Master thesis

Blakers-Massey Connectivity Theorem from the perspective of homotopy type theory

Aloïs Rosset

supervised by
Jérôme Scherer

Spring 2019

Contents

Abstract

This document is a Master thesis, written by Aloïs Rosset at the École Polytechnique Fédérale de Lausanne (EPFL), under the guidance of Dr Scherer Jérôme. It explores the field of homotopy type theory with the goal of demonstrating the Blakers-Massey Connectivity Theorem inside this theory.

Chapter 1

Introduction

Type theory is a branch of mathematics, connected to logic and informatics, which gives an alternative to set theory as a *foundation of mathematics*. Although it is not the most common one, its specificities entail lots of applications, especially in computer sciences and more specifically in the theory of programming languages.

Homotopy type theory, often abbreviated HoTT, is a new field of mathematics, based on recent discoveries on the intrinsic homotopical content that lies inside type theory. From this union between homotopy theory and type theory and the refreshed point of view that it proposes, new ideas for the progress of mathematics have arisen. The *Univalence Foundation* program aims to make the most out of this new approach.

The basic concept of type theory is to have a *term* a of a *type* A, which is denoted $a : A$. It is analogous to the set theoretic affirmation "a is an element of the set A", or even more to the homotopic idea that "a is a point of the space A".

Similarly to set theory, recreating a new foundation of mathematics from scratch requires precautions. A rigorous context of how to create types, what is allowed to be performed and how to manipulate objects will be presented. Type theory differs from set theory, as its ground is not an axiomatization. Because of its origin as a theory of computation, it needs no axioms but consists instead of rules, which are more "procedural". With those rules, each new type or technique to create new types is treated as a primitive concept.

Key element of homotopy type theory and first real axiom encountered, Voevodsky's *univalence axiom* (see Section 2.2) opens a lot of doors and possibilities when combined with further notions, e.g., such as *higher inductive types* (see Chapter 3).

It is important to note that everything in type theory can be written in a purely formalised deductive system. It can for instance be seen as being built on top of the mode of computation that is lambda calculus, or differently with a formalisation in deductive trees similarly to sequent calculus in logic. Thanks to this aspect, there exists computer implementations of type theory and of univalent foundations, which are based on existing proof assistants such as Coq or Agda. Number of important theorems have been translated in HoTT, coded and computer-checked this way. To name a few, it includes the calculations of some homotopy groups of spheres, the construction of the Hopf fibration, of covering spaces and of Eilenberg-Mac Lane spaces, and the demonstration of the Freudenthal suspension theorem, the van Kampen theorem and the Mayer-Vietoris theorem.

The purpose of the present document is to prove inside HoTT one of the famous result of homotopy theory, namely the Blakers-Massey Connectivity Theorem. The main reason behind the choice of this major theorem in particular and not another one is that it represents well the motivations behind the study of HoTT. First thing to note, the proof has been coded in Agda and computer-checked [3]. Secondly, the technique of the proof is the fruit of a back-and-forth progress between category theorists and type theorists. This progress is quite visible in the reading of *Homotopy Type Theory: Univalent Foundations of Mathematics* [7] - the most comprehensive book on HoTT at the moment - especially in the Chapter 8, which is focused on homotopy theory. The proof of the Freudenthal suspension theorem, which is a special case of the Blakers-Massey theorem, is for instance in this chapter. And finally, the demonstration presented in this document has since been adapted in the context of arbitrary ∞-toposes [5] in a significantly more elementary way that what existed beforehand [6]. Moreover, generalisations of the proof have lead to new homotopy-theoretic results [1]. This last fact is crucial, as the generalisation of Blakers-Massey theorem is still a problem of great interest. Hence the motivation to approach the subject

from another point of view and to translate the proofs in the other setting each time a new progress is made. This way of proceeding looks promising for future advancements in mathematics.

1.1 Basics of type theory

The essence of type theory is that mathematical objects are constructed with a minimalist mindset. Everything is reduced to its core aspect. To illustrate that, constructing a type is done in two steps. It starts by stating what elements are put in the type and it ends by describing how to use them. For instance, the unit type $\mathbf{1}$ is a very basic example. Its purpose is to contain only one element, hence the presence of a constructing rule stating that $\star : \mathbf{1}$. The symbol $\star$ already refers to a singleton in mathematics, just as we wish to do here. Then, the only other rule states that defining a function between types $f : \mathbf{1} \to A$ requires only to specify the image $f(\star)$ of our single element. As announced, there is no other rule that those two in the definition of $\mathbf{1}$; only the last one is in fact formulated in a slightly broader way (see Section 1.4). Having nothing more than those two rules might seem weird, because having $\star : \mathbf{1}$ does not seem to prevent us from having other elements $x : \mathbf{1}$, coming from who knows where. But in fact, they suffice and $\mathbf{1}$ fulfils perfectly the role of the single element type. For instance, the two rules prove that any unknown and mysterious element $x : \mathbf{1}$ we might have, is in fact not so mysterious as it must be equal to $\star$.

To sum it up, types are kind of an abstract way to look at collections of objects, in clear contrast with set theory where the hierarchy of sets is constructed in the most explicit way. A type is therefore often suited for an illustration with a Venn diagram, where only the elements in use are specified.

Definition 1.1. Given a type A, it is said to be *inhabited* when it is known that some term a is of type A, i.e., $a : A$.

