsf{in}_1}(c) : \mathsf{in}_1(a_1) = \mathsf{in}_1(a)$, and so we define
$$\mathsf{decode}(x,c) :\equiv \mathsf{ap}_{\mathsf{in}_1}(c)$$
- If $x \equiv \mathsf{in}_2(b)$ for any $b : B$, we have no choice as $c : \mathbf{0}$.

Let us prove that encode and decode are quasi-inverses. We begin by supposing given $x : A+B$ and $p : \mathsf{in}_1(a_1) = x$, and we want to show that

$$\mathsf{decode}(x, \mathsf{encode}(x,p)) = p.$$

Using based path induction, it suffices that we show it for $x \equiv \mathsf{in}_1(a_1)$ and $p \equiv \mathsf{refl}_{\mathsf{in}_1(a_1)}$. Thus

$$\begin{aligned}\mathsf{decode}(\mathsf{in}_1(a_1), \mathsf{encode}(\mathsf{in}_1(a_1), \mathsf{refl}_{\mathsf{in}_1(a_1)})) &\equiv \mathsf{decode}(\mathsf{in}_1(a_1), \mathsf{transport}^{\mathsf{code}}(\mathsf{refl}_{\mathsf{in}_1(a_1)}, \mathsf{refl}_{a_1}))\\ &\equiv \mathsf{decode}(\mathsf{in}_1(a_1), \mathsf{refl}_{a_1})\\ &\equiv \mathsf{ap}_{\mathsf{in}_1}(\mathsf{refl}_{a_1})\\ &\equiv \mathsf{refl}_{\mathsf{in}_1(a_1)}\\ &\equiv p.\end{aligned}$$

On the other hand, if $x : A+B$ and $c : \mathsf{code}(x)$, we want $\mathsf{encode}(x, \mathsf{decode}(x,c)) = c$. We have then two possibilities:

- If $x \equiv \mathsf{in}_1(a)$ for any $a : A$, then $c : a_1 = a$ and $\mathsf{decode}(x,c) = \mathsf{ap}_{\mathsf{in}_1}(c)$. Hence
$$\begin{aligned}\mathsf{encode}(\mathsf{in}_1(a), \mathsf{decode}(\mathsf{in}_1(a), c)) &\equiv \mathsf{transport}^{\mathsf{code}}(\mathsf{ap}_{\mathsf{in}_1}(c), \mathsf{refl}_{a_1})\\ &\equiv \mathsf{transport}^{a \mapsto (a_1 = a)}(c, \mathsf{refl}_{a_1}) && \text{by using 20}\\ &\equiv \mathsf{refl}_{a_1} \cdot c && \text{by using 35}\\ &\equiv c.\end{aligned}$$
- If $x \equiv \mathsf{in}_2(b)$, then $c : \mathbf{0}$, so we may conclude anything we want.

□

Of course, there is an equivalent version for B. In particular, this lemma says that

$$(\mathsf{in}_1(a) =_{A+B} \mathsf{in}_1(a')) \to (a =_A a')$$

which may be read as the fact that in_1 is "injective" —actually, codomain and domain are equivalent. Wealso have that

$$(\mathsf{in}_1(a) =_{A+B} \mathsf{in}_2(b)) \to \mathbf{0}$$

or "the injections of A and B are disjoint in $A + B$."

Finally we characterize the action of transport over coproducts:

Lemma 40. *Let $X : \mathcal{U}$ be a type, $p : x_1 =_X x_2$ a path, and types families $A, B : X \to \mathcal{U}$. Then*

$$\begin{aligned} \mathit{transport}^{A+B}(p, \mathit{in}_1(a)) &= \mathit{in}_1(\mathit{transport}^{A}(p, a)) \\ \mathit{transport}^{A+B}(p, \mathit{in}_2(b)) &= \mathit{in}_2(\mathit{transport}^{B}(p, b)) \end{aligned}$$

where $(A + B)(x) :\equiv A(x) + B(x)$.

Proof: Path induction over p. □

3.3.6 Natural numbers

We will use again the $\mathsf{encode}/\mathsf{decode}$ method in order to characterize the path space of natural numbers, although in this case we will not fix the one of the endpoints: we characterize it for paths with both endpoints as variables. Hence, we have a family

$$\mathsf{code} : \mathbb{N} \to \mathbb{N} \to \mathcal{U}$$

defined by double recursion over $\mathbb{N}$:

$$\begin{aligned} \mathsf{code}(0, 0) &:\equiv \mathbf{1} \\ \mathsf{code}(\mathsf{succ}(m), 0) &:\equiv \mathbf{0} \\ \mathsf{code}(0, \mathsf{succ}(n)) &:\equiv \mathbf{0} \\ \mathsf{code}(\mathsf{succ}(m), \mathsf{succ}(n)) &:\equiv \mathsf{code}(m, n) \end{aligned}$$

and a dependent function recursively defined, $r : \prod_{n:\mathbb{N}} \mathsf{code}(n, n)$, such that

$$\begin{aligned} r(0) &:\equiv * \\ r(\mathsf{succ}(n)) &:\equiv r(n). \end{aligned}$$

Theorem 41. *For any $m, n : \mathbb{N}$, $(m = n) \simeq \mathit{code}(m, n)$.*

Proof: Let $\mathsf{encode} : \prod_{m,n:\mathbb{N}}(m = n) \to \mathsf{code}(m, n)$ be defined as

$$\mathsf{encode}(m, n, p) :\equiv \mathsf{transport}^{\mathsf{code}(m,-)}(p, r(m))$$

This makes sense: if $p : m =_{\mathbb{N}} n$, then $\mathsf{transport}^{\mathsf{code}(m,-)}(p, -) : \mathsf{code}(m, m) \to \mathsf{code}(m, n)$.

On the other hand we define $\mathsf{decode} : \prod_{m,n:\mathbb{N}} \mathsf{code}(m, n) \to (m = n)$ by using double induction over m and n as follows:

- If $m \equiv 0$ and $n \equiv 0$, then we need a function $\mathbf{1} \to (0 = 0)$, which will be $\lambda x.\mathsf{refl}_0$;
- if we have $\mathsf{succ}(m)$ and 0, the domain is $\mathbf{0}$;
- if we have 0 and $\mathsf{succ}(n)$, the domain is $\mathbf{0}$;
- if both arguments are not 0, then we define

$$\mathsf{decode}(\mathsf{succ}(m), \mathsf{succ}(n)) \equiv \mathsf{code}(m, n) \xrightarrow{\mathsf{decode}(m.n)} (m = n) \xrightarrow{\mathsf{ap}_{\mathsf{succ}}} (\mathsf{succ}(m) = \mathsf{succ}(n))$$

We prove that they are quasi-inverses for each other. We begin with $p : m = n$; by path induction over p, it suffices to consider $m \equiv n$ and refl_m. We want to prove that $\mathsf{decode}(m, m, \mathsf{encode}(m, m, \mathsf{refl}_m)) = \mathsf{refl}_m$.

By definition, $\mathsf{encode}(m, m, \mathsf{refl}_m) \equiv r(m)$. Hence we have to prove that $\mathsf{decode}(m, m, r(n)) = \mathsf{refl}_m$; we then use induction over m: for the base case $\mathsf{decode}(0, 0, *) = \mathsf{refl}_0$, and if $\mathsf{decode}(m, m, r(m)) = \mathsf{refl}_m$, by definition of decode, we have $\mathsf{decode}(\mathsf{succ}(m), \mathsf{succ}(m), r(\mathsf{succ}(m))) = \mathsf{ap}_{\mathsf{succ}}(\mathsf{refl}_m)$, which is $\mathsf{refl}_{\mathsf{succ}(m)}$.

On the other hand, if we begin with $c : \mathsf{code}(m,n)$, we proceed by double induction over $m, n : \mathbb{N}$: if both are 0, then $\mathsf{decode}(0,0,c) \equiv \mathsf{refl}_0$, and so $\mathsf{encode}(0,0,\mathsf{refl}_0) \equiv r(0) \equiv *$ as are all inhabitants of $\mathbf{1}$; if $m \equiv 0$ or $n \equiv 0$ but not both of them, $c : \mathbf{0}$ and we can conclude what we want. Finally, if we have $\mathsf{succ}(m), \mathsf{succ}(n)$,

$$\begin{aligned}
\mathsf{encode}(\mathsf{succ}(m), \mathsf{succ}(n), \mathsf{decode}(\mathsf{succ}(m), \mathsf{succ}(n), c)) &= \mathsf{encode}(\mathsf{succ}(m), \mathsf{succ}(n), \mathsf{ap}_{\mathsf{succ}}(\mathsf{decode}(m,n,c))) \\
&= \mathsf{transport}^{\mathsf{code}(\mathsf{succ}(m),-)}(\mathsf{ap}_{\mathsf{succ}}(\mathsf{decode}(m,n,c)), r(\mathsf{succ}(m))) \\
&= \mathsf{transport}^{\mathsf{code}(\mathsf{succ}(m),\lambda x.\mathsf{succ}(x))}(\mathsf{decode}(m,n,c), r(\mathsf{succ}(m))) \\
&= \mathsf{transport}^{\mathsf{code}(m,)}(\mathsf{decode}(m,n,c), r(m)) \\
&= \mathsf{encode}(m,n,\mathsf{decode}(m,n,c)) \\
&= c
\end{aligned}$$

by inductive hypotheses. □

In particular, we have

$$\mathsf{encode}(\mathsf{succ}(m), 0) : (\mathsf{succ}(m) = 0) \to \mathbf{0}$$

which means that 0 is not the successor of any natural number (third axiom of Peano's Arithmetic). We can even show that succ is injective:

$$(\mathsf{succ}(m) = \mathsf{succ}(n)) \xrightarrow{\mathsf{encode}} \mathsf{code}(\mathsf{succ}(m), \mathsf{succ}(n)) \equiv \mathsf{code}(m,n) \xrightarrow{\mathsf{decode}} (m = n)$$

3.4 Some universal properties

We conclude the chapter by giving some universal properties of the type-formers, without proof. They are encoded in the induction principles of the types we have already seen.

For products, we have:

Theorem 42. *Let $A, B, X : \mathcal{U}$ be three types. We have then an equivalence*

$$(X \to A \times B) \simeq (X \to A) \times (X \to B)$$

defined by

$$\begin{aligned}
\lambda f.\left(ap_{pr_1}(f), ap_{pr_2}(f)\right) &: (X \to A \times B) \to (X \to A) \times (X \to B) \\
\lambda w.\,(\lambda x.(pr_1(w))(x), \lambda y.(pr_2(w))(y)) &: (X \to A) \times (X \to B) \to (X \to A \times B).
\end{aligned}$$

.

Of course, we also have a dependent version:

Theorem 43. *Let $A, B : X \to \mathcal{U}$ be two type families. We have then an equivalence*

$$\left(\prod_{x:X} A(x) \times B(x)\right) \simeq \left(\prod_{x:X} A(x)\right) \times \left(\prod_{x:X} B(x)\right).$$

We can generalize currying for dependent functions, also. Indeed, we have the theorem

Theorem 44. *If $C : A \times B \to \mathcal{U}$ is a type family, then*

$$\left(\prod_{w:A\times B} C(w)\right) \simeq \left(\prod_{a:A}\prod_{b:B} C(a,b)\right)$$

where the right-to-left function is the induction principle for couples. In particular, if C is constant, we have

$$(A \times B \to C) \simeq (A \to (B \to C)).$$

This result is even more general:

Theorem 45. *Let $B : A \to \mathcal{U}$ be a type family, and $C : \sum_{a:A} B(a) \to \mathcal{U}$ another. Then*

$$\left(\prod_{w:\sum_{a:A} B(a)} C(w)\right) \simeq \left(\prod_{a:A}\prod_{b:B(a)} C(a,b)\right)$$

where the right-to-left function is the induction principle for dependent pairs.

Path induction can also be seen as an equivalence:

Theorem 46. *Let $a : A$ be a point and a type family $B : \prod_{x:A}(a = x) \to \mathcal{U}$. Then*

$$\left(\prod_{x:A}\prod_{p:a=x} B(x,p)\right) \simeq B(a, \mathsf{refl}_a)$$

where the right-to-left function is based path induction.

The "axiom of choice" is a theorem, II

We can give a more general version of the "axiom of choice" given in Chapter 2, where what we called B is now a type family.

Theorem 47. *Let $X : \mathcal{U}$ be a type, $A : X \to \mathcal{U}$ a type family and $P : \prod_{x:X} A(x) \to \mathcal{U}$ another. We have an equivalence*

$$\left(\prod_{x:X}\sum_{a:A(x)} P(x,a)\right) \simeq \left(\sum_{g:\prod_{x:X} A(x)}\prod_{x:X} P(x, g(x))\right).$$

Remark 19. Under the propositions-as-types interpretation, the left-to-right direction can be read as:

> If for any $x : X$ there exists $a : A(x)$ such that $P(x,a)$, then there exists a dependent choice function $g : \prod_{x:X} A(x)$ such that for any $x : X$, we have $P(x, g(x))$.

Theorem 47 says that, actually, this implication is an equivalence. Nevertheless, notice that this statement has not the usual sense, since here the choice function g is specified. In next chapter we will see a more classical version of this axiom which turns out to be an axiom.

Proof: The right-to-left direction is $\lambda f.\left(\mathsf{ap}_{\mathsf{pr}_1}(f), \mathsf{ap}_{\mathsf{pr}_2}(f)\right)$, as it was before. Now we define its quasi-inverse as $\lambda w.\,(\lambda x.((\mathsf{pr}_1(w))(x), (\mathsf{pr}_2(w))(x)))$.

Let $f : \prod_{x:X}\sum_{a:A(x)} P(x,a)$ be a dependent function, the double composition is

$$\lambda x.\,(\mathsf{pr}_1(f(x)), \mathsf{pr}_2(f(x)))$$

but by corollaire 29 this is, exactly, $\lambda x.f(x)$; hence, by η-reduction, f.

In the other direction, if we have $(g,h) : \sum_{g:\prod_{x:X} A(x)}\prod_{x:X} P(x, g(x))$, the double composition is

$$(\lambda x.g(x), \lambda x.h(x))$$

which is, again by η-reduction, (g,h). □

Chapter 4

Logic

4.1 Sets

In previous sections we have studied the structure of types analogously to the structure of topologic spaces or groupoids. Nevertheless, it is useful to have a subclass of types which works somehow similarly to sets in the classic sense. Hence, we will consider types which —up to homotopy!— form "discrete groupoids", an analogue to discrete topological spaces. Thus, sets are types without additional homotopic structure: as with topological spaces, discrete topology (where each subset is an open set) "forgets" topological structure.

Definition 8. A type A is a *set* if for all $x, y : A$ and all $p, q : x =_A y$, there exists $r : p = q$.

I.e., we have a proposition

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

Remark 20. This sets are not exactly like ZF sets. In particular, we do not have a global membership relation, say "$\in$"; they are more like sets in category theory, where elements are just "abstract points."

Example 5. After theorem 41, $\mathbb{N}$ is a set. We can also see that $\mathbf{0}$ and $\mathbf{1}$ are sets: the first because every proposition concerning it holds, and the second because of theorem 11.

We have the following results:

Lemma 48. *If $A \simeq B$ and A is a set, then B is a set.*

Proof: Let $f : A \to B$ be the equivalence, and $g : B \to A$ its quasi-inverse. Suppose $x, y : A$, and $p, q : x =_A y$; recall lemma 34:

If $f : A \to B$ is an equivalence, then $\mathsf{ap}_f : (a =_A a') \to (f(a) =_B f(a'))$ is also an equivalence.

Therefore, we have

$$(x =_A y) \simeq (g(f(x)) =_A g(f(y)))$$

since $g \circ f$ is also an equivalence. Hence, $\mathsf{ap}_{g\circ f}(p), \mathsf{ap}_{g\circ f}(q) : g(f(x)) =_A g(f(y))$. But since A is a set

$$\mathsf{ap}_{g\circ f}(p) = \mathsf{ap}_{g\circ f}(q)$$

Let F be the quasi-inverse of $\mathsf{ap}_{g\circ f}$. We have

$$F\left(\mathsf{ap}_{g\circ f}(p)\right) = F\left(\mathsf{ap}_{g\circ f}(q)\right)$$

and as $F \circ \mathsf{ap}_{g\circ f} \sim \mathsf{id}_{=_A}$, $F\left(\mathsf{ap}_{g\circ f}(p)\right) = p$ and $F\left(\mathsf{ap}_{g\circ f}(p)\right) = q$. We finish by transitivity of equality. □

Lemma 49. *If A and B are sets, then $A \times B$ is also a set.*

Lemma 50. *If A and B are sets, then $A + B$ is also a set.*

Lemma 51. *If A is any type and $B : A \to \mathcal{U}$ is a family of types such that $B(x)$ is a set for any $x : A$, then the type $\prod_{x:A} B(x)$ is also a set.*

Proof: Suppose that we have $f, g : \prod_{x:A} B(x)$ and $p, q : f = g$. After function extensionality, we have

$$p = \mathsf{funext}(\lambda x.\mathsf{happly}(p, x)) \quad \text{and} \quad q = \mathsf{funext}(\lambda x.\mathsf{happly}(q, x))$$

and for any $x : A$, indeed

$$\mathsf{happly}(p, x) : f(x) = g(x) \quad \text{and} \quad \mathsf{happly}(q, x) : f(x) = g(x).$$

But since $B(x)$ is a set for every $x : A$, $\mathsf{happly}(p, x) = \mathsf{happly}(q, x)$. By function extensionality, $\lambda x.\mathsf{happly}(p, x) = \lambda x.\mathsf{happly}(q, x)$, and so by applying $\mathsf{ap}_{\mathsf{funext}}$ we have $p = q$. □

Thus, if A is any type and B is a set, functions $A \to B$ form a set.