A core concept in type theory is the desire to have everything internal to the theory, in opposition again to set theory where there are two "layers", the mathematical objects and the propositions about them. Here, we will see the notion of *propositions as types*, where mathematical statements are expressed through types. Proving a proposition is then done by inhabiting the concerned type. As a small preview of it, consider the idea that two functions, say $f, g : A \to B$ are equals if and only if they are equals point by point, i.e., all their values are equal. This will have the name of *function extensionality*. With the idea of propositions as types, it is stated as

$$(f = g) \simeq \prod_{x:A} (f(x) = g(x)) .$$

If we identify "$\simeq$" with "if and only if" and "$\prod_{(x:A)}$" with "for all x in A", we can read the statement as the desired proposition. An inhabitant of this proposition as type is then a "witness" or "proof" that the proposition holds. This is even more important in homotopy type theory with the role that the equality plays. As used implicitly above, the equality between two elements of a type $a, b : A$ can be expressed through a new type $a =_A b$ (see Section 1.5). The homotopic point of view then offers to see a witness $p : a =_A b$ that the equality holds as a *path* in the space A. This interpretation and analogy fits actually with wonder as we will see starting from Chapter 2.

As another example, consider the following theorem of type theory:

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

If R is read as a relation, the same identifications as before with in addition "$\sum_{(y:B)}$" read as "there exists y in B" gives us a version of the axiom of choice where the existence of elements y satisfying the relations $R(x, y)$ for all x is equivalent to having a choice function f. The provability of the axiom of choice seems absurd, but it comes down to the fact that type theory can be qualified of a "constructive" approach to mathematics. Because of propositions as types, having a proposition P and a proof of it $p : P$ allows us to manipulate this latter, for instance to obtain then proofs of other propositions. These methods are therefore often called "proof-relevant".

In fact here, "$\sum_{(y:B)}$" does not exactly correspond to "there exists y in B", for the reason that it carries extra information about an explicit element y, whereas "there exists y in B" does not tell us anything about y except that it exists (more details in Sections 1.3 and 1.6).

One can naturally wonder about the question of where do all the types live. If we call $\mathcal{U}$ some kind of universe of types, then what is $\mathcal{U}$? Wishing $\mathcal{U}$ to be a type too, we may call $\mathcal{U}$ by $\mathcal{U}_0$ instead and suppose that $\mathcal{U}_0 : \mathcal{U}_1$ where $\mathcal{U}_1$ is a new type universe. But then the same question arises for $\mathcal{U}_1$. With this idea in mind, we introduce a hierarchy of universes

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

This avoids paradoxes, like Russell's in set theory, as one shall refrain from defining an immense "type-universe" containing absolutely all types including itself. From now on, supposing the existence of a type is done by letting $A : \mathcal{U}_i$. The subscript can be dropped when it is not relevant to specify the exact universe. For curiosity's sake, the two rules about universes are that $\mathcal{U}_i : \mathcal{U}_{i+1}$ and that $A : \mathcal{U}_i$ implies $A : \mathcal{U}_{i+1}$, but there will be no need anyway to dive deeper than that in the subject of universe in this document.

One more difference between type theory and set theory lies in the treatment of equality. As said above, given $a, b : A$, a new type $a =_A b$ can be formed (see Section 1.5). Following the notion of propositions as types, the equality between a and b holds exactly when the equality type $a =_A b$ is inhabited, and then a and b are said to be *(propositionally) equal*.

But there is also the need for another equality in type theory. Called the *judgmental equality* or *definitional equality*, it serves to express when two elements are by nature the same fundamental object. It is denoted $a \equiv b : A$, with here $a, b : A$. We can see it as b being simply another letter or name to talk about the exact same object as when we talk about a, and vice-versa. The judgmental equality also works to talk about types. For instance, $a : A \equiv B$ tells us that $a : A$ and equivalently that $a : B$. When introducing a new name for an object, to say that we define it as judgmentally equal to something else we use the symbols "$:\equiv$" or "$\equiv:$".

1.2 Function types and Π-types

Given two types A and B, we wish to have a function type $A \to B$ so that the usual mathematical notation $f : A \to B$ would also make sense in type theory. To already have an understanding of the usual procedure in defining a type former, the steps are

- a *formation rule*, stating when the type former can be applied;
- *introduction rules*, stating how to inhabit the type;
- *elimination rules*, or an induction principle, stating how to use an element of the type;
- *computation rules*, explaining how the eliminations rules react when combined with the introduction rules;
- optionally a *uniquess principle*, stating how every element of the type is uniquely determined by the means of its application within elimination rules.

Definition 1.2. The *function type* from a type $A : \mathcal{U}$ to another $B : \mathcal{U}$ is defined as follows.

- We create $A \to B : \mathcal{U}$ as a type on its own.
- To obtain a term $f : A \to B$, we want to do that either by *direct definition*, with

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

  where Φ is an expression which can include the variable x and which is of type B whenever $x : A$. Or by *λ-abstraction* with

  $$f :\equiv \lambda(x : A).\, \Phi$$

  with Φ as above. When the information is clear, we may omit the type of x and write $\lambda x.\, \Phi$. The notation "λ" comes from λ-calculus, since the latter is one of the ways to approach formally type theory. As alternative notations, we have the more common $x \mapsto \Phi$, or also the use of the symbol "$-$" in place of the variable x inside Φ, such as $g(x, -)$ instead of $\lambda y.\, g(x, y)$.

  The convention for parentheses is that what comes after "$\lambda x.$ " is in its scope unless the delimitations are made clear with parentheses or punctuation.

Note that the variable x in a definition such as $\lambda x.\,\Phi$ is said to be *binded*, or *not occurring free* in it. When that is the case, the *substitution* of a variable is exactly what is expected, which is to replace every occurrence of the variable with something else, except for the binded variables since they only belong and make sense in their respective subexpression.

- To make use of a function $f : A \to B$, we let $f(a) : B$ whenever we have some $a : A$. We say that we *apply* f to a and $f(a)$ is called the *value* of f at a. It is not uncommon to drop the parentheses and to write fa for $f(a)$.
- The computation rule for a function $f :\equiv (\lambda x.\,\Phi) : A \to B$ is that

$$f(a) \equiv (\lambda x.\,\Phi)(a) :\equiv \Phi'$$

  where Φ' is Φ with the substitution of having each free occurrence of x replaced by an a. A notation for Φ' is $\Phi[a/x]$.
- We wish to say that every function $f : A \to B$ is uniquely determined by all its values $f(x)$ at each $x : A$. This is stated through a uniqueness principle

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

An easy example of function is the *identity function*, where given a type $A : \mathcal{U}$, we have $\mathsf{id}_A : A \to A$ defined as $a \mapsto a$. With this function type former, we can now also go further and define for instance functions with multiple variables. A possibility would be $A \times B \to C$ with the use of a Cartesian product (as we will define later). But an alternative is the *currying* $A \to (B \to C)$. The convention is that we have associativity to the right so that we can also write $f : A \to B \to C$ for a term of this type. Given $a : A$, $b : B$, it is also practical to denote $f(a)(b)$ by $f(a, b)$. Such an f can be defined with what was elaborated before for functions with one variable, for instance with

$$f :\equiv \lambda x.\, \lambda y.\, \Phi$$

where Φ is an expression which may contain the variables x, y, and which is of type C whenever $x : A$ and $y : B$.

Definition 1.3. Given a type $A : \mathcal{U}$, a function $B : A \to \mathcal{U}$ is called a *family of types*. Therefore for each $a : A$ is $B(a)$ a type on its own.

With the definition of a family of types, the question arises about functions whose codomain may be dependent of the variable. For instance, given $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, we wish for the existence of a function with its value at any $a : A$ being of type $B(a)$.

Definition 1.4. The *dependent function type* or Π-*type*, given a type $A : \mathcal{U}$ and a family of types $B : A \to \mathcal{U}$ is defined as follows.

- We create $\prod_{(x:A)} B(x) : \mathcal{U}$ as a type on its own. Alternatively, it can also be denoted

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

- An element $f : \prod_{(x:A)} B(x)$ is, as with (non-dependent) function types, constructed either as

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

  or

$$f :\equiv \lambda x.\, \Phi$$

  where Φ is an expression of type $B(x)$ whenever $x : A$.
- Given $f : \prod_{(x:A)} B(x)$, we can apply it to some $a : A$ and obtain

$$f(a) : B(a).$$

- The computation rule is, still in a similar fashion, that for $f :\equiv (\lambda x.\, \Phi) : \prod_{(x:A)} B(x)$,

$$f(a) \equiv (\lambda x.\, \Phi)(a) :\equiv \Phi[a/x].$$

- The uniqueness principle does not change either and for $f : \prod_{(x:A)} B(x)$, we have

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

The convention is that everything after the "Π" symbol is in its scope, unless specified with parenthesis or punctuation. Observe that if the family of types $B : A \to \mathcal{U}$ is constant, i.e. $B(a) \equiv C$ for some type $C : \mathcal{U}$ and for all $a : A$, then we have another characterisation of the (non-dependent) function $A \to C$ as

$$\left(\prod_{x:A} B(x)\right) \equiv \left(\prod_{x:A} C\right) \equiv: A \to C.$$

In fact, this is even the real formal definition, but for understanding reasons, it is a lot more natural to see first function types and then to generalise to Π-types.

Composition of functions in type theory is done as follows.

$$\begin{aligned} &\mathsf{composition} : \prod_{A,B,C:\mathcal{U}} (A \to B) \to (B \to C) \to (A \to C) \\ &\mathsf{composition}(A, B, C, f, g, a) :\equiv g(f(a)), \end{aligned}$$

and we denote $\mathsf{composition}(A, B, C, f, g)$ by $g \circ f$.

1.3 Product types and Σ-types

With the same reasoning as for function types, let us first introduce the (non-dependent) notion of cartesian product and then generalise to dependent pair types.

Definition 1.5. The *product type* of two types $A, B : \mathcal{U}$ is defined as follows.

- We create $A \times B : \mathcal{U}$ as a type on its own.
- Given $a : A$ and $b : B$, we construct pairs

$$(a, b) : A \times B.$$

- For the elimination rule, we want to know how to use the product when facing cases as $A \times B \to C$ or $\prod_{(x:A\times B)} C(x)$. For the non-dependent situation first, suppose $C : \mathcal{U}$. To explicit a function $A \times B \to C$, we want to describe its values on pairs (a, b), since they are the elements that we want to have in $A \times B$ and that we have supposed to exist so far. This can be done by the means of currying as mentioned earlier. So given $g : A \to B \to C$, we let

$$\begin{cases} f : A \times B \to C, \\ f((a, b)) :\equiv g(a)(b). \end{cases}$$

  The fact that all the information needed is inside functions $A \to B \to C$ can be packed into a *recursion principle*

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

  with a defining equation

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

  For instance, projections can be defined either directly

$$\begin{cases} \mathsf{pr}_1 : A \times B \to A, \\ \mathsf{pr}_1((a, b)) :\equiv a, \end{cases} \qquad \begin{cases} \mathsf{pr}_2 : A \times B \to B, \\ \mathsf{pr}_2((a, b)) :\equiv b, \end{cases}$$

or through recursion

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

For the dependent case, suppose that $C : A \times B \to \mathcal{U}$. Similarly, given $g : \prod_{(a:A)} \prod_{(b:B)} C((a,b))$, we can define a dependent function

$$\begin{cases} f : \displaystyle\prod_{x:A\times B} C(x), \\ f((a,b)) :\equiv g(a)(b). \end{cases}$$

This can be packed into an *induction principle*

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

with a defining equation

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

Note that the recursion and induction principles are our eliminators, and that the recursion is a particular case of the induction where $C : A \times B \to \mathcal{U}$ is a constant family of types such as $C :\equiv \lambda x.\, D$. Moreover, their defining equations are the computation rules since they tell us what happens on pairs, which are the elements given by the constructor.