Lemma 52. *If A is a set, and $B : A \to \mathcal{U}$ is a type family such that $B(x)$ is a set for any $x : A$, then $\sum_{x:A} B(x)$ is a set.*

But not everything is a set, and hence this classification makes sense:

Example 6. The universe $\mathcal{U}$ is not a set. It suffices to find a type A and a path $p : A = A$ such that p is not equal to refl_A.

Consider $A = \mathbf{2}$ and $f : A \to A$ defined by

$$\begin{array}{rcl} f(0_{\mathbf{2}}) & :\equiv & 1_{\mathbf{2}} \\ f(1_{\mathbf{2}}) & :\equiv & 0_{\mathbf{2}} \end{array}$$

Then $f(f(x)) = x$ for any $x : \mathbf{2}$, and f is an equivalence. Thus, by the univalence axiom, there is path $\mathsf{ua}(f) : A = A$. But if $\mathsf{ua}(f)$ was refl_A, again by univalence, f would be $\mathsf{id}_{\mathbf{2}}$, and hence $0_{\mathbf{2}} = 1_{\mathbf{2}}$.

4.1.1 n-types

We have defined sets as the subclasse of types A such that, for any $x, y : A$, every two proofs $p, q : x =_A y$ are equal. We can also consider types A where, for any $x, y : A$, for any $p, q : x =_A y$, for any $r, s : p = q$, we have $r = s$; i.e.,

$$\prod_{x,y:A} \prod_{p,q:x=y} \prod_{r,s:p=q} r = s$$

We call these 1-types; by analogy, sets are called 0-types. We can keep adding products for each groupoid level, and hence build a complete type hierarchy: the n-types.

A set is a 0-type, but it is also a 1-type :

Lemma 53. *If A is a set, then it is also a 1-type.*

Proof: Let $f : \mathsf{isSet}(A)$ be a proof that A is a set. We have

$$f(x, y, p, q) : p = q$$

Consider $g :\equiv \lambda q.f(x, y, p, q)$ in such a way that $g : \prod_{q:x=_A y} p = q$.

If $r : q = q'$, with $q' : x =_A y$, we have $\mathsf{ap}_g(r) : r_*(g(q)) = g(q')$. By lemma 35, we deduce $g(q) \cdot r = g(q')$:

$$\mathsf{transport}^{q \mapsto (p=q)}(r, -) : (p = q) \to (p = q')$$

and so $g(q') = \mathsf{transport}^{q \mapsto (p=q)}(r, g(q)) = g(q) \cdot r$. Therefore, if $r, s : p = q$,

$$\begin{array}{rcl} g(p) \cdot r & = & g(q) \\ g(p) \cdot s & = & g(q) \end{array}$$

and $r = s$. □

4.2 Logic

In Chapter 2 we have introduced the classical propositions-as-types interpretation, or Heyting's semantics. In it, for example, a judgement

$$(a, p(a)) : \sum_{x:A} P(x)$$

represents a proof of the proposition "there exists an $x : A$ such that $P(x)$"; it consists of a witness $a : A$ of the evidence, and the proof $p(a)$ that a actually satisfies $P(a)$. But strictly speaking, in classical Mathematics when we want a proof of this statement we do not really need the witness —although we might construct it and even use it. For instance, when we want to proof an isomorphism between two structures, almost invariantly we actually give such an isomorphism and the proof that it is so. However, this is not always the case: the classic proof of bijectivity between $\mathbb{Z}$ and $\mathbb{Q}$ does not construct a concrete bijection between them.

It may be useful, then, to have a way to express propositions *without* necessarily giving any extra information: we want to be able to say, to mimic better classical Mathematical reasoning, that "there exists an $x : A$ such that $P(x)$" and that's all, no necessarily exhibiting the actual $a : A$ for which property P holds.

Another reason we may think about to reconsider our current metatheoretic interpretation of Type Theory is the fact that we have actually been able to prove a statement which can be read as the axiom of choice. And even worse, propositions-as-types interpretation not only does not allow us to prove the law of excluded middle, but we find it to be incompatible with the univalence axiom:

Theorem 54. *It is not the case that for all $A : \mathcal{U}$ we have $\neg(\neg A) \to A$.*

Proof: It suffices to assume some $f : \prod_{A:\mathcal{U}} \neg(\neg A) \to A$ and derive an element of $\mathbf{0}$. Let $e : \mathbf{2} \simeq \mathbf{2}$ an equivalence, defined by

$$\begin{aligned} e(0_{\mathbf{2}}) &:\equiv 1_{\mathbf{2}} \\ e(1_{\mathbf{2}}) &:\equiv 0_{\mathbf{2}} \end{aligned}$$

By univalence, there exists a path $\mathsf{ua}(e) : \mathbf{2} = \mathbf{2}$; we call this path p.

By hypotheses, we have supposed there existed $f(\mathbf{2}) : \neg(\neg\mathbf{2}) \to \mathbf{2}$, hence $\mathsf{apd}_f(p) : \mathsf{transport}^{A\mapsto\neg(\neg A)\to A}(p, f(\mathbf{2})) = f(\mathbf{2})$. Thus,

$$\mathsf{happly}(\mathsf{apd}_f(p), u) : \mathsf{transport}^{A\mapsto\neg(\neg A)\to A}(p, f(\mathbf{2}))(u) = f(\mathbf{2})(u)$$

But by using equation 3.1,

$$\mathsf{transport}^{A\mapsto\neg(\neg A)\to A}(p, f(\mathbf{2}))(u) = \mathsf{transport}^{A\mapsto A}(p, f(\mathbf{2})(\mathsf{transport}^{A\mapsto\neg(\neg A)}(p^{-1}, u)))$$

By function extensionality, all functions $\neg(\neg\mathbf{2}) \equiv (\neg\mathbf{2}) \to \mathbf{0}$ are the same, since they map $x : \neg\mathbf{2}$ to something in $\mathbf{0}$: if $u, v : \neg(\neg\mathbf{2})$, then we can derive the conclusion $u(x) = v(x)$ for any $x : \mathbf{2}$.

Therefore, $\mathsf{transport}^{A\mapsto\neg(\neg A)}(p^{-1}, u) = u$, and thus

$$\mathsf{transport}^{A\mapsto\neg(\neg A)\to A}(p, f(\mathbf{2}))(u) = \mathsf{transport}^{A\mapsto A}(p, f(\mathbf{2})(u))$$

By definition, transporting along $A \mapsto A$ (i.e., idtoeq) by $\mathsf{ua}(e)$ is like applying e, and hence

$$f(\mathbf{2})(u) = e(f(\mathbf{2})(u))$$

but this is false by case analysis on $f(\mathbf{2})(u) : \mathbf{2}$. □

Corollary 55. *In general it is not true that $A + (\neg A)$.*

Proof: We will show that, having $g : \prod_{A:\mathcal{U}}(A + (\neg A))$, we would have $\prod_{A:\mathcal{U}} \neg(\neg A) \to A$, which would be a contradiction.

Let $A : \mathcal{U}$ be a type and $u : \neg(\neg A)$. By definition, $g(A) : A + (\neg A)$, and so by case analysis we may have $g(A) \equiv \mathsf{in}_1(a)$ for some $a : A$, or $g(A) \equiv \mathsf{in}_2(w)$ for $w : \neg A$.

Thus, applying $u : \neg(\neg A)$ in, say, the first case, we would have $u(a) : A$ by definition, and we can conclude A; in the second case, $u(\omega) : \mathbf{0}$, so we may conclude whatever we wish to, in particular A. □

Thus, we have a conflict between the univalence axiom (which we want to keep since is essential to the homotopic interpretation) and naive propositions-as-types semantics. A possible solution is a compromise between the two: we introduce a kind of type, mere propositions, which will also solve problems stated above.

4.2.1 Mere propositions

The problems above arose, essentially, from the fact that a type contains much more information than a simple truth value: an element of $A+B$ is not only a witness of the fact that A or B, but it actually says whether A or B is inhabited. We wish to consider *truncated* types, those which express only if they are inhabited or not.

Definition 9. A type P is a *mere proposition* if, for all $x, y : P$, $x = y$.

I.e., we have

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

We have then a notion of *logical equivalence*:

Lemma 56. *If P and Q are mere propositions and there exist functions $P \to Q$ and $Q \to P$, then $P \simeq Q$.*

Proof: If $f : P \to Q$ and $g : Q \to P$, $g(f(x)) =_P x$ and $f(g(y)) =_Q y$ since both are mere propositions. Thus, they are quasi-inverses. □

Corollary 57. *If P is a mere proposition and $x_0 : P$, then $P \simeq \mathbf{1}$.*

Proof: $\mathbf{1}$ is a mere proposition (lemma 11). □

Thus, mere propositions have no higher homotopic structure, by lemma 11: all the levels are equivalent copies of $\mathbf{1}$. By syntactic analogy to sets, we can consider mere propositions to be (-1)-types.

There are, up to equivalence, two types of mere propositions: those equivalent to $\mathbf{1}$ (called *contractible* by the homotopy theoretic interpretation, corresponding to "true" propositions in logic) and those equivalent to $\mathbf{0}$.

By definition, a set is a type such that its identity types are mere propositions. We also have an immediate consequence of lemma 48 and the above remarks:

Lemma 58. *Every mere proposition is a set.*

And the following statement:

Lemma 59. *For any type $A : \mathcal{U}$, $\mathsf{isProp}(A)$ and $\mathsf{isSet}(A)$ are mere propositions.*

Proof: Consider $f, g : \mathsf{isProp}(A)$; we want to show that $f = g$, but it suffices to show that, for every $x, y : A$, $f(x, y) = g(x, y)$ (by function extensionality). But since $f, g : \mathsf{isProp}(A)$, A *is* a proposition, and hence a set, and hence $x = y$ a mere proposition. Since $f(x, y), g(x, y) : x = y$, they are equal.

Notice that we have just used the fact that A was a set: we can use the exact same proof for the fact that $\mathsf{isSet}(A)$ is a mere proposition, changing $f(x, y) = g(x, y)$ for $f(x, y, p, q) = g(x, y, p, q)$, since $f, g : \mathsf{isSet}(A)$. □

Decidability

We can now formulate an appropriate statement for the law of excluded middle:

$$\mathsf{LEM} :\equiv \prod_{A:\mathcal{U}} (\mathsf{isProp}(A) \to (A + \neg A))$$

which can be seen to be equivalent to double negation. This statement is compatible with our Type Theory —theorem 54 does not hold since $\mathbf{2}$ is not a mere proposition. However, it can't be proved within it: it has to be assumed as an axiom. We will not need it here, and so we are not going to add it to our theory. However, it is worth mentioning that even without assuming it, the law of excluded middle can be true for some propositions in constructive Mathematics. These propositions are called *decidable* in intuitionistic logic tradition.

Hence, assuming LEM is to say that all mere propositions are deducible, and there are some Mathematical problems (such as the *Word problem* for groups, [7] and [8]) that suggest that it might be a hypotheses a bit two strong.

4.2.2 Subsets, subtypes, subuniverses

We have already suggested that we could use Σ-types to mimic subtypes of a given type A: $\sum_{x:A} P(x)$ could be seen as the type of elements $x : A$ such that $P(x)$. However, for two different witnesses $p, q : P(x)$, we would have that $(x, p) \neq (x, q)$, even if the element in A is the same. This is of course counter-intuitive, but notice that if P is a mere proposition then it can not happen: the type $P(x)$, if inhabited, has only one inhabitant. This can be restated as the following lemma.

Lemma 60. *Let $P : A \to \mathcal{U}$ be a type family such that $P(x)$ is a mere proposition for every $x : A$. Let $u, v : \sum_{x:A} P(x)$ two elements such that $\mathsf{pr}_1(u) = \mathsf{pr}_2(v)$. Then, $u = v$.*

Now it makes sense to talk about subtypes in general: types of the form $\sum_{x:A} P(x)$ where $P(x)$ is always a mere proposition. We may even use the classical notation $\{x : A | P(x)\}$ to refer them. Of course, if A is a set, then $\{x : A | P(x)\}$ will also be a set, and thus we can call it a *subset* of A. We can even write that $a \in \{x | P(x)\}$ if $P(a)$ holds, and that $\{x : A | P(x)\} \subseteq \{x : A | Q(x)\}$ if there exists a function $f : \prod_{x:A} P(x) \to Q(x)$. However, we will not use this notation.

It also makes sense to talk about subtypes $\mathsf{Set}_{\mathcal{U}} :\equiv \{A : \mathcal{U} | \mathsf{isSet}(A)\}$ and $\mathsf{Prop}_{\mathcal{U}} :\equiv \{A : \mathcal{U} | \mathsf{isProp}(A)\}$, often avoiding writing the notation if the context clarifies the universe. We call these subtypes *subuniverses*, and since our universes are cumulative, we have the arrows

$$\begin{array}{rcl} \mathsf{Set}_{\mathcal{U}_i} & \to & \mathsf{Set}_{\mathcal{U}_{i+1}} \\ \mathsf{Prop}_{\mathcal{U}_i} & \to & \mathsf{Prop}_{\mathcal{U}_{i+1}}. \end{array}$$

However, the first arrow can not be an equivalence, since it would lead to Cantorian self-reference paradoxes such as Russell's. And although the second one could be an equivalence without damaging the consistency of the system, we can not prove it to be; nevertheless, we can assume it as an axiom, called of *propositional resizing*. This is not unusual, and it was first introduced as the *axiom of reducibility* in Russell and Whitehead's *Principia Mathematica*. We will not, however, use it at all.

4.2.3 Classical logic: propositional truncation

In past sections we have stated that one of the differences between classical set-theoretic logic and its type-theoretic counterpart is that in the first we have two layers, propositions and sets, and in the second we only have one, types. We can mimic this approach using mere propositions and general types, but it cannot be done naively.

For instance, it is easy to show that, if A and B are mere propositions, so is $A \times B$. We can even give an analogous result to lemma 51 with mere propositions, so that if A is any type and $B(x)$ a mere proposition for every $x : A$, $\prod_{x:A} P(x)$ is a mere proposition. These type formers are said to preserve mere propositions, but there are ones which do not: for example, even if $\mathbf{1}$ is a mere proposition, $\mathbf{2} = \mathbf{1} + \mathbf{1}$ is not. We have the same problem with Σ-types: even if $P(x)$ is a mere proposition for any $x : A$, $\sum_{x:A} P(x)$ need not to be one. Hence, we need a way yo truncate types into mere propositions. And of course we have it: it is called *propositional truncation*.

Propositional truncation, or (-1)-truncation, is a particular case of a family of type formers that, given any type, build an n-type out of it. We will not talk about them in general, but only in the cases of propositions and sets. We can see it as the type "generated" —we will talk about this later on— by two constructors:

- A function $\|-\| : A \to \|A\|$; i.e., that for any $a : A$ we have an $\|a\| : \|A\|$, and
- the assertion that $\|A\|$ is a proposition; i.e. that for any two $x, y : \|A\|$, we have a path $x = y$.

It also has an induction and a recursion principle. We will return on the issue of induction principles of "freely generated" types later. The recursion principle of $\|A\|$ says

$$\mathsf{rec}_{\|A\|} : \prod_{C:\mathcal{U}} \mathsf{isProp}(C) \to (A \to C) \to (\|A\| \to C)$$

such that

$$\mathsf{rec}_{\|A\|}(C, \varphi, g, \|a\|) :\equiv g(a),$$

while the induction one states

$$\mathsf{ind}_{\|A\|} : \prod_{C:\|A\|\to\mathcal{U}} \left(\prod_{x:\|A\|} \mathsf{isProp}(C(x))\right) \to \left(\prod_{a:A} C(a)\right) \to \prod_{x:\|A\|} C(x)$$

such that

$$\mathsf{ind}_{\|A\|}(C, \varphi, g, \|a\|) :\equiv g(a).$$

In particular, the recursion principle confirms that when trying to prove a mere proposition out of $\|A + B\|$, we are still allowed to do case analysis in $A + B$; it also says that, when trying to prove something on $\|\sum_{x:A} P(x)\|$, we *can* assume we have $(x, p) \sum_{x:A} P(x)$.

Thus, we can recover traditional logic notation in the form of mere propositions or truncated types: if P and Q denote mere propositions, or families of mere propositions, we are allowed to write (but again we will not usually do it):

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

and even set-theoretic operations can be written this way, although as we don't have LEM there are no complements in general:

$$\begin{aligned}
\{x : A|P(x)\} \cap \{x : A|Q(x)\} &:\equiv \{x : A|P(x) \wedge Q(x)\} \\
\{x : A|P(x)\} \cup \{x : A|Q(x)\} &:\equiv \{x : A|P(x) \vee Q(x)\} \\
A \setminus \{x : A|P(x)\} &:\equiv \{x : A|\neg P(x)\}
\end{aligned}$$

but in general it is not true that $(A \setminus B) \cup B = A$.

The axiom of choice is an axiom

At last, we are able to state a classical version of the axiom of choice which turns out to be an axiom.

Axiom 3 (Axiom of choice)**.** *Let X be a set and type families $A : X \to \mathcal{U}$ and $P : \prod_{x:X} A(x) \to \mathcal{U}$ such that $A(x)$ is a set for every $x : X$ and $P(x, a)$ is a mere proposition for all $x : X$ and $a : A(x)$. Then,*

$$\left(\prod_{x:X} \left\| \sum_{a:A(x)} P(x, a) \right\|\right) \to \left\| \sum_{g:\prod_{x:X} A(x)} \prod_{x:X} P(x, g(x)) \right\|.$$

Remark 21. Remark that propositional truncation appears twice: at the beginning, implying that although we know that there exists an $a : A(x)$ such that $P(x, a)$ for any $x : X$, we don't know which one; and at the ending, implying that we are not able to recover the "choice function" that gives us such an a.