- This definition so far can seem odd since we know we can create pairs and use them, but for now nothing seems to prevent having other kind of elements inside $A \times B$. For that, there is use for an uniqueness principle stating that every element of $A \times B$ is equal to a pair. This current definition is well-made in the sense that we do not need to suppose it as a rule but we can prove it as propositional uniqueness principle, that is with a propositional equality. In other words, we claim the existence of an element

$$\mathsf{uniq}_{A\times B} : \prod_{x:A\times B} x =_{A\times B} (\mathsf{pr}_1\, x, \mathsf{pr}_2\, x).$$

For that, we need to manipulate an equality type (see Section 1.5), but the only knowledge needed here is that whenever we have a term $z : Z$, there is a witness that the propositional equality is reflexive, $\mathsf{refl}_z : z =_Z z$. Therefore, the induction principle allows us to define

$$\mathsf{uniq}_{A\times B} :\equiv \mathsf{ind}_{A\times B}(\lambda x.\, x =_{A\times B} (\mathsf{pr}_1\, x, \mathsf{pr}_2\, x), \lambda a.\, \lambda b.\, \mathsf{refl}_{(a,b)})$$

as desired.

Similarly as for the passage of function types to Π-types, we now wish to consider a product of types where the second coordinate is in a type which depends on the first coordinate.

Definition 1.6. The *dependent pair type* or Σ*-type* of a type $A : \mathcal{U}$ and a family of types $B : A \to \mathcal{U}$ is defined as follows.

- We create $\sum_{(x:A)} B(x)$ as a type on its own. Alternatively, it can be also denoted

$$\sum_{x:A} B(x).$$

- Given $a : A$ and $b : B(a)$, we construct pairs

$$(a,b) : \sum_{x:A} B(x).$$

- Similarly as for product types, we have first a recursion principle for the non-dependent case, which tells us that all the information needed is in the pairs,

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

with the defining equation

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

For the dependent case, the induction principle becomes

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

with the defining equation

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

As before, the projections can be defined from these principles, except that this time the projection on the second coordinate is a dependent function

$$\mathsf{pr}_1 : \left(\sum_{x:A} B(x)\right) \to A \qquad \text{with } \mathsf{pr}_1 :\equiv \mathsf{rec}_{\sum_{(x:A)} B(x)}(A, \lambda a.\, \lambda b.\, a)$$

$$\mathsf{pr}_2 : \prod_{z:\sum_{(x:A)} B(x)} B(\mathsf{pr}_1\, z) \qquad \text{with } \mathsf{pr}_2 :\equiv \mathsf{ind}_{\sum_{(x:A)} B(x)}(\lambda z.\, B(\mathsf{pr}_1\, z), \lambda a.\, \lambda b.\, b).$$

The notation convention is, as for Π-types, that everything after the "Σ" symbol is in its scope, unless specified with parenthesis or punctuation

1.4 Basic examples

Definition 1.7. The *empty type* **0** is defined as follows.

- We create **0** as a type on its own.
- There is no rule for constructing elements since the purpose is to not have any.
- Given $C : \mathcal{U}$, a function $\mathbf{0} \to C$ always exists. Usually called the empty function, it simply does nothing. But it can still be expressed in a recursion principle

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

From a logic point of view, the empty type **0** corresponds to "False", also denoted ⊥. Therefore, its recursion principle corresponds to the "Principle of explosion" or *ex falso quod libet* stating that anything can be deduced from a contradiction, from the false. There is also an induction principle, which is of little use.

Definition 1.8. The *unit type* **1** is defined as follows.

- We create **1** as a type on its own.
- The constructor states that there is only a single element denoted $\star : \mathbf{1}$.
- Given $C : \mathcal{U}$, a function $\mathbf{1} \to C$ consists simply in specifying a single element of C. The recursion principle translates that as

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

with the defining equation

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

The induction principle generalises that as

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

with the defining equation

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

- We have a propositional uniqueness principle

$$\mathsf{uniq}_{\mathbf{1}} : \prod_{x:\mathbf{1}} (x = \star)$$

defined as

$$\mathsf{uniq}_{\mathbf{1}} :\equiv \mathsf{ind}_{\mathbf{1}}(\lambda x.\, x = \star, \mathsf{refl}_{\star}).$$

Definition 1.9. The *two element type* or *Boolean type* $\mathbf{2}$ is defined as follows.

- We create $\mathbf{2}$ as a type on its own.
- The constructor states that there are two elements,
  - $0_{\mathbf{2}} : \mathbf{2}$ and
  - $1_{\mathbf{2}} : \mathbf{2}$.
- Given $C : \mathcal{U}$, a function $\mathbf{2} \to C$ consists simply in specifying two elements of C. The recursion principle translates that as

$$\mathsf{rec}_{\mathbf{2}} : \prod_{C:\mathcal{U}} C \to C \to \mathbf{2} \to C$$

with the defining equations

$$\mathsf{rec}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0,$$
$$\mathsf{rec}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1.$$

The induction principle generalises that,

$$\mathsf{ind}_{\mathbf{2}} : \prod_{C:\mathbf{2}\to\mathcal{U}} C(0_{\mathbf{2}}) \to C(1_{\mathbf{2}}) \to \prod_{x:\mathbf{2}} C(x)$$

with the defining equation

$$\mathsf{ind}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) :\equiv c_0,$$
$$\mathsf{ind}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) :\equiv c_1.$$

Definition 1.10. The type of *natural numbers* is defined as follows

- We create $\mathbb{N} : \mathcal{U}$ as a type on its own.
- We give two ways to construct an element of $\mathbb{N}$, either it is 0 or a successor of another natural number,
  - $0 : \mathbb{N}$,
  - $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.
- Given $C : \mathcal{U}$, the word recursion corresponds here to the way it is usually employed in mathematics when talking about natural numbers, in the sense that a function $f : \mathbb{N} \to C$ can be obtained from two data, $c_0 : C$ and $c_s : \mathbb{N} \to C \to C$ by having

$$f(0) :\equiv c_0,$$
$$f(\mathsf{succ}(n)) :\equiv c_s(n, f(n)).$$

The recursion principle is

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

with the defining equations

$$\begin{aligned} &\mathsf{rec}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0, \\ &\mathsf{rec}_{\mathbb{N}}(C, c_0, c_s, \mathsf{succ}(n)) :\equiv c_s(n, \mathsf{rec}_{\mathbb{N}}(C, c_0, c_s, n)). \end{aligned}$$

This generalises to an induction principle

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

with the defining equations

$$\begin{aligned} &\mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0, \\ &\mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, \mathsf{succ}(n)) :\equiv c_s(n, \mathsf{ind}_{\mathbb{N}}(C, c_0, c_s, n)). \end{aligned}$$

For instance, suppose we want to define a function $\mathsf{summation} : \mathbb{N} \to \mathbb{N}$ which informally sends "n" to "$0 + 1 + \ldots n$". For that, we first need addition, $\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Its recursive definition is

$$\begin{aligned} \mathsf{add}(0, m) &:\equiv m, \\ \mathsf{add}(\mathsf{succ}(n), m) &:\equiv \mathsf{succ}(\mathsf{add}(n, m)). \end{aligned}$$

In other words, with the notation as before we have here $C :\equiv \mathbb{N} \to \mathbb{N}$ and

$$\begin{aligned} c_0 &: \mathbb{N} \to \mathbb{N}, & c_0(m) &:\equiv m, \\ c_s &: \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}), & c_s(n, g, m) &:\equiv \mathsf{succ}(g(m)), \end{aligned}$$

where "g" will then correspond to "$\mathsf{add}(n, -)$". All this information packed together gives

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

Now, $\mathsf{summation} : \mathbb{N} \to \mathbb{N}$ is recursively defined as

$$\begin{aligned} \mathsf{summation}(0) &:\equiv 0, \\ \mathsf{summation}(\mathsf{succ}(n)) &:\equiv \mathsf{add}(\mathsf{summation}(n), \mathsf{succ}(n)), \end{aligned}$$

or in other words,

$$\mathsf{summation} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, 0, \lambda n.\, \lambda m.\, \mathsf{add}(m, \mathsf{succ}(n))).$$

1.5 The equality type

Definition 1.11. Given a type $A : \mathcal{U}$ and elements $a, b : A$, the *identity type* is defined as follows.

- We create $a =_A b : \mathcal{U}$ as a type on its own. One can look at it as a family

  $$(-) =_A (-) : A \to A \to \mathcal{U}.$$

  When $a =_A b$ is inhabited, a and b are said to be *(propositionally) equal*. The subscript specifying the type in which the equality occurs is sometimes omitted, as it is usually clear from the context.

- We construct a witness that the equality is reflexive,

  $$\mathsf{refl}_a : a =_A a.$$

- The induction principle, also called *path induction* because of the homotopic point of view - soon to be developed - is the following. Imagine we have an element $p : a =_A b$ and we want to prove a property $C(a, b, p)$ about it (expressed as a type). Then, we only need to prove it for the elements given by the constructors of the equality type, the constants paths $\mathsf{refl}_x : x =_A x$ for $x : A$ arbitrary. In other words, we need to inhabit $C(x, x, \mathsf{refl}_x)$. In summary, we prove $C(a, b, p)$ by changing a and b for an arbitrary variable x and by extending from the case $\mathsf{refl}_x : x =_A x$. Now that we have the idea, here is the mathematical statement. Given a family

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

and a function

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

there is a function

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

such that

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

It may sometimes be practical to have the path induction condensed into only one type, which is done by having

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

with the defining equation

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

- An alternative approach for the induction principle is the *based path induction* and goes as follows. In order to prove a proposition $C(a, b, p)$ as before, it offers to replace b for a (or vice-versa), reducing the proof to the reflexivity case on one endpoint of the path $p : a =_A b$ that we have. Given $a : A$, a family

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

and a function

$$c : C(a, \mathsf{refl}_a),$$

there is a function

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

such that

$$f(a, \mathsf{refl}_a) :\equiv c.$$

Condensed, it becomes

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

with the defining equation

$$\mathsf{ind}'_{=_A}(a, C, c, a, \mathsf{refl}_a) :\equiv c.$$

We have seen two ways of having the induction on the equality type, because sometimes one is more practical than the other. But it does not matter which one we suppose to hold since it turns out that these two induction principle are in fact equivalent, as shown in the next lemma.