Written in classical notation, we have

$$(\forall (x : X).\exists (a : A(x)).P(x, a)) \Rightarrow \left(\exists \left(g : \prod_{x:X} A(x)\right).\forall (x : X).P(x, g(x))\right)$$

Classically, the axiom of choice is equivalent to a lot of statements. One of them is "the cartesian product of a family of nonempty sets is nonempty." This is also true in Homotopy Type Theory:

Theorem 61. *The axiom of choice is equivalent to the statement that for any set X and any $Y : X \to \mathcal{U}$ a family of sets, we have*

$$\left(\prod_{x:X} \|Y(x)\|\right) \to \left\|\prod_{x:X} Y(x)\right\|.$$

Proof: Recall theorem 47, what we called the "axiom of choice". By it, we have

$$\left\|\sum_{g:\prod_{x:X} A(x)} \prod_{x:X} P(x, g(x))\right\| \simeq \left\|\prod_{x:X} \sum_{a:A(x)} P(x, a)\right\|$$

Hence, the axiom of choice is equivalent to the equation in the statement of this theorem when $Y(x) :\equiv \sum_{a:A(x)} P(x, a)$. The converse is also true: the equation stated in this theorem is equivalent to the axiom of choice when $A(x) :\equiv Y(x)$ and $P(x, a) :\equiv \mathbf{1}$ for all $x : X$, $a : A(x)$.

Since they both are mere propositions and they are logically equivalent, they are equivalent as types. □

The axiom of choice can not be proved within our current Type Theory, but it may be consistently added to it. If X is not a set, it turns out to be false: as usual, the booleans provide a counterexample:

Lemma 62. *The equation*

$$\left(\prod_{x:X} \|Y(x)\|\right) \to \left\|\prod_{x:X} Y(x)\right\|.$$

where $Y(x)$ is a set for every $x : X$ does not hold in general if X is a type.

Proof: We want to construct a counterexample; consider $X :\equiv \sum_{A:\mathcal{U}} \|\mathbf{2} = A\|$, and suppose we have $x_0 :\equiv (\mathbf{2}, \|\mathsf{refl}_{\mathbf{2}}\|) : X$; since $\|\mathbf{2} = A\|$ is a mere proposition, for any $(A, p), (B, q)$ we have an equivalence

$$((A, p) =_X (B, q)) \simeq (A \simeq B)$$

In particular,

$$(x_0 =_X x_0) \simeq (\mathbf{2} \simeq \mathbf{2})$$

and thus X is not a set.

But A is a set, by induction on the truncation and the fact that $\mathbf{2}$ is. Hence, $A \simeq B$ is a set by lemmas 48 and 49, and therefore $x_1 =_X x_2$ is a set for any $x_1, x_2 : X$, thus X is a 1-type.

We define $Y(x) :\equiv (x_0 = x)$. Now, by definition of X, for any $(A, p) : X$, we have $\|\mathbf{2} = A\|$, which is equivalent to saying $\|x_0 =_X (A, p)\|$. We have just showed that $\prod_{x:X} \|Y(x)\|$ holds. Then, if the equation in the statement of this lemma holds, so does $\left\|\prod_{x:X} Y(x)\right\|$.

Now, by induction over the truncation, we can suppose $\prod_{x:X} Y(x)$, i.e. $\prod_{x:X}(x_0 = x)$. But then X is a proposition, and therefore a set. We have then a contradiction. □

4.2.4 Contractibility

There is still one more level at the bottom of the n-types ladder: the (-2)-types. These types are called *contractible*, and represent true mere propositions. They are, as we expect them to be, equivalent to $\mathbf{1}$. We define then

$$\mathsf{isContr}(A) :\equiv \sum_{x_0:A} \prod_{x:A} (x_0 = x)$$

We will show that these types are equivalent to $\mathbf{1}$, which will match the topological definition of a contractible space: one that is a singleton up to homotopy.

The following result is straightforward:

Lemma 63. *For any type $A : \mathcal{U}$, the following propositions are logically equivalent:*

i) $\mathsf{isContr}(A)$*;*

ii) $\mathsf{isProp}(A) \times \sum_{a:A} A$*;*

iii) $A \simeq \mathbf{1}$.

As it was the case for mere propositions and types, $\mathsf{isContr}(A)$ is always a mere proposition:

Lemma 64. *For any type A, $\mathsf{isContr}(A)$ is a mere proposition.*

Proof: Suppose that we are given $c, c' : \mathsf{isContr}(A)$; by the induction principle of Σ-types, we will suppose they are of the form

$$\begin{aligned} c &:\equiv (a, p) \\ c' &:\equiv (a', p') \end{aligned}$$

Since A is contractible, it is a mere proposition, and hence its identity type also is a mere proposition (because A is a set). Thus, the type $\prod_{x:A}(y = x)$ is a mere proposition for all $y : A$, and we have already remarked that (lemma 60) that to show that $c = c'$ it suffices to show that $a = a'$. But this is indeed the case, since A is contractible. □

Corollary 65. *If A is contractible, so is $\mathsf{isContr}(A)$.*

Actually, we have an equivalence

Theorem 66. *For all A, we have $\mathsf{isProp}(A) \simeq (A \to \mathsf{isContr}(A))$.*

Proof: We want to show that

$$\left(\prod_{x,y:A} x =_A y\right) \simeq \left(A \to \sum_{x:A}\prod_{y:A} x =_A y\right)$$

From left to right, let $f : \prod_{x,y:A} x =_A y$ be a witness of the fact that A is a mere proposition. We can construct a function

$$\lambda(x : A).(x, f(x)) : A \to \sum_{x:A}\prod_{y:A} x =_A y.$$

From right to left, if $g : A \to \sum_{x:A}\prod_{y:A} x =_A y$, let $x, y : A$ be two points. We have that $z :\equiv \mathsf{pr}_1(g(x)) : A$, which is such that $z = x$ and $z = y$. Thus, $x = y$ by the properties of paths. □

4.3 Equivalences

In order to properly prove theorems in this sections we should introduce the concept of *half-adjoint equivalences*, which is related to adjoint functors. They are the bridge that connects the concepts of *bi-invertible* maps and maps with *contractible fibers*, which turn out to be equivalent (between them and to the half-adjoint maps), and it turns out that they satisfy the two conditions that we imposed to a type to be $\mathsf{isequiv}$ of a function f, namely

- $\mathsf{qinv}(f) \leftrightarrow \mathsf{isequiv}(f)$, and
- $\mathsf{isequiv}(f)$ is a mere proposition.

We will not introduce half-adjoint equivalences in this text, since they are a bit too technical and they will not be used later on: this text is already too long. All the details can be found in [13]. We will however introducte bi-invertible maps and contractible fibers, and as they turn out to be mere propositions and logically equivalent, they are equivalent as types. Thus, invoking univalence, we will use this concepts indistinguishably by calling them simply equivalences.

We finish the chapter by recovering the concept of bijective map between sets, which we will show is an equivalence.

4.3.1 Bi-invertible maps

Definition 10. We say that $f : A \to B$ is a *bi-invertible map* if it has both a left inverse and a right inverse; namely, if the type

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

is inhabited.

Remark 22. Notice that, even if we have used g in both sums, they are not necessarily the same inverse.

Theorem 67. *For any $f : A \to B$, the type $\mathsf{biinv}(f)$ is a mere proposition.*

Theorem 68. *For any $A, B : \mathcal{U}$, for any $f : A \to B$, we have*

$$\mathsf{qinv}(f) \leftrightarrow \mathsf{biinv}(f).$$

Proof: A left-to-right function is easily defined: for $(g, \alpha, \beta) : \mathsf{qinv}(f)$, we construct $(g, \alpha, g, \beta) : \mathsf{biinv}(f)$.

From right to left, suppose given (g, α, h, β), and suppose γ be the homotopy defined by $\gamma(x) :\equiv \beta^{-1}(g(x)) \cdot h(\alpha(x))$:

$$g \overset{\beta^{-1}}{\sim} h \circ f \circ g \overset{\alpha}{\sim} h$$

Now consider homotopy $\delta(x) :\equiv \gamma(f(x)) \cdot \beta(x)$; i.e.

$$g \circ f \overset{\gamma}{\sim} h \circ f \overset{\beta}{\sim} \mathsf{id}_A$$

we thus have that $(g, \alpha, \delta) : \mathsf{qinv}(f)$. □

4.3.2 Contractible fibers

We define the *fiber* of a function f at a point y as

$$\mathsf{fib}_f(y) :\equiv \sum_{x:A} f(x) = y$$

which is a sort of measure of surjectivity of $f : A \to B$. Thus, we say that a map has contractible fibers if, for all $y : B$, $\mathsf{fib}_f(y)$ is contractible. That is, we defin

$$\mathsf{isContr}(f) :\equiv \prod_{y:B} \mathsf{isContr}(\mathsf{fib}_f(y))$$

Notice that a type A is contractible when the map $A \to \mathbf{1}$ is.

Since $\mathsf{isContr}(\mathsf{fib}_f(y))$ is a mere proposition for each y, we have that

Theorem 69. *For any f, $\mathsf{isContr}(f)$ is a mere proposition.*

Finally, by using half-adjoint equivalences one can show

Theorem 70. *For any $A, B : \mathcal{U}$, for any $f : A \to B$, $\mathsf{biinv}(f) \leftrightarrow \mathsf{isContr}(f)$.*

Since both are mere propositions, this logical equivalence becomes a type equivalence. By univalence, we can consider $\mathsf{biinv}(f) = \mathsf{isContr}(f)$ for any f, and hence we will just call them $\mathsf{isequiv}(f)$.

4.3.3 Surjections and embeddings

In set theory, an equivalence of sets is a bijective map: that is, a map which is both injective and surjective. In type theory, we define those properties as follows:

Definition 11. Let $f : A \to B$ be a function.

i) We say f is *surjective* if for every $b : B$, $\|\mathsf{fib}_f(b)\|$ is inhabited.

ii) We say that f is an *embedding* if for every $x, y : A$, the function $\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y))$ is an equivalence.

If A and B are sets, its identity types are mere propositions. Thus, by lemma 56, we have that f is an embedding when

$$\prod_{x,y:A} (f(x) = f(y)) \to (x = y)$$

and hence we will say that f is injective. Note that being an embedding and a surjection are mere propositions.

Theorem 71. *A function $f : A \to B$ is an equivalence if, and only if, it is a surjection and an embedding.*

Proof: We have already proved in 34 that any equivalence is an embedding. Moreover, if f is an equivalence, then each fiber is contractible, and so f is surjective by our definition.

In the other direction, let us assume f is surjective and an embedding. Since f is surjective, the fiber of each point $b : B$ is inhabited. We have to show now that it is a mere proposition. Consider $x, y : A$, and suppose that $p : f(x) = b$ and $q : f(y) = b$. As f is an equivalence, there exists $r : x = y$ such that $\mathsf{ap}_f(r) = p \cdot q^{-1}$.

But, by lemma 35, that is like saying $r_*(p) = q$. And so, we have a path $(x, p) = (y, q)$ in the fiber, by theorem 28. □

As being an embedding and being surjective are mere propositions, the former logical equivalence is actually an equivalence of types.

Chapter 5

Inductive and higher inductive types

In order to understand inductive and higher inductive types, we will rely more on an informal discussion than in excessive formalism. Indeed, higher inductive types are right now object of current research, and so it would be too risky to enter there. As for inductive types, the only thing we will really need in this text is the idea itself, and so we will not stop in them.

5.1 Inductive types

We have already seen a few examples of inductive types, such as $\mathbb{N}$ and $A+B$. Loosely speaking, an inductive types W is a type "freely generated" by a finite number of functions whose codomain is W, which are called *constructors*. These functions can have any finite number of arguments; in particular, if they do not have any argument at all we have a an element $w : W$.

Example 7. Let $A, B : \mathcal{U}$ be two types. The *coproduct type* is the inductive type generated by the functions

- $\mathsf{in}_1 : A \to A + B$,
- $\mathsf{in}_2 : B \to A + B$,

defined as above.

The induction principle of an inductive type is just the affirmation that the type is "generated" by the constructors : it says that to build a dependent function from that type to any type, it is sufficient to give its values on the constructors.

Example 8. If we continue with the coproduct example, we will recall that the inductive principle is as follows:

$$\mathsf{ind}_{A+B} : \prod_{C:A+B\to\mathcal{U}} \left(\prod_{a:A} C(\mathsf{in}_1(a))\right) \to \left(\prod_{b:B} C(\mathsf{in}_2(b))\right) \to \prod_{x:A+B} C(x)$$

such that

$$\mathsf{ind}_{A+B}(C, c_1, c_2, \mathsf{in}_1(a)) :\equiv c_1(\mathsf{in}_1(a)), \quad \mathsf{ind}_{A+B}(C, c_1, c_2, \mathsf{in}_2(b)) :\equiv c_2(\mathsf{in}_2(b))$$

To define a dependent function with domain $A + B$ it is sufficient to give the values on the injections.

The recursion principle is the special case of deriving a non-dependent function using the induction principle; that is to say, when the codomain family is constant.

5.1.1 Uniqueness

We would expect that two types which satisfy the same induction principle —and hence, have the same constructors— are equivalent (and, by Voevodsky's univalence axiom, *equal*). This is indeed the case, but we will not prove it here, since it distances us from our goal, which is to prove Seifert-Van Kampen's theorem in Homotopy Type Theory. However, we will give an example in the (arguably) oldest, most intuitive inductive type: the natural numbers.

Remark 23. It is analogous to the fact, in category theory, that two objects with the same universal property are equivalent. It is, actually, possible to express the induction principle in a category theoretic point of view.

Natural numbers are indeed an inductive type, whose constructors are

- $0 : \mathbb{N}$,
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

To define a dependent function over $\mathbb{N}$ (which is to say, under Heyting semantics, to prove an statement concerning *all* natural numbers) it suffices to define it for 0 and to define it for $\mathsf{succ}(n)$, assuming it is defined for n:

$$\mathsf{ind}_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}} C(0) \to \left(\prod_{n:\mathbb{N}} C(n) \to C(\mathsf{succ}(n))\right) \to \prod_{n:\mathbb{N}} C(n)$$

such that

$$\mathsf{ind}_{\mathbb{N}}(C, c_0, c_{\mathsf{succ}}, 0) :\equiv c_0(0), \quad \mathsf{ind}_{\mathbb{N}}(C, c_1, c_{\mathsf{succ}}, \mathsf{succ}(n)) :\equiv c_{\mathsf{succ}}(n, \mathsf{ind}_{\mathbb{N}}(C, c_0, c_{\mathsf{succ}}, n))$$

Although it does not seem to characterize the function explicitly (in fact, strictly speaking it only assures its existence), it turns out to be sufficient to give an uniqueness principle:

Theorem 72. *Let $f, g : \prod_{x:\mathbb{N}} C(x)$ be two functions which satisfy the recurrences*

$$c_0 : C(0), \quad c_{succ} : \textstyle\prod_{n:\mathbb{N}} C(n) \to C(succ(n))$$

such that $f(0) = g(0)$ and

$$\prod_{n:\mathbb{N}} f(succ(n)) = c_{succ}(n, f(n))$$

$$\prod_{n:\mathbb{N}} g(succ(n)) = c_{succ}(n, g(n))$$

Then $f = g$.

Proof: We proceed by induction over $\mathbb{N}$, defining the type family

$$D(n) :\equiv f(n) = g(n)$$

Of course, $D(0)$ holds since $f(0) = g(0) = e_0$. If we consider an $n : \mathbb{N}$ such that $f(n) = g(n)$, then

$$f(\mathsf{succ}(n)) = c_{\mathsf{succ}}(n, f(n)) = c_{\mathsf{succ}}(n, g(n)) = g(\mathsf{succ}(n))$$

and hence we have pointwise equality of f and g. Invoking function extensionality, we have finished. □

Now, if we look back in time to, say, twenty thousand years ago, we would find some other understanding of natural numbers, codified as lists of points or bars found in the walls of caves, in such a way that the length of a list yields the considered number. This method might seem to modern Mathematicians rather archaic, but we will see that it actually represents the same type (to no-one's surprise).

For that purpose we introduce the inductive type of lists over a type A: the type $\mathsf{List}(A)$. It is a well-known type for any functional programmer, with some programming languages such as HASKELL or LISP giving some extremely good implementations of it. It has two constructors:

- $\mathsf{nil} : \mathsf{List}(A)$, and
- $\mathsf{cons} : A \times \mathsf{List}(A) \to \mathsf{List}(A)$.