Remark. This equality type seems a bit mysterious with only one constructor and an induction principle telling us how to extend proofs from the relatively simple reflexivity case. For the natural numbers $\mathbb{N}$, the constructors and the induction principle are both speaking about how an arbitrary natural number is either 0 or a successor number, with in particular the idea that this arbitrary number can be obtained by repeatedly invoking the constructors. But then, does it mean here that every equality type is in fact reflexive? This assertion is wrong as there exists various examples of a type X and path $p : x =_X x$ not equal to refl_x. This confusion arises when forgetting that it is the whole family $(-) =_A (-) : A \to A \to \mathcal{U}$ that is being defined. Therefore, in the same way that $\mathbb{N}$ is a structure "*freely generated*" by its constructors 0 and succ, the equality family $(-) =_A (-)$ should be looked at as freely generated by the elements $\mathsf{refl}_x : x =_A x$. The notion of free generation is the same as in group theory where the free group on two generators x and y contains any elements obtainable from x and y through the operations of the ambient structure, which are the group multiplication xy and element inversion x^{-1}. The operations with the equality type are going to be explored later, but one shall just retain from this interesting question that our definition still makes sense and that there are more complex notions validating the current reasoning (see for instance the notion of *homotopy-initial algebra* in [7]).

Lemma 1.12. *For the equality type, having the path induction or the based path induction gives us also the other one for free.*

Proof. Suppose first that the based path induction holds. Given a type A,

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

and an element $a : A$, we define

$$\begin{cases} D : \prod_{y:A} (x =_A y) \to \mathcal{U}, \\ D :\equiv C(x), \end{cases} \qquad \begin{cases} d : D(x, \mathsf{refl}_x) \equiv C(x, x, \mathsf{refl}_x), \\ d :\equiv c(x). \end{cases}$$

The based point induction implies the existence of

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

such that

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

Thus the conclusion of path induction also holds with

$$\begin{cases} f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p), \\ f(x) :\equiv g, \end{cases}$$

since

$$f(x, x, \mathsf{refl}_x) \equiv g(x, \mathsf{refl}_x) \equiv d \equiv c(x).$$

The first direction was easy since we just had to lock one endpoint of our path to apply the based point induction. Conversely, if we suppose that the path induction holds, we need to cover all cases of based path induction into one path induction. Suppose given $x : A$,

$$D' : \prod_{z:A} (x =_A z) \to \mathcal{U}, \quad \text{and}$$
$$d' : D'(x, \mathsf{refl}_x).$$

We define

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

and

$$\begin{cases} c' : \left(\prod_{x:A} C'(x,x,\mathsf{refl}_x)\right) \equiv \left(\prod_{(x:A)} \prod_{(D':\prod_{(z:A)}(x=_A z)\to\mathcal{U})} D'(x,\mathsf{refl}_x) \to D'(x,\mathsf{refl}_x)\right), \\ c'(x,D',d') :\equiv d'. \end{cases}$$

The path induction applies and gives

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

such that

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

Thus the conclusion of the based path induction also holds with

$$\begin{cases} g' : \prod_{(z:A)} \prod_{(p:x=_A z)} D'(z,p), \\ g'(z,p) :\equiv f'(x,z,p,D',d'), \end{cases}$$

since

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

□

1.6 Propositions as types

Now that the very basics of type theory have been established, let us come back to the notion of propositions as types. The core idea is that a type $A : \mathcal{U}$ can correspond to a proposition. We will see the explicit definition in Chapter 3, but the exact types that are in fact manipulated as propositions are the one which have only two states: either true or false, corresponding to either inhabited or not. The examples seen so far were for instance the propositional uniqueness principle,

$$\mathsf{uniq}_{A\times B} : \prod_{x:A\times B} x = (\mathsf{pr}_1\, x, \mathsf{pr}_2\, x),$$

and

$$\mathsf{uniq}_{\mathbf{1}} : \prod_{x:\mathbf{1}} x = \star.$$

Recall that each of these two proofs was obtained by manipulating a proof of a reflexive equality which was used in the respective induction principle. This aspect of manipulating and transforming proofs is the cornerstone of propositions as types.

Consider for instance two propositions $A, B : \mathcal{U}$, and observe that the new type $A \to B$ is nothing else than "If A then B". Indeed, suppose $A \to B$ holds, i.e., there is a function $f : A \to B$. Now, if A holds, so there is a proof $p : A$, then $f(p) : B$ also holds.

Similarly, given a proposition $A : \mathcal{U}$ and a family of proposition $B : A \to \mathcal{U}$ which is called a *predicate*, note that $\prod_{(x:A)} B(x)$ means "for all $x : A$, we have $B(x)$". Indeed, suppose it holds, i.e., there is $f : \prod_{(x:A)} B(x)$. Now, for all $x : A$ we have that $B(x)$ holds as it is inhabited by $f(x)$.

To avoid over-explaining every one of these correspondence, let us give a list of how logic is recreated in type theory. For the sake of being exhaustive, here is the full list; the new symbols will be discussed just below.

Notation 1.13. Given $A, B : \mathcal{U}$ which denote propositions, or families thereof, (see Definition 3.7 for

the formal definition), we set

$$\begin{aligned}
\mathsf{False} &:\equiv \mathbf{0},\\
\mathsf{True} &:\equiv \mathbf{1},\\
A \Rightarrow B &:\equiv A \to B,\\
A \wedge B &:\equiv A \times B,\\
A \vee B &:\equiv \|A + B\|,\\
\neg A &:\equiv A \to \mathbf{0},\\
\forall x : A\ (B(x)) &:\equiv \prod_{x:A} B(x),\\
\exists x : A\ (B(x)) &:\equiv \left\| \sum_{x:A} B(x) \right\|.
\end{aligned}$$

These notations allow us to perform logic inside our theory. For instance, two propositions A, B are said to be *logically equivalent*, denoted $A \Leftrightarrow B$, if we have $(A \Rightarrow B) \wedge (B \Rightarrow A)$. We will see a bit later that it can also be denoted $A \simeq B$ (thanks to the notion of equivalence from Definition 2.24) or even $A = B$ (thanks to the Univalence axiom p. 25).