Here, nil stands for the "empty list", while cons is the operation of concatenation of an element of A in the head of an existing list. The induction principle of lists is

$$\mathsf{ind}_{\mathsf{List}(A)} : \prod_{C:\mathsf{List}(A)\to\mathcal{U}} C(\mathsf{nil}) \to \left(\prod_{a:A}\prod_{l:\mathsf{List}(A)} C(l) \to C(\mathsf{cons}(a,l)\right) \to \prod_{l:\mathsf{List}(A)} C(l)$$

where

$$\mathsf{ind}_{\mathsf{List}(A)}(C, c_{\mathsf{nil}}, c_{\mathsf{cons}}, \mathsf{nil}) :\equiv c_{\mathsf{nil}}, \quad \mathsf{ind}_{\mathsf{List}(A)}(C, c_{\mathsf{nil}}, c_{\mathsf{cons}}, \mathsf{cons}(a,l)) :\equiv c_{\mathsf{cons}}(a, l, \mathsf{ind}_(C, c_{\mathsf{nil}}, c_{\mathsf{cons}}, l))$$

We expect natural numbers to be equal to the type $\mathsf{List}(\mathbf{1})$. We have already seen that the only inhabitant of $\mathbf{1}$ is $* : \mathbf{1}$ (using the induction principle of $\mathbf{1}$), so we can rename the function $\mathsf{cons} : \mathbf{1} \times \mathsf{List}(\mathbf{1}) \to \mathsf{List}(\mathbf{1})$ to be $\mathsf{succ}' : \mathsf{List}(1) \to \mathsf{List}(1)$, such that $\mathsf{succ}'(l) :\equiv \mathsf{cons}(*, l)$. To be consistent with this notation, we will also rename nil to be $0'$. Now the induction principle can be expressed as

$$\mathsf{ind}_{\mathsf{List}(\mathbf{1})} : \prod_{C:\mathsf{List}(\mathbf{1})\to\mathcal{U}} C(0') \to \left(\prod_{l:\mathsf{List}(\mathbf{1})} C(l) \to C(\mathsf{succ}'(l))\right) \to \prod_{l:\mathsf{List}(\mathbf{1})} C(l)$$

and of course, if we define

$$\begin{aligned} e_0 &:\equiv e_{\mathsf{nil}}, \\ e_{\mathsf{succ}}(l,x) &:\equiv e_{\mathsf{cons}}(*, l, x) \end{aligned}$$

we have an induction principle which is syntactically equivalent to the natural numbers' one. Thus, we have two inductive types defined in a syntactically equal manner. We are, of course, tempted to simply say that they are "obviously" the same type, and just forget the issue.

Formally, what we need is only to show that they are equivalent (and hence, by univalence, the same). Consider the following functions:

$$f :\equiv \mathsf{rec}_{\mathbb{N}}(\mathsf{List}(\mathbf{1}), \mathsf{nil}, \lambda n.\mathsf{cons}(*)) : \mathbb{N} \to \mathsf{List}(\mathbf{1}), \quad g :\equiv \mathsf{rec}_{\mathsf{List}(\mathbf{1})}(\mathbb{N}, 0, \lambda l.\mathsf{succ}) : \mathsf{List}(\mathbf{1}) \to \mathbb{N}$$

we will see that they are quasi-inverses. To show that $g \circ f$ is the identity over $\mathbb{N}$, we proceed by induction: let $D(n) :\equiv g(f(n)) =_{\mathbb{N}} n$. We have:

- if $n = 0$, then $g(f(0)) = g(\mathsf{nil}) = 0$;
- if we have $g(f(n)) = n$, we want to show that $g(f(\mathsf{succ}(n))) = \mathsf{succ}(n)$. Hence,

$$\begin{aligned} g(f(\mathsf{succ}(n))) &\equiv g(c_{\mathsf{succ}}(n, f(n))) \\ &\equiv g(\mathsf{cons}(*, f(n))) \\ &\equiv c_{\mathsf{cons}}(f(n), g(f(n))) \\ &\equiv \mathsf{succ}(g(f(n))) \\ &= \mathsf{succ}(n). \end{aligned}$$

The proof in the other direction is analogous. Thus, we have $\mathbb{N} \simeq \mathsf{List}(\mathbf{1})$. By univalence, $\mathbb{N} = \mathsf{List}(\mathbf{1})$.

5.2 Higher inductive types

If we think of types as ∞-groupoids, then we would expect to allow constructors in "other levels" of the higher groupoid structure. These are what we call *higher* inductive types: types whose constructors are not

only functions which generate points in the type, but also paths, and paths between paths, and so on. In the following pages we will discuss some examples of higher groupoids that we will need to be able to pursue our goal, taking in consideration that the induction principles will be *justified* rather than *proved*, as it is an open question to deduce the inductive principle of a higher inductive type. They come from [13] and [14].

Although, before that, we will discuss some points regarding higher inductive types which are worth keeping in mind. But for that we might find it useful to have some example in mind. Such an example could very well be S^1: the circle.

We define S^1 as the (higher) inductive type generated by

- a point $\mathsf{base} : S^1$, and
- a path $\mathsf{loop} : \mathsf{base} =_{S^1} \mathsf{base}$.

A path constructor such as loop generates a new inhabitant of the type $\mathsf{base} =_{S^1} \mathsf{base}$ which is not —at least *a priori*, and we will see that this is the case— equal to $\mathsf{refl}_{\mathsf{base}}$. This distinguishes essentially S^1 from $\mathbf{1}$.

With this in mind, we should take into account the following:

1. When we say that the constructors "generate" the higher inductive type structure we mean exactly that: analogously to group free generation, the operations inherent to the ∞-groupoid structure create objects which are not explicitly generated. For example, the path $\mathsf{loop} \cdot \mathsf{loop}$ is not equal to the path $\mathsf{loop} \cdot \mathsf{loop} \cdot \mathsf{loop}$, and nor are they equal to loop^{-1} and so on.

2. The generation is "free" in the sense that it does not allow us to impose axioms such as $\mathsf{loop} \cdot \mathsf{loop} = \mathsf{refl}_{\mathsf{base}}$. By all means, we could astutely impose a path between $\mathsf{loop} \cdot \mathsf{loop}$ and $\mathsf{refl}_{\mathsf{base}}$, but then we would have to worry about the coherence laws that this path between path should satisfy, hidden in the shadows of the higher groupoid structure.

3. Although the higher inductive definition of a type W contains information about the identity types $=_W$, $=_{=_W}$, etc., it is still the definition of W, and the induction principle generated by it only applies to W.

   It may be, and usually is, nontrivial to compute the path spaces of a higher inductive type. For example, we could wonder if there are any paths between base and itself which are not composites of loop and its inverse. The reader familiar with algebraic topology will of course know that the answer is no (and we will prove it in this text). But this same reader may note that, in general, if we climb levels of the ∞-groupoid structure of spheres we will quite immediately find ourselves in the presence of problems such as the computation of fundamental groups of spheres, which is still an open problem.

4. Finally, the "levels" to which the constructors belong (i.e. whether they are a path between points, or paths, or higher paths) has no direct relation with the levels in which the generated type has trivial homotopy structure. But this happens even with (ordinary) inductive types: if B is generated by a function $f : A \to B$, then every path p in A yields a path $\mathsf{ap}_f(p)$ in B.

The recursion principle of a higher inductive type is, usually, easy to describe: given any type A which has the same structure as the one induced by the constructors of a higher inductive type W, then there exists $f : W \to A$ which maps the constructors to that structure. In the example of S^1, it only says that given a type A with $a : A$ and $p : a = a$ there is a function $f : S^1 \to A$ such that $f(\mathsf{base}) = a$ and $\mathsf{ap}_f(\mathsf{loop}) = p$.

These two equalities are *computation rules*. We will impose that the computation rules regarding *points* are judgemental (i.e., $f(\mathsf{base}) \equiv a$), in part because, otherwise, the type $\mathsf{ap}_f(\mathsf{loop}) = p$ would be ill-typed —since $\mathsf{ap}_f(\mathsf{loop}) : f(\mathsf{base}) = f(\mathsf{base})$ and $p : a = a$. The computational rules of paths and higher paths will not, however, be taken as judgemental. We might, if we want to emphasize the "definitional" character of these, write $\mathsf{ap}_f(\mathsf{loop}) :\equiv p$.

As for the induction principle, we may want to have some sort of "action" on the fibers, analogous to the case of inductive types. For example, in the natural numbers case to prove $\prod_{n:\mathbb{N}} P(n)$ we need an element $b : P(0)$ in the fiber of P over 0 and, for each n, a function $f : P(n) \to P(\mathsf{succ}(n))$. Analogously, to

prove $\prod_{x:S^1} P(x)$, we should specify a point $b : P(\mathsf{base})$ and a path between $u : P(\mathsf{base})$ and $v : P(\mathsf{base})$: $\mathsf{transport}^P(\mathsf{loop}, u) = v$. We will return to this question later.

Remark 24. Note that we don't mean simply a path $b = b$: we want a path that "lies over" the path loop, and hence we need it to be a dependent path.

From now on, we allow ourselves to write "$u =^P_p v$" for "$\mathsf{transport}^P(p, u) = v$". The rest of the chapter centers in a description of some higher inductive types that we will need.

5.3 The interval

One of the simplest higher inductive types one can think about is the interval. We note it by I. It is generated by:

- Two points $0_I, 1_I : I$,
- a path $\mathsf{seg} : 0_I = 1_I$.

The recursion principle for the interval says that if we want to build a non-dependent function from it to a type, we need to specify the values in 0_I and 1_I, and then path between the images. That is,

$$\mathsf{rec}_I : \prod_{C:\mathcal{U}} \left(\prod_{c_0, c_1 : C} c_0 = c_1 \right) \to I \to C$$

such that, if $f :\equiv \mathsf{rec}_I(C, c_0, c_1, s)$, we have

$$\begin{aligned} f(0_I) &:\equiv c_0 \\ f(1_I) &:\equiv c_1 \\ \mathsf{ap}_f(\mathsf{seg}) &:= s. \end{aligned}$$

Similarly, the induction principle is

$$\mathsf{ind}_I : \prod_{C:I\to\mathcal{U}} \left(\prod_{c_0:P(0_I)} \prod_{c_1:P(1_I)} c_0 =^C_{\mathsf{loop}} c_1 \right) \to \prod_{x:I} C(x)$$

with the same definitional equations as the recursion principle.

When regarded in purely homotopic terms, the interval turns out to be of little interest:

Lemma 73. *The type I is contractible.*

Proof: We proceed by I-induction. We want to show that I is contractible; that is, we want to build a function $f : \prod_{x:I} x = 1_I$. We define it as follows

$$\begin{aligned} f(0_I) &:\equiv \mathsf{seg} : 0_I = 1_I \\ f(1_I) &:\equiv \mathsf{refl}_{1_I} : 1_I = 1_I \end{aligned}$$

The only thing left is to define $\mathsf{apd}_f(\mathsf{seg}) : \mathsf{seg} =^{\lambda x. x = 1_I}_{\mathsf{seg}} \mathsf{refl}_{1_I}$. But after 35, this is equal to $\mathsf{seg}^{-1} \cdot \mathsf{seg} = \mathsf{refl}_{1_I}$. □

Nevertheless, it can be very useful from the type-theoretic point of view, since it allows us to prove what we had assumed as an axiom: function extensionality.

Lemma 74 (Function extensionality)**.** *If $f, g : A \to B$ are two functions such that $f(x) = g(x)$ for all $x : A$, the $f = g$.*

Proof: Let $p : \prod_{x:A} f(x) = g(x)$ be the proof, which we have by hypotheses. For all $x : A$, we define $\tilde{p}_x : I \to B$ by

$$\begin{array}{rcl} \tilde{p}_x(0_I) & :\equiv & f(x) \\ \tilde{p}_x(1_I) & :\equiv & g(x) \\ \tilde{p}_x(\mathsf{seg}) & := & p(x) \end{array}$$

Now, in order to invoke the recursion principle of I with $D : A \to B$, define $q : I \to (A \to B)$ by

$$q(i) :\equiv (\lambda x.\tilde{p}_x(i))$$

This definition satisfies $q(0_I) = f$, $q(1_I) = g$, and so $q(\mathsf{seg}) : f =_{A \to B} g$. □

Remark 25. We observe that this is exactly the classical scheme of a proof of homotopy between continuous applications.

5.4 Spheres

5.4.1 S^1

We have already introduced informally the induction principle of S^1. Explicitly, it is as follows:

$$\mathsf{ind}_{S^1} : \prod_{C:S^1 \to \mathcal{U}} \left(\prod_{b:C(\mathsf{base})} b =^C_{\mathsf{loop}} b \right) \to \prod_{x:S^1} C(x)$$

such that, if $f :\equiv \mathsf{ind}_{S^1}(C, b, l)$, then

$$\begin{array}{rcl} f(\mathsf{base}) & :\equiv & b \\ \mathsf{apd}_f(\mathsf{loop}) & := & l. \end{array}$$

We expect to show a recursion principle out of the induction one, but it turns out to be nontrivial in this case:

Lemma 75. *Let A be a type, $a : A$ and $p : a =_A a$ a path. Then there is a function $f : S^1 \to A$ such that*

$$f(\mathsf{base}) :\equiv a, \quad \mathsf{ap}_f(\mathsf{loop}) := p.$$

Proof: We want to apply the induction principle of S^1 to the constant family $\lambda x.A : S^1 \to \mathcal{U}$. In order to do that, we need a point in $(\lambda x.A)(\mathsf{base}) \equiv A$, a, and a dependent loop $a =^{x \mapsto A}_{\mathsf{loop}} a$. We might be tempted to say that this is $p : a =_A a$, but they do not have the same type.

From lemma 18,

$$\mathsf{transportconst}^A_{\mathsf{loop}}(a) : \mathsf{transport}^{x \mapsto A}(\mathsf{loop}, a) = a$$

and we consider $\mathsf{transportconst}^A_{\mathsf{loop}}(a) \cdot p : \mathsf{transport}^{x \mapsto A}(\mathsf{loop}, a) = a$. Finally, we can apply induction with

$$f(\mathsf{base}) :\equiv a, \quad \mathsf{apd}_f(\mathsf{loop}) := \mathsf{transportconst}^A_{\mathsf{loop}}(a) \cdot p$$

and hence, by lemma 18,

$$\mathsf{apd}_f(\mathsf{loop}) = \mathsf{transportconst}^A_{\mathsf{loop}}(f(\mathsf{base})) \cdot \mathsf{ap}_f(\mathsf{loop}),$$

and thus $\mathsf{ap}_f(\mathsf{loop}) = p$. □

As we promised, we can state the following result.

Corollary 76. *The circle is not trivial: $\mathsf{loop} \neq \mathsf{refl}_{\mathsf{base}}$.*

Proof: Suppose $\mathsf{loop} = \mathsf{refl}_{\mathsf{base}}$. From the recursion principle above, for every type A with $p : a =_A a$, we have a function $f : S^1 \to A$ such that $f(\mathsf{base}) :\equiv a$ and $\mathsf{ap}_f(\mathsf{loop}) := p$.

But then, we would have $p = \mathsf{ap}_f(\mathsf{loop}) = \mathsf{ap}_f(\mathsf{refl}_{\mathsf{base}}) = \mathsf{refl}_a$, implying that *every type* is a set. However, we have already seen that this is not the case: the universe $\mathcal{U}$ itself is not a set. □

The circle as an inductive type also has an uniqueness principle.

Lemma 77. *Let $A : \mathcal{U}$ and $f, g : S^1 \to A$ be such that there exist*

$$p : f(\mathit{base}) =_A g(\mathit{base}), \quad q : \mathit{ap}_f(\mathit{loop}) =_p^{\lambda x. x =_A x} \mathit{ap}_g(\mathit{loop})$$

then we have $f(x) =_A g(x)$ for all $x : S^1$.

Proof: We do, of course, induction over family $D(x) :\equiv (f(x) =_A g(x))$. We already have $f(\mathsf{base}) = g(\mathsf{base})$ by hypotheses; what we need now is $p =_{\mathsf{loop}}^{\lambda x. f(x) = g(x)} p$.

By theorem 37 we have $\mathsf{transport}^{x \mapsto f(x) = g(x)}(\mathsf{loop}, p) = (\mathsf{ap}_f(\mathsf{loop}))^{-1} \cdot p \cdot \mathsf{ap}_g(\mathsf{loop})$. Note that q is of type $\mathsf{transport}^{x \mapsto x = x}(p, \mathsf{ap}_f(\mathsf{loop})) = \mathsf{ap}_g(\mathsf{loop})$; applying theorem 36, we have

$$\big(\mathsf{transport}^{x \mapsto x = x}(p, \mathsf{ap}_f(\mathsf{loop})) = \mathsf{ap}_g(\mathsf{loop})\big) \simeq (\mathsf{ap}_f(\mathsf{loop}) \cdot p = p \cdot \mathsf{ap}_g(\mathsf{loop}))$$

and hence $p =_{\mathsf{loop}}^{\lambda x. f(x) = g(x)} p$. □

Actually, as we had the expected property of "continuity" of an application from the circle to any type —we expect that the image of a loop by a continuous application is a loop—, it turns out to be a universal property of the circle. Indeed, we have:

Lemma 78. *For any type A we have a natural equivalence*

$$(S^1 \to A) \simeq \sum_{x:A} x = x$$

Proof: We already have a function $(S^1 \to A) \to \sum_{x:A}(x = x)$, namely $f :\equiv \lambda g.(g(\mathsf{base}), \mathsf{ap}_g(\mathsf{loop}))$. To show that it is an equivalence, we show that it has contractible fibers. We remember that the fiber of this application f in a point $y : \sum_{x:A}(x = x)$ is

$$\mathsf{fib}_f(y) = \sum_{g:S^1 \to A} f(g) = y$$

The induction principle says that the fiber of any $y : \sum_{x:A} x = x$ is inhabited. We have to show that it is contractible; but this follows directly from the uniqueness principle of S^1, lemma 77. □

Finally, we have the following property. Topologically, it says that we can trace the circle from any point of it, and not only from base.

Lemma 79. *For each $x : S^1$ there exists an element $x =_{S^1} x$ which is not refl_x.*

Proof: We want to define $f : \prod_{x:S^1} x =_{S^1} x$ by induction. When $x = \mathsf{base}$ we already have loop, which we have already seen is not equal to $\mathsf{refl}_{\mathsf{base}}$.

We want now $\mathsf{transport}^{x \mapsto x = x}(\mathsf{loop}, \mathsf{loop}) = \mathsf{loop}$. But after lemma 35, this is equal to loop. Thus, our function is not $\lambda x.\mathsf{refl}_x$, since its image for base is loop. □

Corollary 80. *If a univer $\mathcal{U}$ contains the type S^1, then $\mathcal{U}$ is not a 1-type.*

Proof: As we have assumed that our universes are univalent, $(S^1 = S^1) : \mathcal{U}$ is equivalent to $S^1 \simeq S^1$. It suffices to show that $S^1 \simeq S^1$ is not a set, hence proving that $\mathsf{id}_{S^1} =_{S^1 \simeq S^1} \mathsf{id}_{S^1}$ is not a mere proposition would do.