First observation, the negation of a proposition, $\neg A$, is done by performing a *reductio ad absurdum* $A \to \mathbf{0}$, i.e., a proof of $\neg A$ is in fact witnessing that A cannot hold without implying the false $\mathbf{0}$.

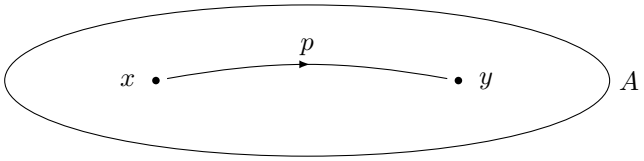
The "+" symbol is the disjoint union for types, which we have not defined as we will not need it in this document.

Then, there is the "$\|-\|$" symbol, called the truncation and which will be defined in detail in Chapter 3. It is used for the logical "or" and "there exists". For the latter, the glance given in the introduction can be developed. The observation is that given $A : U$ and $B : A \to \mathcal{U}$, then "$\sum_{(x:A)} B(x)$" is a stronger version of "$\exists x : A\ (B(x))$". This is because a proof $p : \sum_{(x:A)} B(x)$ is in fact equal, by uniqueness, to a pair (a, b) which witnesses that there exists an element $x : A$ such that $B(x)$ holds **and** that this element is explicitly $a : A$ with the proof of $B(a)$ being b. It carries extra information and therefore, there is the need for a truncation which cuts this down and reduces the type to a simpler version of itself, which contains only a binary data, either true or false.

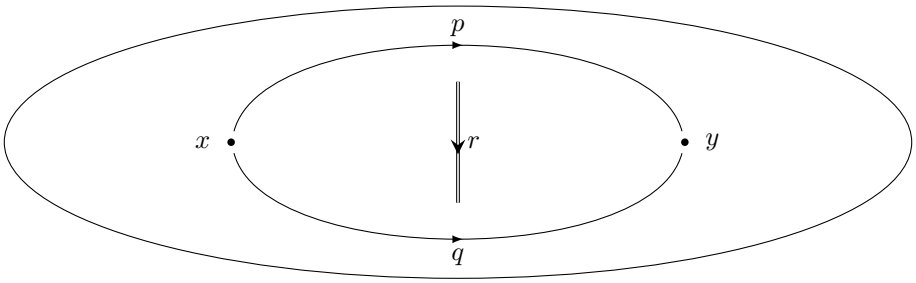
Chapter 2

Homotopy Type theory

Let us switch now to a homotopic point of view. The core idea is to look at a witness of equality $p : x =_A y$ as a *path* in the space A.



Then, if $p, q : x =_A y$ are two paths, witnessing their equality $r : p =_{x=_A y} q$ comes down to a *homotopy of paths* or 2*-path*.



With two homotopies $r, s : p =_{x=_A y} q$, we can then look at $r = s$ to consider *homotopies between homotopies* or 3*-paths* and so on, going up in the dimensions with 4*-paths*, 5*-paths*, etc. It recalls the context of ∞-groupoids from category theory.

In this approach, one quickly wonders about the operations that one desires to have about paths. Observe first that the constructor of the equality type states that the equality is reflexive by having a constant path $\mathsf{refl}_x : x =_A x$ for any $x : A$. As expected, the equality is also symmetric, which corresponds to path inversion.

Lemma 2.1. *Given a type A and $x, y : A$, there is a function*

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

denoted $p \mapsto p^{-1}$, such that $\mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x$ for any $x : A$.

Definition 2.2. Given $p : x = y$, then $p^{-1} : y = x$ is called the *inverse* of p.

With the idea of propositions as types, one shall first translate this lemma into a type. Thus, the aim is to inhabit

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

It comes down to path induction then. Let us first prove it formally with notations explicitly written, and then give a quicker proof with the same argument but in a more natural language which goes straight to the idea.

First proof. Take $A : \mathcal{U}$ and let

$$\begin{cases} C : \prod\limits_{x,y:A} (x = y) \to \mathcal{U}, \\ C(x, y, p) :\equiv (y = x), \end{cases} \qquad \begin{cases} c : \prod\limits_{x:A} C(x, x, \mathsf{refl}_x), \\ c(x) :\equiv \mathsf{refl}_x. \end{cases}$$

By path induction, there is a function

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

such that

$$\mathsf{refl}_x{}^{-1} :\equiv f(x, x, \mathsf{refl}_x) \equiv c(x) \equiv \mathsf{refl}_x.$$

□

Second proof. Take $A : \mathcal{U}$, $x, y : A$ and $p : x = y$. To define $p^{-1} : y = x$, we perform a path induction on $p : x = y$ so it suffices to consider when $y \equiv x$ and $p \equiv \mathsf{refl}_x$. But then, we can define $p^{-1} : x = x$ simply as refl_x. □

Another property needed is the transitivity of the equality, which corresponds to path concatenation.

Lemma 2.3. *Given a type A and $x, y, z : A$, there is a function*

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

denoted $p \mapsto q \mapsto p \cdot q$, such that $\mathsf{refl}_x \cdot \mathsf{refl}_x :\equiv \mathsf{refl}_x$ for any $x : A$.

Definition 2.4. Given $p : x = y$ and $q : y = z$, then $p \cdot q : x = z$ is called the *concatenation* of p and q.