But as $\mathsf{isequiv}(f)$ is a mere proposition for every function $f : S^1 \to S^1$, this type is equivalent to $\mathsf{id}_{S^1} =_{S^1 \to S^1} \mathsf{id}_{S^1}$: indeed, an element of $\mathsf{id}_{S^1} =_{S^1 \simeq S^1} \mathsf{id}_{S^1}$ is a pair (p, r), where $r : \mathsf{isequiv}(f) = \mathsf{isequiv}(g)$ which are both mere propositions.

Then, by function extensionality $\mathsf{id}_{S^1} =_{S^1 \to S^1} \mathsf{id}_{S^1}$ is the same as saying $\prod_{x:S^1} x = x$. But we have already seen that this is not a mere proposition. □

5.4.2 S^2

We expect to define spheres in a general way. In analogy to S^1, we define S^2 as the inductive type generated by

- a point $\mathsf{base} : S^2$,

- a 2-dimensional path $\mathsf{surf} : \mathsf{refl}_{\mathsf{base}} = \mathsf{refl}_{\mathsf{base}}$ in $\mathsf{base} = \mathsf{base}$.

Also analogously, we can state a recursion principle for S^2. For that, we need to define a sort of application ap^2_f, a 2-dimensional version of our ap_f: we have already defined it in lemma 14. Hence, the recursion principle for the 2-sphere is

$$\mathsf{rec}_{S^2} : \prod_{C:\mathcal{U}} \left(\prod_{b:C} \mathsf{refl}_b = \mathsf{refl}_b \right) \to S^2 \to C$$

such that, if $f :\equiv \mathsf{rec}_{S^2}(C, b, s)$, then

$$f(\mathsf{base}) :\equiv b, \quad \mathsf{ap}^2_f(\mathsf{surf}) := s.$$

but of course in order to state a more general induction principle we need a more general version of lemma 14, valid for dependent paths. Unsurprisingly, we will have that also. The following result immediate by path induction over r:

Lemma 81. *Suppose given $P : A \to \mathcal{U}$, $x, y : A$, $p, q : x = y$ and $r : p = q$. Then, for any $u : P(x)$ we have $\mathsf{transport}^2(r, u) : p_*(u) = q_*(u)$.*

This leads us to define the type of dependent 2-paths over r: in the above situation, if $u : P(x)$ and $v : P(y)$, then we know that we have paths $h : p_*(u) = v$ and $k : q_*(u) = v$; thus, it makes sense to ask which is the relation that holds between h and k. Indeed, we can define the type

$$(h =^P_r k) :\equiv h = \mathsf{transport}^2(r, u) \cdot k$$

and hence, we can give such a dependent version of lemma 14:

Lemma 82. *Suppose given $P : A \to \mathcal{U}$, $x, y : A$, $p, q : x = y$, $r : p = q$ and $f : \prod_{x:A} P(x)$. Then, we have $\mathsf{apd}^2_f : \mathsf{apd}_f(p) =^P_r \mathsf{apd}_f(q)$.*

Proof: Path induction over r. □

Finally we can state the induction principle of S^2:

$$\mathsf{ind}_{S^2} : \prod_{C:S^2 \to \mathcal{U}} \left(\prod_{b:C(\mathsf{base})} \mathsf{refl}_b =^C_{\mathsf{surf}} \mathsf{refl}_b \right) \to \prod_{x:S^2} C(x)$$

such that, if $f :\equiv \mathsf{ind}_{S^2}(C, b, s)$, then

$$f(\mathsf{base}) :\equiv b, \quad \mathsf{apd}^2_f(\mathsf{surf}) := s.$$

5.4.3 S^n

In previous sections we have given the definition of pointed types $\mathcal{U}_\bullet$ and types of n-loops. Remember that one such type is given by, given $A : \mathcal{U}$ and $a : A$,

$$\begin{array}{rcl} \Omega^0(A, a) & :\equiv & (A, a), \\ \Omega\ (A, a) & :\equiv & (a =_A a, \mathsf{refl}_a), \\ \Omega^{\mathsf{succ}(n)}(A, a) & :\equiv & \Omega(\Omega^n(A, a)) \end{array}$$

using this notations, we can define S^n to be higher inductive type generated by

- a point $\mathsf{base} : S^n$, and
- a path $\mathsf{loop} : \Omega^n(S^n, \mathsf{base})$

where, if we define some applications ap_f^n and apd_f^n, we can have an induction principle

$$\mathsf{ind}_{S^n} : \prod_{C:S^n \to \mathcal{U}} \left(\prod_{b:C(\mathsf{base})} \varphi =^C_{\mathsf{loop}} \varphi \right) \to \prod_{x:S^n} C(x)$$

where φ is the second component of $\Omega^{n-1}(C(\mathsf{base}), b)$. Then, if $f :\equiv \mathsf{ind}_{S^n}(C, b, s)$, then

$$f(\mathsf{base}) :\equiv b, \quad \mathsf{apd}_f^n(\mathsf{loop}) := s.$$

5.5 Push-outs

A very useful categorical tool, and indeed fundamental for the understanding of Seifert-Van Kampen's theorem, is that of push-outs.

Let $\mathscr{D}$ be the following *span* of types and functions:

$$\mathscr{D} = \begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \\ C & & \end{array}$$

A *push-out* $B \sqcup^A C$ of $\mathscr{D}$ is the higher inductive type generated by

- A function $i : B \to B \sqcup^A C$,
- a function $j : C \to B \sqcup^A C$, and
- a homotopy $h : i \circ f \sim j \circ g$

i.e., such that the diagram

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & \overset{h}{\swarrow\!\!\swarrow} & \downarrow{\scriptstyle i} \\ C & \xrightarrow{j} & B \sqcup^A C \end{array}$$

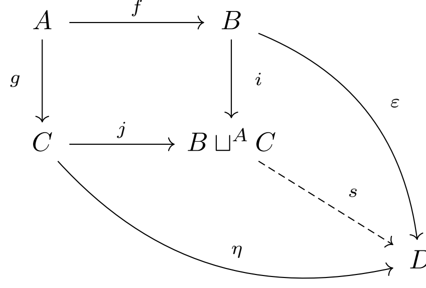
commutes by h. The induction principle of $B \sqcup^A C$ says that if we want to define a dependent function $s : \prod_{x:B\sqcup^A C} D(x)$, we need to define it first on B and on C and see that it respects the identifications:

$$\mathsf{ind}_{B\sqcup^A C} : \prod_{D:B\sqcup^A C \to \mathcal{U}} \left(\prod_{\varepsilon:\prod_{b:B} D(i(b))} \prod_{\eta:\prod_{c:C} D(j(c))} \prod_{a:A} \varepsilon(f(a)) =^D_{h(a)} \eta(g(a))) \right) \to \prod_{x:B\sqcup^A C} D$$

If $s :\equiv \mathsf{ind}_{B\sqcup^A C}(D, \varepsilon, \eta, r)$, it has defining equations,

$$\begin{aligned} s(i(b)) &:\equiv \varepsilon(b) \\ s(j(c)) &:\equiv \eta(c) \\ \mathsf{apd}_s(h(a)) &:= r(a) \end{aligned}$$

Note that the recursion principle is the well known universal property of push-outs: that given a push-out diagram and applications $\varepsilon : B \to D$, $\eta : C \to D$, then there is an application $\zeta : B \sqcup^A C \to D$ such that the following diagram commutes:



The well-known unicity of s comes from the uniqueness principle for push-outs:

Lemma 83. *If $s, s' : B \sqcup^A C \to D$ are two maps such that, for every $a : A$, $b : B$ and $c : C$,*

$$\begin{aligned} p(b) : s(i(b)) &= s'(i(b)) \\ q(c) : s(j(c)) &= s'(j(c)) \end{aligned}$$

and also the equality

$$\mathsf{apd}_s(h(a)) = \mathsf{ap}_{\mathsf{transport}^D(h(a),-)}(p(f(a))) \cdot \mathsf{apd}_{s'}(h(a)) \cdot (q(g(a)))^{-1}$$

holds, then $s = s'$.

Proof: Let $p : \prod_{b:B} s(i(b)) = s'(i(n))$ and $q : \prod_{c:C} s(j(c)) = s'(j(c))$ be as given in the hypotheses of the lemma. Note that the second hypotheses, namely

$$\mathsf{apd}_s(h(a)) = \mathsf{ap}_{\mathsf{transport}^D(h(a),-)}(p(f(a))) \cdot \mathsf{apd}_{s'}(h(a)) \cdot (q(g(a)))^{-1}$$

wants to identify $\mathsf{apd}_s(h(a))$ with $\mathsf{apd}_{s'}(h(a))$ modulo the equalities $p(b)$ and $q(c)$.

Define $E(x) :\equiv s(x) = s'(x)$ for $x : B \sqcup^A C$, we will proceed by induction over it. The functions ε and η are p and q, respectively. Now, we want to see that

$$\mathsf{transport}^{x \mapsto s(x) =_{D(x)} s'(x)}(h(a), p(f(a))) = q(g(a))$$

which, after theorem 38 is such that

$$\mathsf{transport}^{x \mapsto s(x) =_{D(x)} s'(x)}(h(a), p(f(a))) = \mathsf{apd}_s(h(a))^{-1} \cdot \mathsf{ap}_{\mathsf{transport}^D(h(a),-)}(p(f(a))) \cdot \mathsf{apd}_{s'}(h(a))$$

but $\mathsf{apd}_s(h(a)) = \mathsf{ap}_{\mathsf{transport}^D(h(a),-)}(p(f(a))) \cdot \mathsf{apd}_{s'}(h(a)) \cdot (q(g(a)))^{-1}$ by hypotheses, and so the result holds. □

Consider $A : \mathcal{U}$. The push-out of the span $\mathbf{1} \leftarrow A \to \mathbf{1}$ is called the *suspension* ΣA of A.

We give the following statement without a proof:

Lemma 84. $\Sigma\mathbf{2} \simeq S^1$.

Given a span of types and functions $\mathscr{D} = C \overset{g}{\leftarrow} A \overset{f}{\rightarrow} B$, we define the *cocone under $\mathscr{D}$ with vertex D* as the type $\mathsf{cocone}_{\mathscr{D}}(D) :\equiv B \sqcup^A C \to D$.

Remark 26. There are other ways to define a cocone, but they are seen to be equivalent thanks to Voevodsky's univalence.

5.6 Set truncations

We have already introduced propositional truncation, in which we forced a given type A into a proposition $\|A\|$, also written as $\|A\|_{-1}$. This can be generalized to forcing a given type A to be an n-type. In particular, here we will discuss the case in which $n = 0$; i.e., forcing A to be a set.

First of all, let us note that we defined propositional truncation can be thought as a higher inductive type generated by

- a function $\|-\| : A \to \|A\|$,
- a path $p(x,y) : x = y$ for each $x, y : \|A\|$.

As before, the induction principle says that if we want to construct a function $f : \prod_{x:\|A\|} C(x)$, we need just to define a function $g : \prod_{x:A} C(\|x\|)$ and a path that, for every $x, y : \|A\|$, $u : C(x)$, $v : C(y)$, inhabits $u =^C_{p(x,y)} v$; i.e.

$$\mathsf{ind}_{\|A\|} : \prod_{C:\|A\|\to\mathcal{U}} \left(\prod_{a:A} C(\|a\|)\right) \to \left(\prod_{x,y:\|A\|} \prod_{u:C(x)} \prod_{v:C(y)} u =^C_{p(x,y)} v\right) \to \prod_{x:\|A\|} C(x)$$

such that $\mathsf{ind}_{\|A\|}(C, g, \varphi, \|a\|) = g(a)$ and the dependent path between the images is given by φ.

Analogously, we will define the 0-truncation as a way to construct sets from types. It is the higher inductive type generated by

- A function $\|-\|_0 : A \to \|A\|_0$, and
- a path $u(x, y, p, q) : p = q$ for each $x, y : \|A\|_0$ and $p, q : x = y$.

The deduced induction principle is as follows:

$$\mathsf{ind}_{\|A\|_0} : \prod_{C:\|A\|_0\to\mathcal{U}} \left(\prod_{a:A} C(\|a\|_0)\right) \to \left(\prod_{x,y:\|A\|_0} \prod_{p,q:x=y} \prod_{u:C(x)} \prod_{v:C(y)} \prod_{r:u=^C_p v} \prod_{s:u=^C_q v} r =^{\lambda p.u=^C_p v}_{u(x,y,p,q)} s\right) \to \prod_{x:\|A\|_0} C(x)$$

such that, if $f :\equiv \mathsf{ind}_{\|A\|_0}(C, g, \varphi)$,

$$f(\|a\|_0) :\equiv g(a), \quad \mathsf{apd}^2_f(u(x, y, p, q)) := \varphi(x, y, p, q).$$

for all $a : A$. However, this induction principle might seem rather convoluted. A more useful one can be deduced from it:

Lemma 85. *There exists* $\mathsf{ind}_{\|A\|_0}{}' : \prod_{C:\|A\|_0\to\mathcal{U}} (\prod_{a:A} C(\|a\|_0)) \to \left(\prod_{x:\|A\|_0} \mathsf{isSet}(C(x))\right) \to \prod_{x:\|A\|_0} C(x)$ *such that* $\mathsf{ind}_{\|A\|_0}{}'(C, g, \varphi, \|a\|_0) \equiv g(a)$ *for all* $a : A$.

Proof: Suppose given $C, x, y, p, q, u, v, r, s$ as above. We want to build a dependent 2-path between r and s; i.e. a path in the fiber of $\lambda p.u =^C_p v$ between $\mathsf{transport}^{p\mapsto u=^C_p v}(u(x, y, p, q), r)$ and s. But since $C(y)$ is a set, any such 2-path does exist between two parallel paths. □

Which implies the useful (universal) property:

Lemma 86. *Let B be any set, and A an arbitrary type. Then composition $\|-\|_0 : A \to \|A\|_0$ determines an equivalence*

$$(\|A\|_0 \to B) \simeq (A \to B).$$

Proof: Applying the previous lemma when B is a constant family gives us an arrow from right to left, f. We will show that it is the quasi-inverse to composition with $\|-\|_0$.

To show it is a right-inverse, it suffices to remark that if $g : A \to B$ and $f(g) \equiv \mathsf{ind}_{\|A\|_0}{}'(B, g, \varphi)$, then $f \circ \|-\|_0 \equiv g$, by the lemma.

Let us see that it is also a left-inverse. Consider $h : \|A\|_0 \to B$, and $h' :\equiv f(h \circ \|-\|_0) : \|A\|_0 \to B$. We hope to see that they are equal; by the previous lemma we have that

$$h(\|a\|_0) = h'(\|a\|_0) = f(h \circ \|-\|_0)(a)$$

But since B is a set, $h(x) = h'(x)$ is a proposition, and thus a set. So we can apply again the previous lemma with $h(\|a\|_0) = h'(\|a\|_0)$, and hence conclude that $h(x) = h'(x)$ for all $x : \|A\|_0$ and, by function extensionality, $h = h'$. □

In the previous section we have studied push-outs. We have seen that, even if A, B and C are sets, $B \sqcup^A C$ might not be a set. For example, $\mathbf{1} \sqcup^{\mathbf{2}} \mathbf{1}$ is the circle. Hence, we may want to have an alternative "push-out" that preserves essentially the same properties, and is a set. Lemma 86 implies

Corollary 87. *Let $\mathscr{D} = C \overset{g}{\leftarrow} A \overset{f}{\rightarrow} B$ be a span of sets and functions between sets. Then, for any set E there is an equivalence*

$$\left(\left\|B \sqcup^A C\right\|_0 \to E\right) \simeq \left(B \sqcup^A C \to E\right)$$

We refer to this push-out as the *set push-out*

5.7 Quotients

Let $R : A \times A \to \mathsf{Prop}$ be a relation in A (a family of mere propositions, we might want to call it a *mere relation* since the concept is susceptible of being generalized). We are able to define the higher inductive type modeling a *set-quotient* of A by R, A/R, in a fairly intuitive way: it is generated by

- a function $q : A \to A/R$, called the *quotient application*,
- a path $q(a) = q(b)$ for every $a, b : A$ such that $R(a, b)$ is inhabited, and
- for every $x, y : A/R$, $p, q : x =_A y$ and $r, s : p = q$, a path $r = s$.

Note that the last item is just the 0-truncation of A/R, to impose that it is a set. In the rest of the section we will consider the special case in which A is a set, if we do not explicitly say the opposite.

We state the induction principle of the set-quotient type as given in [13], which is the statement that to define a (dependent) function from a quotient A/R to a family of types it suffices to define that function on A and check that it respects the R-identifications:

$$\mathsf{ind}_{A/R} : \prod_{C:A/R\to\mathcal{U}} \left(\prod_{g:\prod_{a:A} C(q(a))} \prod_{a,b:A} R(a,b) \to \left(g(a) =^C_{u(a,b,\varphi(a,b))} g(b)\right)\right) \to \prod_{x:A/R} C(x)$$

where $u(a, b, \varphi(\varphi))$ is the path between $q(a)$ and $q(b)$ that exists provided that $\varphi(a, b) : R(a, b)$. We also have that $\mathsf{ind}_{A/R}(C, g, \psi, q(a)) :\equiv g(a)$.

Lemma 88. *The function $q : A \to A/R$ is surjective.*

Proof: We want to show that $\prod_{x:A/R} \left\|\sum_{a:A} q(a) = x\right\|$. Consider $C(x) :\equiv \left\|\sum_{a:A} q(a) = x\right\|$, we want to apply induction. When $x \equiv q(a)$ for some a, the result holds. And since C in this case is a family of mere propositions, g respects the identifications. □

Of course, have the expected universal property of the quotient:

Lemma 89. *For any set B, precomposing with q is an equivalence*

$$(A/R \to B) \simeq \left(\sum_{f:A\to B} \prod_{a,b:A} R(a,b) \to (f(a) = f(b)) \right).$$

Proof: We have stated that one of the directions, from left to right, is the map $\lambda f.f \circ q$. From right to left we have the recursion principle for quotients:

$$\mathsf{rec}_{A/R} : \prod_{B:\mathcal{U}} \left(\prod_{g:A\to B} \prod_{a,b:A} R(a,b) \to (g(a) = g(b)) \right) \to A/R \to B$$

We must see that they are quasi-inverses.

Define $\overline{f} :\equiv \mathsf{rec}_{A/R}(B, f, p) : A/R \to B$, with $(\overline{f} \circ g)(a) :\equiv f(a)$ for every $a : A$: indeed, $\lambda f. - \circ f$ is well a left-inverse.

On the other direction, if $g : A/R \to B$, we need $g(x) = \overline{g \circ q}(x)$, for every $x : A/R$. But by the surjectivity of q, for every $x : A/R$ there exists $a : A$ such that $q(a) = x$:

$$g(x) = g(q(a)) = \overline{g \circ q}(q(a)) = \overline{g \circ q}(x)$$

using the recursion principle of A/R. $\square$

Of course, we can still talk about *equivalence relations*, namely applications $R : A \to A \to \mathsf{Prop}$ such that there exist three graces $\mathsf{reflexivity} : \prod_{a:A} R(a,a)$, $\mathsf{symmetry} : \prod_{a,b:A} R(a,b) \to R(b,a)$ and $\mathsf{transitivity} : \prod_{a,b,c:A} R(a,b) \times R(b,c) \to R(a,c)$. In this case, we will often use the infix notation $a \sim b :\equiv R(a,b)$.

Sometimes, however, it may be easier to think about quotients in the following way:

Lemma 90. *Let $\sim$ be a relation on a set A, and $r : A \to A$ such that $r \circ r = r$ and for all $x, y : A$, we have an equivalence $(r(x) = r(y)) \simeq (x \sim y)$. Then, the type*

$$A/\sim :\equiv \sum_{x:A} r(x) = x$$

satisfies the universal property of the set-quotient of A by $\sim$.

Remark 27. $\sim$ is an equivalence relation by the equivalence $(r(x) = r(y)) \simeq (x \sim y)$.

Remark 28. The fact that they both types satisfy the same universal property means that they are mutually interchangeable.

Proof: Consider $i : \prod_{x:A} r(r(x)) = r(x)$ from the hypotheses of idempotence for r, and define

$$\begin{array}{rccl} q: & A & \to & A/\sim \\ & x & \mapsto & (r(x), i(x)) \end{array}$$

Note that, by hypotheses, A is a set, and so $q(x) = q(y)$ if, and only if, $r(x) = r(y)$, since $r(r(x)) = r(x)$ is always a mere proposition; but, by definition, this is exactly as demanding $x \sim y$.

Let e be the function

$$\begin{array}{rccl} e: & (A/\sim \to B) & \to & \left(\sum_{g:A\to B} \prod_{x,y:A} (x \sim y) \to (g(x) = g(y))\right) \\ & f & \mapsto & (f \circ q, \varphi) \end{array}$$

where φ is the proof "given $x, y : A$ and $x \sim y$, then $q(x) = q(y)$, $f(g(x)) = f(q(y))$". We want this to be an equivalence; we exhibit now the quasi-inverse

$$\begin{array}{rccl} e': & \left(\sum_{g:A\to B} \prod_{x,y:A} (x \sim y) \to (g(x) = g(y))\right) & \to & (A/\sim \to B) \\ & (g, s) & \mapsto & \lambda(x,p).g(x) \end{array}$$

We prove that e and e' are quasi-inverses:

- Let f be a function $A/\sim \to B$. We have

$$\begin{array}{rcl} e'(e(f))(x,p) & \equiv & f(q(x)) \\ & \equiv & f((r(x), i(x))) \\ & = & f((x,p)) \end{array}$$

  as $p : r(x) = x$ and $(r(x), i(x)) = (x,p)$ since A is a set.

- In the other direction, we have

$$\begin{array}{rcl} e(e'(g,s)) & \equiv & e(g \circ \mathsf{pr}_1) \\ & \equiv & (g \circ \mathsf{pr}_1 \circ q, \varphi) \end{array}$$

  considering that B is a set, it suffices to see that $g = g \circ \mathsf{pr}_1 \circ q$. But since $r(x) \sim x$ by the idempotence of r, we have

$$g(\mathsf{pr}_1(q(x))) = g(r(x)) = g(x)$$

  since g respects the identifications made by $\sim$.

$\square$

5.7.1 The integers

Consider the quotient

$$\mathbb{Z} :\equiv (\mathbb{N} \times \mathbb{N})/\sim$$

where $\sim$ is defined as $(a, b) \sim (c, d) :\equiv (a - b = c - d)$. By lemma 90, we could also think about it as defined using the function

$$r(a,b) = \begin{cases} (a-b, 0) & \text{if } a \geq b \\ (0, b-a) & \text{if } a < b \end{cases}$$

Remark 29. $a \geq b$ is a decidable proposition since $(a \geq b) + (a < b)$, and hence this function is constructively valid.

r is immediately seen to be idempotent, and hence we have an equivalence relation. We note the canonical element $(n, 0)$ by n, while we also note the canonical element $(0, n)$ by $-n$.

We thus have this induction principle for integers:

Lemma 91. *There is*

$$\mathsf{ind}_{\mathbb{Z}} : \prod_{P:\mathbb{Z}\to\mathcal{U}} P(0) \to \left(\prod_{n:\mathbb{N}} P(n) \to P(\mathsf{succ}(n))\right) \to \left(\prod_{n:\mathbb{N}} P(-n) \to P(-\mathsf{succ}(n))\right) \to \prod_{n:\mathbb{Z}} P(n)$$

such that

$$\begin{aligned} \mathsf{ind}_{\mathbb{Z}}(P, d_0, d_+, d_-, 0) &= d_0 \\ \mathsf{ind}_{\mathbb{Z}}(P, d_0, d_+, d_-, \mathsf{succ}(n)) &= d_+ \\ \mathsf{ind}_{\mathbb{Z}}(P, d_0, d_+, d_-, -\mathsf{succ}(n)) &= d_-. \end{aligned}$$

Remark 30. Note that, despite of the previous induction principles, this is only true for propositional equalities.

Proof: For this proof, we will use $\mathbb{Z}$ as defined by $\sum_{x:\mathbb{N}\times\mathbb{N}} r(x) = x$, with r defined as above.

Let $q : \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$ be defined by the rule $x \mapsto (r(x), i(x))$ as in lemma 90. Consider $Q :\equiv P \circ q : \mathbb{N} \times \mathbb{N} \to \mathcal{U}$, then we have

$$\begin{aligned} d'_0 &: Q(0,0) \\ d'_+ &: \textstyle\prod_{n:\mathbb{N}} Q(n,0) \to Q(\mathsf{succ}(n), 0) \\ d'_- &: \textstyle\prod_{n:\mathbb{N}} Q(0,n) \to Q(0, \mathsf{succ}(n)) \end{aligned}$$

Now, by definition of q, we have that $q(n,m) \simeq q(\mathsf{succ}(n), \mathsf{succ}(m))$, and hence we can construct $e_{n,m} : Q(n,m) \simeq Q(\mathsf{succ}(n), \mathsf{succ}(m))$. Hence, we can define a function $g : \prod_{x:\mathbb{N}\times\mathbb{N}} Q(x)$ using double $\mathbb{N}$-induction over x:

$$\begin{aligned} g(0,0) &= d'_0 \\ g(\mathsf{succ}(n), 0) &= d'_+(n, g(n,0)) \\ g(0, \mathsf{succ}(m)) &= d'_-(m, g(0,m)) \\ g(\mathsf{succ}(n), \mathsf{succ}(m)) &= e_{n,m}(g(n,m)). \end{aligned}$$

Finally, we know that, by definition, $\mathsf{pr}_1 : \mathbb{Z} \to \mathbb{N} \times \mathbb{N}$ is such that $q \circ \mathsf{pr}_1 = \mathsf{id}_{\mathbb{Z}}$. Therefore, $Q \circ \mathsf{pr}_1 = P$, and so we can define

$$s : \prod_{n:\mathbb{Z}} Q(\mathsf{pr}_1(n))$$

by $f(z) :\equiv s(z, g(\mathsf{pr}_1(z)))$, and so we have our function in $\prod_{n:\mathbb{Z}} P(n)$. $\square$

Using this propositional induction principle, we can prove the following lemma, which will have important consequences in next section.

Corollary 92. *Let A be a type with $a : A$ and $p : a = a$. Then there is a dependent function $\prod_{n:\mathbb{Z}}(a =_A a)$, defined by $\lambda n.p^n$, such that*

$$\begin{aligned} p^0 &= \mathsf{refl}_a \\ p^{\mathsf{succ}(n)} &= p \cdot p^n && \text{if } n > 0 \\ p^{-\mathsf{succ}(n)} &= p^{-1} \cdot p^{-n} && \text{if } n < 0. \end{aligned}$$

Chapter 6

Seifert-Van Kampen's theorem

Finally, we will give the proof of Seifert-Van Kampen's theorem. The proof contained here is due to Michael Shulman (2013), and it can be found in [10] or [13], corresponding to which he calls "naive Van Kampen". Indeed, an even more general version can be stated and proved, but the proof is basically the same. We will not give it here, but it can also be found in the same references.

We remark that the homotopy theory given there is "synthetic" in the same way Euclidean geometry is: rather than to give analytical definitions of paths, homotopies, etc., we treat them as basic concepts (as we have, admittedly, done all this time).

Recall that for a pointed type (A, a) we defined the space of n-loops as

$$\begin{aligned} \Omega^0(A,a) &:\equiv (A,a), \\ \Omega\ (A,a) &:\equiv (a =_A a, \mathsf{refl}_a), \\ \Omega^{\mathsf{succ}(n)}(A,a) &:\equiv \Omega(\Omega^n(A,a)) \end{aligned}$$

We have already argued, at Chapter 3 that $\Omega^n(A, a)$ actually has a group law for $n \geq 1$. Nonetheless, we would like them to be sets in order to mimic the classical homotopy theory. Thus, we define:

Definition 12 (Homotopy groups)**.** Let $n \geq 1$ be a natural number and (A, a) a pointed type. We define the *homotopy groups* of A at a by

$$\pi_n(A,a) :\equiv \|\Omega^n(A,a)\|_0$$

We have already seen that, if $n \geq 2$, $\Omega^n(A, a)$ —and hence, $\pi_n(A, a)$— is an abelian group, by means of the Eckmann-Hilton theorem.

6.1 $\pi_1(S^1)$

We begin by computing the fundamental group of S^1, not because it is needed in the proof of Seifert-Van Kampen's theorem, but because it is traditionally included in first level Homotopy Theory Courses. Homotopy Type Theory offers us a way to directly mimic the classical proof in this new formalism, and the reader will immediately note that the proof is analogous to the classical one.

In fact, it is worth clarifying that we are actually going to compute $\Omega(S^1, \mathsf{base})$; as it will turn out to be $\mathbb{Z}$, as we expected it to be, and as $\mathbb{Z}$ is a set since it is the set quotient of $\mathbb{N} \times \mathbb{N}$ by a certain relation, we will have that

$$\pi_1(S^1, \mathsf{base}) = \left\|\Omega(S^1, \mathsf{base})\right\|_0 = \|\mathbb{Z}\|_0 = \mathbb{Z}$$

But the fact that $\mathbb{Z}$ is a set will have further impact: when trying to compute $\pi_2(S^1, \mathsf{base}) = \|\Omega(\mathbb{Z}, \mathsf{refl}_{\mathsf{base}})\|$ we will find $\mathbf{1}$, since the higher groupoid structure of any set is trivial. Hence, $\pi_n(S^1, \mathsf{base}) = \mathbf{1}$ for every $n \geq 2$: having computed $\pi_1(S^1, \mathsf{base})$ we will have computed any fundamental group of the circle.

Recall the last result we have given, corollary 92. By restating it for $\mathsf{loop} : \Omega(S^1, \mathsf{base})$, we define a function

$$\mathsf{loop}^n = \begin{cases} \underbrace{\mathsf{loop} \cdot \mathsf{loop} \cdot \dots \cdot \mathsf{loop}}_{n} & \text{if } n > 0 \\ \underbrace{\mathsf{loop}^{-1} \cdot \mathsf{loop}^{-1} \cdot \dots \cdot \mathsf{loop}^{-1}}_{n} & \text{if } n < 0 \\ \mathsf{refl}_{\mathsf{base}} & \text{if } n = 0 \end{cases}$$

which we might note by $\lambda n.\mathsf{loop}^n$, or simply by loop^-. This will give us one of the directions, $\mathbb{Z} \to \Omega(S^1, \mathsf{base})$; by exhibiting a quasi-inverse $\Omega(S^1, \mathsf{base}) \to \mathbb{Z}$, we will have shown that $\Omega(S^1, \mathsf{base}) \simeq \mathbb{Z}$, hence $\Omega(S^1, \mathsf{base}) = \mathbb{Z}$ applying univalence.

Such a function could be defined as follows. Note that $\mathsf{succ} : \mathbb{Z} \to \mathbb{Z}$ is an equivalence, and hence it induces a path $\mathsf{ua}(\mathsf{succ}) : \mathbb{Z} = \mathbb{Z}$, and so by applying the recursion principleof S^1 we can define a function $c : S^1 \to \mathcal{U}$ such that $c(\mathsf{base}) :\equiv \mathbb{Z}$ and $\mathsf{ap}_c(\mathsf{loop}) := \mathsf{ua}(\mathsf{succ})$. Thus, having $\mathsf{ap}_c : \Omega(S^1, \mathsf{base}) \to \Omega(\mathcal{U}, \mathbb{Z})$, we can define $g : \Omega(S^1, \mathsf{base}) \to \mathbb{Z}$ by the rule

$$g :\equiv \lambda p.\mathsf{transport}^{X \mapsto X}(\mathsf{ap}_c(p), 0)$$

where $\mathsf{transport}^{X \mapsto X}(q, -) : \mathbb{Z} \to \mathbb{Z}$ when $q : \mathbb{Z} =_{\mathcal{U}} \mathbb{Z}$, so our function is well-typed. We will now give a generalization of this morphism which will be suitable for the proof.

6.1.1 The proof

The proof of the statement will follow the $\mathsf{encode}/\mathsf{decode}$ scheme given in previous sections. Recall the idea of the classical proof: we consider the morphism $\exp : \mathbb{R} \to S^1$ as the universal covering of S^1. The fiber of each point of S^1 by exp will be an isomorphic copy of $\mathbb{Z}$; fix one point in S^1, say base. Each loop from base to itself lifts in a path between the points of its fiber; loosely speaking we consider the action of $\pi_1(S^1, \mathsf{base})$ over the fiber of base, and this is proven to be a group isomorphism between $\pi_1(S^1, \mathsf{base})$ and the fiber seen as $\mathbb{Z}$.

In Type Theory, the fiber over a point $a : A$ corresponds to the determination $B(a)$ of $B : A \to \mathcal{U}$, a type family. If the type family is $\lambda a.a_0 = a$ for a fixed $a_0 : A$, the space of paths that begin in a_0 and end in a, varying a in A; note that $B(a_0) = \Omega(A, a_0)$. Now, if we let A be S^1 and a_0 be base, what we get is the type family $x \mapsto (\mathsf{base} = x)$; this wants to imitate the space of paths turning around the loop until x is reached.

We define the universal cover of S^1 *ad hoc*:

Definition 13 (Universal cover of S^1)**.** Define $\mathsf{code} : S^1 \to \mathcal{U}$ by circle recursion, with

$$\begin{aligned} \mathsf{code}(\mathsf{base}) &:\equiv \mathbb{Z} \\ \mathsf{ap}_{\mathsf{code}}(\mathsf{loop}) &:= \mathsf{ua}(\mathsf{succ}). \end{aligned}$$

This is exactly how we defined c above. We now want to describe the action of $\Omega(S^1, \mathsf{base})$ over the fiber of codes. We have the expected property:

Lemma 93. *With the above notations, we have*

$$\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, n) = n + 1 \quad \text{and} \quad \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}, n) = n - 1.$$

Proof: We proof the first equation; the second follows from the fact that for any type family B, $\mathsf{transport}^B(p, -)$ is the quasi-inverse of $\mathsf{transport}^B(p^{-1}, -)$. By lemma 20, we have, since $\mathsf{code} = \mathsf{code} \circ \mathsf{id}_{\mathcal{U}}$,

$$\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, n) = \mathsf{transport}^{X \mapsto X}(\mathsf{ap}_{\mathsf{code}}(\mathsf{loop}), n) = \mathsf{transport}^{X \mapsto X}(\mathsf{ua}(\mathsf{succ}), n).$$

The computation rule of the univalence axiom says that

$$\mathsf{transport}^{X \mapsto X}(\mathsf{ua}(f), x) = f(x),$$

(remember that $\mathsf{transport}^{X\mapsto X}$ was also referred to as idtoeq) and so the above equation is, as desired, $\mathsf{succ}(n)$. □

Generalizing the ideas at the end of the previous section, we will now want to construct functions

$$\mathsf{encode} : \prod_{x:S^1}((\mathsf{base}=x)\to\mathsf{code}(x))$$
$$\mathsf{decode} : \prod_{x:S^1}(\mathsf{code}(x)\to(\mathsf{base}=x))$$

which will lead to the previously defined g and loop^-, respectively, when x is base.

Definition 14. We define $\mathsf{encode} : \prod_{x:S^1}((\mathsf{base}=x)\to\mathsf{code}(x))$ as

$$\mathsf{encode}_x(p) :\equiv \mathsf{transport}^{\mathsf{code}}(p,0).$$

Remark 31. Note that the argument x is already implicit in the type of p, so we could avoid to explicitly write it.

This is a translation into Type Theory of the action of loops on the circle on $\mathbb{R}$: we lift a path over the universal cover and make it begin at 0. When this path is loop, we have already stated what happens in lemma 93. Moreover, the functoriality of $\mathsf{transport}^{\mathsf{code}}$ —lemma 19— states that, when we have a composition of paths, we will have a composition of transport functions. In particular, when the paths are loops, applying lemma 93 will give us a group homomorphism!

Now we would want to define $\mathsf{decode} : \prod_{x:S^1}(\mathsf{code}(x)\to(\mathsf{base}=x))$ by circle induction in such a way that $\mathsf{decode}(\mathsf{base}) :\equiv \mathsf{loop}^-$. In order to do so, we first have to check whether loop^- is such that $\mathsf{loop}^- =^{\lambda x.\mathsf{code}(x)\to(\mathsf{base}=x)}_{\mathsf{loop}} \mathsf{loop}^-$.

Lemma 94. *$\mathsf{transport}^{\lambda x.\mathsf{code}(x)\to(\mathsf{base}=x)}(\mathsf{loop},\mathsf{loop}^-) = \mathsf{loop}^-$.*

Proof: We want to prove that such a path exists. We have

$$\begin{aligned}
\mathsf{transport}^{\lambda x.\mathsf{code}(x)\to(\mathsf{base}=x)}(\mathsf{loop},\mathsf{loop}^-) &= \mathsf{transport}^{\lambda x.(\mathsf{base}=x)}(\mathsf{loop},\mathsf{loop}^-(\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^-1,-))) && \text{by eq. 3.1}\\
&= \mathsf{loop}^-(\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^-1,-))\cdot\mathsf{loop} && \text{by lemma 35}\\
&= \lambda n.\mathsf{loop}^-(n-1)\cdot\mathsf{loop} && \text{by lemma 93}\\
&= \lambda n.\mathsf{loop}^n && \text{by lemma 91}\\
&= \mathsf{loop}^-.
\end{aligned}$$

□

Definition 15. We define $\mathsf{decode} : \prod_{x:S^1}(\mathsf{code}(x)\to(\mathsf{base}=x))$ by circle induction, with $\mathsf{decode}(\mathsf{base}) :\equiv \mathsf{loop}^-$ and the path defined in the above proof. We write decode_x for $\mathsf{decode}(x)$

Thus, we can show

Theorem 95. *$(\mathsf{base}=x)\simeq\mathsf{code}(x)$ for all $x:S^1$.*

Proof: We need only to show that decode and encode are quasi-inverses.

We begin by showing that, for all $x:S^1$ and for all paths $p:\mathsf{base}=x$,

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

We proceed by path induction over p. If $x\equiv\mathsf{base}$ and $p\equiv\mathsf{refl}_{\mathsf{base}}$,

$$\begin{aligned}
\mathsf{decode}_{\mathsf{base}}(\mathsf{encode}_x(\mathsf{refl}_{\mathsf{base}})) &\equiv \mathsf{decode}_{\mathsf{base}}(\mathsf{transport}^{\mathsf{code}}(\mathsf{refl}_{\mathsf{base}},0))\\
&= \mathsf{decode}(\mathsf{base},0)\\
&\equiv \mathsf{loop}^0\\
&= \mathsf{refl}_{\mathsf{base}}.
\end{aligned}$$

In the other direction, we want to show that, for any $x:S^1$ and $c:\mathsf{code}(x)$, we have

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

We proceed by circle induction. Since $\mathsf{code}(\mathsf{base})$ is $\mathbb{Z}$, a set, the path "over loop" will immediately exist: the dependent function we want to construct has codomain, indeed, $\mathsf{encode}_x(\mathsf{decode}_x(c)) = c$, and hence its elements are paths, and two parallel paths are always equal over a set.

Hence, it will be sufficient to show that, for all $n : \mathbb{Z}$, the equation

$$\mathsf{encode}_{\mathsf{base}}(\mathsf{loop}^n) = n$$

holds. Indeed, we proceed now by the induction principle of $\mathbb{Z}$ —lemma 91. We have

- For $n = 0$, $\mathsf{encode}_{\mathsf{base}}(\mathsf{refl}_{\mathsf{base}}) \equiv \mathsf{transport}^{\mathsf{code}}(\mathsf{refl}_{\mathsf{base}}, 0) = 0$.
- For $\mathsf{succ}(n)$ assuming that it is true for n,

$$\begin{aligned}
\mathsf{encode}_{\mathsf{base}}(\mathsf{loop}^{n+1}) &= \mathsf{encode}(\mathsf{loop}^n \cdot \mathsf{loop}) \\
&= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^n \cdot \mathsf{loop}) \\
&= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^n, 0)) && \text{by lemma 19} \\
&= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, n) && \text{by inductive hypotheses} \\
&= n + 1. && \text{by lemma 93}
\end{aligned}$$

- For $-\mathsf{succ}(n)$ assuming it holds for $-n$ we proceed analogously.

□

Corollary 96. *$\Omega(S^1, \mathit{base}) \simeq \mathbb{Z}$; hence, by univalence,*

$$\Omega(S^1, \mathit{base}) = \mathbb{Z}.$$

The identification is a group isomorphism.

Proof: The first part is done by instantiating the previous theorem. By induction, we can easily show that encode and decode are both group homomorphisms. □

Finally,

Corollary 97. *$\pi_1(S^1, \mathit{base}) = \mathbb{Z}$ and $\pi_n(S^1, \mathit{base}) = \mathbf{1}$ for $n \geq 2$.*

6.2 Seifert-Van Kampen's theorem

Seifert-Van Kampen's theorem is a classical tool in homotopy theory: it computes the fundamental group of a push-out of topological spaces. The result turns out to be the push-out of fundamental group, providing that the base point of them is in the center of the span. The scheme is, abstractly speaking, analogous to the previous proof: we will use the $\mathsf{encode}/\mathsf{decode}$ method, and so we will describe the desired type as a type $\mathsf{code}(x)$, which we hope will be easier to understand.

Actually, the theorem we will state and prove is a bit more general than that: it holds for *fundamental groupoids*, and hence classical Van Kampen's will be a special case. Hence, first of all we need to define what a fundamental groupoid is.

We write $\Pi_1 X : X \to X \to \mathcal{U}$ the type family given by

$$\Pi_1 X(x, y) :\equiv \|x = y\|_0$$

Then, of course we have that $\Pi_1 X(x, x) \equiv \pi_1(X, x)$. We also have groupoid operations as follow

$$\begin{aligned}
(-, -) &: \Pi_1 X(x, y) \to \Pi_1 X(y, z) \to \Pi_1 X(x, z) \\
(-)^{-1} &: \Pi_1 X(x, y) \to \Pi_1 X(y, x) \\
\mathsf{refl}_x &: \Pi_1 X(x, x) \\
\mathsf{ap}_f &: \Pi_1 X(x, y) \to \Pi_1 Y(f(x), f(y))
\end{aligned}$$

Lemma 98. *For all $x : X$, $p : x = y$ the following equation holds.*

$$\mathit{transport}^{y \mapsto \Pi_1 X(x,y)}(p, \|q\|_0) = \|q \cdot p\|_0 .$$

Proof: It follows directly from lemma 21, with $P(y) :\equiv (x = y)$ and $Q(y) :\equiv \|x = y\|_0$, and then applying lemma 35. $\square$

Hence, we will usually abuse the notation, using the same letter for the path and its truncated version in $\Pi_1 X$.

Consider now a span of types and functions $\mathscr{D} = C \overset{g}{\leftarrow} C \overset{f}{\rightarrow} B$, and its push-out

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ {\scriptstyle g}\downarrow & \overset{h}{\swarrow\!\!\swarrow} & \downarrow{\scriptstyle i} \\ C & \xrightarrow{\quad j \quad} & B \sqcup^A C \end{array}$$

For simplicity, we will write P for $B \sqcup^A C$. Now, we want to define $\mathsf{code} : P \to P \to \mathcal{U}$, and we will proceed by double induction over it. Recall that the induction principle for push-outs is as follows:

$$\mathsf{ind}_{B \sqcup^A C} : \prod_{D : B \sqcup^A C \to \mathcal{U}} \left(\prod_{\varepsilon : \prod_{b:B} D(i(b))} \prod_{\eta : \prod_{c:C} D(j(c))} \prod_{a:A} \varepsilon(f(a)) =^D_{h(a)} \eta(g(a)) \right) \to \prod_{x : B \sqcup^A C} D$$

with

$$\begin{array}{rcl} s(i(b)) & :\equiv & \varepsilon(b) \\ s(j(c)) & :\equiv & \eta(c) \\ \mathsf{apd}_s(h(a)) & := & r(a) \end{array}$$

We should note that the idea of the proof is analogous to the classical one: given a path $x = y$, we want to decompose it in a sequence of sub-paths, in a way that each interior end-point belongs to the image of A by $i \circ f$ into P, and each intermediate sub-paths is completely contained into the image of B by i or the image of C by j. With this in mind, we define:

- $\mathsf{code}(ib, ib')$ as the set-quotient of sequences of the form

  $$(b, p_0, x_1, q_1, y_1, p_1, x_2, q_2, y_2, p_2, \dots, y_n, p_n, b')$$

  such that $n : \mathbb{N}$ and

  - $x_k, y_k : A$ for $0 < k \leq n$;
  - if $n = 0$, $p_0 : \Pi_1 B(b, b')$, otherwise, $p_0 : \Pi_1 B(b, fx_1)$ and $p_n : \Pi_1 B(fy_n, b')$;
  - $p_k : \Pi_1 B(fy_k, fx_{k+1})$ for $0 < k < n$, and
  - $q_k : \Pi_1 C(gx_k, gy_k)$ for $0 < k \leq n$.

  Note that the type of these recurrences may be harder to explicitly define than to understand, and so we do not really attempt to do it here. We define the relation in the quotient as generated by the equalities

  $$\begin{array}{rcl} (\dots, q_k, y_k, \mathsf{refl}_{fy_k}, y_k, q_{k+1}, \dots) & = & (\dots, q_k \cdot q_{k+1}, \dots) \\ (\dots, p_k, y_k, \mathsf{refl}_{gx_k}, x_k, p_{k+1}, \dots) & = & (\dots, p_k \cdot p_{k+1}, \dots). \end{array}$$

  Of course, the idea is the one we have explained before: each path p_k or q_k is contained entirely in B or in C. The above relation states that, if we want to consider to consecutive paths in the same type, say, B, we just need to state that the intermediate path in C is a reflexivity path.

- For $\mathsf{code}(jc, jc')$ we just need to reverse the roles of x and y, and of p and q.

- For $\mathsf{code}(ib, jc')$ we proceed as for $\mathsf{code}(ib, ib')$, but changing the parity in such a way that the sequence ends in C. We define $\mathsf{code}(jc, ib')$ analogously.

- For the homotopy between $\mathsf{code}(ib, ifa)$ and $\mathsf{code}(ib, jga)$ we impose an equivalence by two maps

$$(\ldots, y_n, p_n, fa) \mapsto (\ldots, y_n, p_n, a, \mathsf{refl}_{ga}, ga)$$

and

$$(\ldots, x_n, q_n, ga) \mapsto (\ldots, x_n, q_n, a, \mathsf{refl}_{fa}, fa)$$

Note that, as we said, we are abusing the notation by saying that $\mathsf{refl}_{fa} : \Pi_1 B(fa, fa)$ and $\mathsf{refl}_{ga} : \Pi_1 C(ga, ga)$. Now we should see via the induction principle of the quotient that these functions can be defined; i.e. that they respect the relations. We will not do this since the proof follows by definition.

To show that they are indeed quasi-inverses, we give one composition

$$(\ldots, y_n, p_n, fa) \mapsto (\ldots, y_n, p_n, a, \mathsf{refl}_{ga}, ga) \mapsto (\ldots, y_n, p_n, a, \mathsf{refl}_{ga}, a, \mathsf{refl}_{fa}, fa)$$

which is equal to $(\ldots, y_n, p_n, fa)$ using the relations of the quotient. The other is done in the exact some way.

- We impose equivalences

$$\begin{aligned} \mathsf{code}(jc, ifa) &\simeq \mathsf{code}(jc, jga) \\ \mathsf{code}(ifa, ib') &\simeq \mathsf{code}(jga, ib') \\ \mathsf{code}(ifa, jc') &\simeq \mathsf{code}(jga, jc') \end{aligned}$$

in a very similar way. For the last two, this process will be done at the beginning of the sequence, rather than in the end.

- To finish the induction we need to impose that, for all $a, a' : A$, the following diagram

$$\begin{array}{ccc} \mathsf{code}(ifa, ifa') & \longrightarrow & \mathsf{code}(jga, ifa') \\ \downarrow & & \downarrow \\ \mathsf{code}(ifa, jga') & \longrightarrow & \mathsf{code}(jga, jga') \end{array}$$

commutes.

With these definitions, we are able to characterize the action of functions and the transport over code:

Lemma 99. *The following equations hold*

- *if* $w : \Pi_1 A(x_k, x'_k)$,

$$(\ldots, p_{k-1} \cdot \mathsf{ap}_f(w), x'_k, q_k, \ldots) = (\ldots, p_{k-1}, x_k, \mathsf{ap}_g(w) \cdot q_k, \ldots);$$

- *if* $w : \Pi_1 A(y_k, y'_k)$,

$$(\ldots, q_k \cdot \mathsf{ap}_g(w), y'_k, p_k, \ldots) = (\ldots, q_k, y_k, \mathsf{ap}_f(w) \cdot p_k, \ldots).$$

Proof: It suffices to apply path induction when $x_k \equiv x'_k$ and $w \equiv \mathsf{refl}_{x_k}$, and when $y_k \equiv y'_k$ and $w \equiv \mathsf{refl}_{y_k}$. □

Remark 32. This innocent-looking lemma will lead to $\mathsf{code}(u, u)$ working as the amalgamated product of the fundamental groups of B and C.

Lemma 100. *The following equations hold:*

- *if* $p : b =_B b'$ *and* $u : P$, *then*

$$\mathsf{transport}^{x \mapsto \mathsf{code}(u, ib)}(p, (\ldots, y_n, p_n, b)) = (\ldots, y_n, p_n \cdot p, b');$$

- *if* $q : c =_C c'$ *and* $u : P$*, then*

$$\mathit{transport}^{x \mapsto \mathit{code}(u, jc)}(q, (\ldots, x_n, q_n, c)) = (\ldots, x_n, q_n \cdot q, c').$$

Proof: It follows directly from path induction over p and q. ☐

Lemma 101. *If* $a : A$ *and* $u : P$*, we have*

$$\mathit{transport}^{v \mapsto \mathit{code}(u,v)}(ha, (\ldots, y_n, p_n, fa)) = (\ldots, y_n, p_n, a, \mathit{refl}_{ga}, ga).$$

Proof: It follows from the definition of code: the action on the path-constructor computes to a dependent path, i.e. a transport equation. ☐

Finally, as in the proof of theorem 41, we construct a function

$$r : \prod_{u:P} \mathsf{code}(u, u)$$

by push-out induction over u, with

$$\begin{array}{rcl} rib & :\equiv & (b, \mathsf{refl}_b, b) \\ rjc & :\equiv & (c, \mathsf{refl}_c, c) \end{array}$$

and the equality

$$\mathsf{transport}^{u \mapsto \mathsf{code}(u,u)}(ha, ha), (fa, \mathsf{refl}_{fa}, fa)) = (ga, \mathsf{refl}_{ga}, a, \mathsf{refl}_{fa}, a, \mathsf{refl}_{ga}, ga)$$

as given by lemma 101, and thus

$$\mathsf{transport}^{u \mapsto \mathsf{code}(u,u)}(ha, ha), (fa, \mathsf{refl}_{fa}, fa)) = (ga, \mathsf{refl}_{ga}, ga)$$

by the definitional relations in code.

6.2.1 The theorem

We finally proceed to state and prove Seifert-Van Kampen's theorem.

Theorem 102 (Seifert-Van Kampen)**.** *For all* $u, v : P$ *there is an equivalence*

$$\Pi_1 P(u, v) \simeq \mathit{code}(u, v).$$

Proof: We proceed by defining two functions encode and decode and show that they are quasi-inverses. We will now define them:

To define encode : $\Pi_1 P(u, v) \to \mathsf{code}(u, v)$ it suffices, since code is a set, by lemma 86, to define a function $(u =_P v) \to \mathsf{code}(u, v)$. Then, let $p : u =_P v$, and define

$$\mathsf{encode}(p) :\equiv \mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(p, r(u)).$$

In the other direction, we define decode : $\mathsf{code}(u, v) \to \Pi_1 P(u, v)$ by using double push-out induction over $u, v : P$. We give, as usual, the definition when $u \equiv ib$ and $v \equiv ib'$, but the other cases are analogous. We define

$$\mathsf{decode}(b, p_0, x_1, q_1, y_1, p_1, \ldots, y_n, p_n, b') :\equiv (ip_0) \cdot h(x_1) \cdot (jq_1) \cdot \cdot (h(y_0))^{-1} \cdot (ip_1) \cdot \cdots \cdot (h(y_n))^{-1} \cdot (ip_n)$$

which is well defined since $ip_0 : ib =_B ifx_1$, $h(x_1) : ifx_1 = jgx_1$, $jq_1 : jgx_1 = jgy_1$,

$$\begin{array}{rlcl} ip_0 & : ib & = & ifx_1, \\ h(x_1) & : ifx_1 & = & jgx_1, \\ jq_1 & : jgx_1 & = & jgy_1, \end{array}$$

and so on. We need now to show that the definition respects the set-quotient relations, and this will be essentially a consequence of the naturality of h. Indeed: let $a : A$, we want a dependent path

$$\mathsf{transport}^{v\mapsto \Pi_1 P(u,v)}(ha, \mathsf{decode}(\ldots, y_n, p_n, fa)) = \mathsf{decode}(\ldots, y_n, p_n, a, \mathsf{refl}_{ga}, ga)$$

so, by applying lemma 98, we get

$$\begin{aligned}
\mathsf{transport}^{v\mapsto \Pi_1 P(u,v)}(ha, \mathsf{decode}(\ldots, y_n, p_n, fa)) &= \mathsf{decode}(\ldots, y_n, p_n, fa) \cdot ha \\
&= (hy_n)^{-1} \cdot ip_n \cdot ha \\
&= (hy_n)^{-1} \cdot ip_n \cdot ha \cdot \mathsf{refl}_{jga} \\
&= (hy_n)^{-1} \cdot ip_n \cdot ha \cdot j\mathsf{refl}_{ga} \\
&= \mathsf{decode}(\ldots, y_n, p_n, a, \mathsf{refl}_{ga}, ga)
\end{aligned}$$

as we wanted to show.

We have defined encode and decode; now we will show that they are quasi inverses. We begin by showing that the composition

$$\Pi_1 P(u,v) \xrightarrow{\mathsf{encode}} \mathsf{code}(u,v) \xrightarrow{\mathsf{decode}} \Pi_1 P(u,v)$$

is the identity. Since $\mathsf{code}(u,v) \to \Pi_1(u,v)$ is a set (because it is a type of functions between sets, lemma 51), we can begin with a path $p : u =_P v$, invoking lemma 86. We proceed by path induction over p.

Suppose $u \equiv v$, $p \equiv \mathsf{refl}_u$; then, we have

$$\mathsf{encode}(\mathsf{refl}_u) \equiv \mathsf{transport}^{v\mapsto \mathsf{code}(u,v)}(\mathsf{refl}_u, r(u)) \equiv r(u)$$

and thus we want to show that $\mathsf{decode}(r(u)) = \mathsf{refl}_u$; we do this by induction over u in P. Let $D(u) :\equiv \mathsf{decode}(r(u)) = \mathsf{refl}_u$; we have

$$\begin{aligned}
\mathsf{decode}(rib) &\equiv \mathsf{decode}(b, \mathsf{refl}_b, b) \\
&\equiv i(\mathsf{refl}_b) = \mathsf{refl}_{ib} \\
\mathsf{decode}(rjc) &\equiv \mathsf{decode}(c, \mathsf{refl}_c, c) \\
&\equiv j(\mathsf{refl}_c) = \mathsf{refl}_{jc}
\end{aligned}$$

and as $\mathsf{code}(u,v)$ and $\Pi_1(u,v)$ are sets, $\mathsf{decode}(r(u)) = \mathsf{refl}_u$ is a mere proposition; hence, the dependent path

$$\mathsf{transport}^{u\mapsto \mathsf{decode}(ru)=\mathsf{refl}_u}(ha, \varphi_{r(ifa)}) = \varphi_{r(jga)}$$

exists. Therefore, we have finally shown this direction.

Let us see now the other one; *c'est à dire*, that the composition

$$\mathsf{code}(u,v) \xrightarrow{\mathsf{decode}} \Pi_1 P(u,v) \xrightarrow{\mathsf{encode}} \mathsf{code}(u,v)$$

As we have done before, we will show this by using double induction over $u, v : P$. We remark that, as we have just done, the existence of dependent paths needed in the induction is just immediate because $\Pi_1 P(u,v)$ and $\mathsf{code}(u,v)$ are sets. So we just have to show the property for the couples $(ib, ib'), (ib, jc'), (jc, ib'), (jc, jc')$. As usual, we consider the first one: if $u \equiv ib$ and $v \equiv ib'$, we have

$$\mathsf{decode}(b, p_0, x_1, q_1, y_1, p_1, \ldots, y_n, p_n, b') \equiv i(p_0) \cdot h(x_1) \cdot j(q_1) \cdot (h(y_1))^{-1} \cdot i(p_1) \cdot \cdots \cdot (h(y_n))^{-1} \cdot i(p_n)$$

by definition. We apply now encode to it; we get

$$\begin{aligned}
\big(i(p_0) \cdot h(x_1) \cdot j(q_1) \cdot (h(y_1))^{-1} \cdot i(p_1) \cdot \cdots \cdot (h(y_n))^{-1} \cdot i(p_n)\big)_* (r(ib)) &= \\
(i(p_n))_* \cdot (h(y_n))_* \cdot \cdots \cdot (i(p_n))_* \cdot (h(y_1))_*^{-1} \cdot (j(q_1))_* \cdot (h(x_1))_* \cdot (i(p_0))_*(b, \mathsf{refl}_b, b) &= \\
(i(p_n))_* \cdot (h(y_n))_* \cdot \cdots \cdot (i(p_n))_* \cdot (h(y_1))_*^{-1} \cdot (j(q_1))_* \cdot (h(x_1))_* \cdot (b, p_0, fx_1) &= \\
(i(p_n))_* \cdot (h(y_n))_* \cdot \cdots \cdot (i(p_n))_* \cdot (h(y_1))_*^{-1} \cdot (j(q_1))_* \cdot (b, p_0, x_1, \mathsf{refl}_{gx_1}, gx_1) &= \\
(i(p_n))_* \cdot (h(y_n))_* \cdot \cdots \cdot (i(p_n))_* \cdot (h(y_1))_*^{-1} \cdot (j(q_1))_* \cdot (b, p_0, x_1, q_1, y_1) &=
\end{aligned}$$

and so it goes. By $\mathbb{N}$-induction over n, we get finally

$$\mathsf{encode}\left(\mathsf{decode}(b, p_0, x_1, q_1, y_1, p_1, \ldots, y_n, p_n, b')\right) = (b, p_0, x_1, q_1, y_1, p_1, \ldots, y_n, p_n, b')$$

which is exactly what we wanted. □

At this point the reader might feel a bit disappointed. Seifert-Van Kampen's theorem was supposed to help us identify the fundamental group of a space, not to hide the problem inside an artificial construct, code, which does not, at least at simple sight, give us any information about it.

The follow two examples will show that it is possible to use it and actually get some interesting information. We will finish by give some remarks about algebraic structure in the theorem.

Example 9 (The circle)**.** As we have stated, $\Sigma\mathbf{2} \simeq S^1$. Thus, we can try to apply Van Kampen's theorem to it, where now $A \equiv \mathbf{2}$ and B and C both are $\mathbf{1}$. That is, we have the situation

$$\begin{array}{ccc} \mathbf{2} & \xrightarrow{f} & \mathbf{1} \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle i} \\ \mathbf{1} & \xrightarrow{j} & P \end{array}$$

Suppose we want to describe $\mathsf{code}(i(*), i(*))$, which consists of sequences x of the form

$$(*, p_0, x_1, q_1, y_1, p_1, x_2, q_2, y_2, p_2, \ldots, y_n, p_n, *)$$

where each p_k belongs to $\Pi_1\mathbf{1}(fy_k, fx_{k+1})$, which is to say $\|* = *\|_0$; as $\mathbf{1}$ is a set, $p_k = \mathsf{refl}_*$. The same is true for each q_k. Hence, the sequence x is actually given by

$$(*, x_1, y_1, x_2, y_2, \ldots, x_n, y_n, *)$$

with reflexivity paths in between, where $x_k, y_k : \mathbf{2}$ for all k. But if $x_k = y_k$ for any k, the relations in the definition of code cancel both, so it is safe to assume that $x_k \neq y_k$ for every k. There are then only two equal sequences with the same length: the one which begins in $x_1 \equiv 0_\mathbf{2}$, and the one which begins with $x_1 \equiv 1_\mathbf{2}$. The sequence $(*, \mathsf{refl}_*, *)$ is unique; if we map it to 0, and choose coherently whether to map a sequence to its length or minus its length, we have an obvious equivalence of sets

$$\pi_1(S^1, i(*)) \simeq \mathsf{code}(i(*), i(*)) \simeq \mathbb{Z}$$

although this is, *a priori*, only a bijection.

A classical application of Seifert-Van Kampen's theorem is to compute the fundamental group of the wedge product of two spaces. In its most popular instantiation, it computes the fundamental group of a rose of n petals.

Example 10 (Wedge product)**.** Consider the types B and C, its *wedge product* is the push-out

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{f} & B \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle i} \\ C & \xrightarrow{j} & P \end{array}$$

Here, as the opposite of previous situation, all elements in the sequences of $\mathsf{code}(i(b), i(b'))$ are $* : \mathbf{1}$, and thus the only information is given by the loops. We have elements of the form

$$(p_0, q_1, p_1, \ldots, q_n, p_n)$$

with stars between them. This is, at least intuitively, the free product of $\pi_1(B, f(*))$ and $\pi_1(C, g(*))$.

6.2.2 Structure

We have seen, at least for the moment, an equivalence as types. But as we expected, functions encode and decode are morphisms in the category of groupoids (this is a purely semantic statement); they respect the groupoid laws.

We already knew that $\Pi_1 P(u,v)$ was a groupoid —indeed, we constructed it as one. But we have to show that $\mathsf{code}(u,v)$ also behaves like one. Fortunately, this is easy.

We define the concatenation of sequences as an operation

$$::_v : \mathsf{code}(u,v) \times \mathsf{code}(v,w) \to \mathsf{code}(u,w)$$

for every $u,v,w : P$. Of course, we will define it by push-out induction, this time over v: we define $D(b) :\equiv \mathsf{code}(u,v) \times \mathsf{code}(v,w) \to \mathsf{code}(u,w)$. If $v :\equiv ib$, we have sequences

$$(\dots, y_n, p_n, b)$$

and

$$(b, p'_0, x'_1, \dots)$$

which we merge into

$$(\dots, y_n, p_n, b, \mathsf{refl}_{gb}, b, p'_0, x'_1, \dots) = (\dots, y_n, p_n \cdot p'_0, x'_1, \dots).$$

The case for $v :\equiv jc$ is analogous. Now, if we have $u :\equiv ifa$, we have to see that our definition respects products. But it does by lemma 101 and its analogous for family $u \mapsto \mathsf{code}(u,v)$. Recall the equation 3.1, and name $A(v) :\equiv \mathsf{code}(u,v) \times \mathsf{code}(v,w)$ and $B :\equiv \mathsf{code}(u,w)$. We then have

$$\mathsf{transport}^{v \mapsto A(v) \to B}(ha, ::_{ifa}) = \mathsf{transport}^{B}(ha, ::_{ifa}\ (\mathsf{transport}^{v \mapsto A(v)}((ha)^{-1}, -)))$$

which we want to be equal to $::_{jga}$. But this is straightforward, since we recall that B actually does not depend at all on v. It can also be seen to be associative.

We can proceed also to formally define the inverses, but we will not. Essentially, given a sequence

$$(\dots, y_n, p_n, b)$$

in $\mathsf{code}(u, ib)$, its inverse is a sequence

$$(b, p_n^{-1}, y_n, \dots)$$

in $\mathsf{code}(ib, u)$, with the same points but all paths reversed.

Finally, if we consider $\mathsf{code}(u,u)$ we actually have a group. What remains is to define an identity, e_u for each u, but it will also come (as everything does) by induction with $D(u) :\equiv \mathsf{code}(u,u)$. For $u :\equiv ib$, our inverse is

$$e_{ib} :\equiv (b, \mathsf{refl}_b, b)$$

and of course

$$e_{jc} :\equiv (c, \mathsf{refl}_c, c)$$

for $\mathsf{code}(jc, jc)$. We want also to check whether

$$\mathsf{transport}^{u \mapsto \mathsf{code}(u,u)}((ha, ha), e_{ifa}) = e_{jga}$$

which is as true as it was when we defined $r : \prod_{u:P} \mathsf{code}(u,u)$. Note that the product of two inverses gives, by the definition of the code family, the identities defined here.

Thus, we can state the following results

Lemma 103. *encode respects the product and the inverses. If $u \equiv v$, it also respects the identity.*

Proof: Let $p : \Pi_1 P(u,v)$ and $q : \Pi_1 P(v,w)$ be two truncated paths. We want to know whether

$$\mathsf{encode}_{u,w}(p \cdot q) = \mathsf{encode}_{u,v}(p) ::_v \mathsf{encode}_{v,w}(q)$$

or not, where the subindexes denote which instantiation of the encode family are we using. But by definition, this is the same as asking if

$$\mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(p \cdot q, r(u)) = \mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(p, r(u)) ::_v \mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(q, r(v))$$

but by definition of the product in code, this is actually

$$\mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(p \cdot q, r(u)) = \mathsf{transport}^{v \mapsto \mathsf{code}(u,v)} \left(q, \mathsf{transport}^{v \mapsto \mathsf{code}(u,v)}(p, r(u))\right)$$

which is true by functoriality; i.e. lemma 19.

As usual, to show it preserves the inverses we will show that it preserves the identity in $\mathsf{code}(u,u)$. Now, the identity in $\Pi_1 P(u,u)$ is refl_u; so we want to know if

$$\mathsf{transport}^{u \mapsto \mathsf{code}(u,v)}(\mathsf{refl}_u, u) = e_u$$

is the identity or not. But of course this is true: by induction, with family $D(u) :\equiv \mathsf{transport}^{u \mapsto \mathsf{code}(u,v)}(\mathsf{refl}_u, u) = e_u$. The cases for $u :\equiv ib$ and $u :\equiv jc$ are trivial by the definition of $r(u)$, and to show that it preserves the push-out relation it suffices to remark that $D(u)$ is a mere proposition since $\Pi_1(u,v)$ and $\mathsf{code}(u,v)$ are always sets. $\square$

Lemma 104. *decode respects the product and the inverses. If $u \equiv v$, it also respects the identity.*

Proof: This follows directly by definition and the functoriality of transport. $\square$

And the groups?

If we base the groupoids in the same point they become groups. Thus, we have the diagram

$$\begin{array}{ccc} \pi_1(A,a) & \xrightarrow{\mathsf{ap}_f} & \pi_1(B,fa) \\ \big\downarrow{\scriptstyle \mathsf{ap}_g} & & \big\downarrow{\scriptstyle \mathsf{ap}_i} \\ \pi_1(C,ga) & \xrightarrow{\mathsf{ap}_j} & \pi_1(P,ifa) \end{array}$$

By the univalence axiom, the fact that there exists an equivalence $\pi_1(P,ifa) \simeq \pi_1(P,jga)$ leads to a propositional equality. The same is true for the identification with $\mathsf{code}(ifa,ifa)$.

Now, by lemma 99, paths in A can be regarded indistinctly in B or in C up to homotopy (but in the first fundamental group, of course, we work up to homotopy!). Thus, $\mathsf{code}(ifa,ifa)$ *does* work as if it was the amalgamated product of the space of paths in B and the space of paths in C. Therefore, it recovers the Seifert-Van Kampen theorem that we all know and love.

Bibliography

[1] Thierry Coquand, *The paradox of trees in Type Theory*, BIT vol. 32, 1991.

[2] Ilia Itenberg, "Groupe Fondamental et Revêtements", https://webusers.imj-prg.fr/~ilia.itenberg/enseignement/groupe_fondamental.pdf, 2016.

[3] Allen Hatcher, *Algebraic Topology*, http://www.math.cornell.edu/~hatcher/AT/AT.pdf, 2001.

[4] Peter John Hilton and Urs Stammbach, *A course in Homological Algebra* (first edition), Graduate Texts in Mathematics 4, Springer-Verlag, 1971.

[5] Saunders Maclane, *Categories for the working Mathematician* (second edition), Graduate texts in Mathematics, Springer, 1998.

[6] Barry Mazur, "When is one thing equal to some other thing?", http://www.math.harvard.edu/~mazur/preprints/when_is_one.pd, 2008.

[7] Charles F. Miller III, "Decision problems for groups - survey and reflections", http://www.ms.unimelb.edu.au/~cfm/papers/paperpdfs/msri_survey.all.pdf

[8] Joseph J. Rotman, *An introduction to the theory of groups* (fourth edition), Springer-Verlag, Berlin, 1995. http://shi.matmor.unam.mx/algebra/Rotman%20J.J.%20Introduction%20to%20the%20theory%20of%20groups.pdf

[9] Barnaby Sheppard, *The Logic of Infinity*, Cambridge University Press, 2014.

[10] Michael Shulman and Kuen-Bang Hou, "Kuen-Bang Hou", http://home.sandiego.edu/~shulman/papers/vankampen.pdf, 2016.

[11] Stanford Encyclopedia of Philosophy, "Intuitionistic Type Theory", http://plato.stanford.edu/entries/type-theory-intuitionistic/, 2016

[12] Stanford Encyclopedia of Philosophy, "Type Theory", http://plato.stanford.edu/entries/type-theory/, 2014.

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

[14] Niccolò Veltri, "Two Set-based Implementations of Quotients in Type Theory", http://ceur-ws.org/Vol-1525/paper-14.pdf

[15] Michael Alton Warren, "Homotopy Models of Intensional Type Theory", http://mawarren.net/papers/prospectus.pdf, 2006