First proof. The goal is to inhabit

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

Take $A : \mathcal{U}$ and let

$$\begin{cases} C : \prod\limits_{x,y:A} (x = y) \to \mathcal{U}, \\ C(x, y, p) :\equiv \prod\limits_{z:A} (y = z) \to (x = z). \end{cases}$$

To perform a path induction, one need

$$c : \left(\prod_{x:A} C(x, x, \mathsf{refl}_x) \right) \equiv \left(\prod_{(x:A)} \prod_{(z:A)} (x = z) \to (x = z) \right).$$

For that, one can make use of another path induction with

$$\begin{cases} D : \prod\limits_{x,z:A} (x = z) \to \mathcal{U}, \\ D(x, z, q) :\equiv (x = z), \end{cases} \qquad \begin{cases} d : \left(\prod\limits_{x:A} D(x, x, \mathsf{refl}_x) \right) \equiv \left(\prod\limits_{x} (x = x) \right), \\ d(x) :\equiv \mathsf{refl}_x, \end{cases}$$

which gives us

$$c : \prod_{x,z:A} (x = z) \to (x = z)$$

such that

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

Therefore, by the first path induction we also have

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

which allows us to define

$$g : \prod_{x,y,z:A} (x = y) \to (y = z) \to (x = z),$$
$$g(x, y, z, p, q) :\equiv f(x, y, p, z, q)$$

as desired. □

Second proof. Take $A : \mathcal{U}$, $x, y, z : A$, $p : x = y$ and $q : y = z$. To define, $p \centerdot q : x = z$, we perform a path induction on p so that $y \equiv x$ and $p \equiv \mathsf{refl}_x$. We then perform another path induction, but on q, so that $z \equiv x$ and $q \equiv \mathsf{refl}_x$. We can now define $p \centerdot q : x = x$ as refl_x. □

Remark. Observe that in the proof above, one made voluntarily a second path induction whereas it was possible to do as follows:

$$c : \prod_{(x:A)} \prod_{(z:A)} (x = z) \to (x = z),$$
$$c(x, z, q) :\equiv q.$$

Doing so gives at the end a valid definition of the concatenation $p \centerdot q$, but a different one nonetheless. It would be propositionally equal to the concatenation we obtained but not judgmentally equal. This observation puts emphasis about the constructivism of the theory, where the choice of one proof-inhabitant over another might be important.

In practice, transitivity of equalities as with $p : x = y$ and $q : y = z$ will sometimes be denoted in the familiar way

$$\begin{aligned} x &= y && \text{(by p)} \\ &= z. && \text{(by q)} \end{aligned}$$

Now that we have our operations to perform inversion and concatenation, it is worth investigating about the properties they carry, which turns out to be the ones that could be expected.

Lemma 2.5. *Given a type A, $x, y, z, w : A$, $p : x = y$, $q : y = z$ and $r : z = w$, the following hold.*

i) $p = p \centerdot \mathsf{refl}_y$, *and* $\mathsf{refl}_x \centerdot p = p$.

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

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

iv) $(p \centerdot q)^{-1} = q^{-1} \centerdot p^{-1}$.

v) $p \centerdot (q \centerdot r) = (p \centerdot q) \centerdot r$.

For later use, let us give names to some of those properties, say $\mathsf{left\_refl}(p) : p = \mathsf{refl}_x \centerdot p$, $\mathsf{right\_refl}(p) : p = p \centerdot \mathsf{refl}_y$, $\mathsf{linv}(p) : p^{-1} \centerdot p = \mathsf{refl}_y$ *and* $\mathsf{rinv}(p) : p \centerdot p^{-1} = \mathsf{refl}_x$.

Proof. i) Just for this first property, let us do both the detailed and the concise proof of $p = p \centerdot \mathsf{refl}_y$. Demonstrating $\mathsf{refl}_x \centerdot p = p$ is then similar.

First proof. The aim is to prove the proposition-family

$$\begin{cases} C : \prod_{x,y:A} (x = y) \to \mathcal{U}, \\ C(x, y, p) :\equiv (p = p \centerdot \mathsf{refl}_y). \end{cases}$$

Let

$$\begin{cases} c : \left(\prod_{x:A} C(x,x,\mathsf{refl}_x)\right) \equiv \left(\prod_x \mathsf{refl}_x = \mathsf{refl}_x \cdot \mathsf{refl}_x\right) \equiv \left(\prod_{x:A} \mathsf{refl}_x = \mathsf{refl}_x\right), \\ c(x) :\equiv \mathsf{refl}_{\mathsf{refl}_x}. \end{cases}$$

By path induction, we have

$$\mathsf{ind}_{=_A}(C,c,x,y,p) : p = p \cdot \mathsf{refl}_y.$$

□

Second proof. By path induction on p, it suffices to consider when $y \equiv x$ and $p \equiv \mathsf{refl}_x$. But then, we already have the judgmental equality $\mathsf{refl}_x \equiv \mathsf{refl}_x \cdot \mathsf{refl}_x$. □

ii) By path induction on p, it suffices to consider when $y \equiv x$ and $p \equiv \mathsf{refl}_x$. But then,

$$\mathsf{refl}_x{}^{-1} \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x, \text{ and}$$

$$\mathsf{refl}_x \cdot \mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x.$$

iii) By path induction on p, it suffices to consider when $y \equiv x$ and $p \equiv \mathsf{refl}_x$. But then,

$$(\mathsf{refl}_x{}^{-1})^{-1} \equiv \mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x.$$

iv) By path induction on p and then on q, it suffices to consider when $y \equiv x$, $p \equiv \mathsf{refl}_x$, $z \equiv x$ and $q \equiv \mathsf{refl}_x$. But then,

$$(\mathsf{refl}_x \cdot \mathsf{refl}_x)^{-1} \equiv \mathsf{refl}_x \equiv \mathsf{refl}_x{}^{-1} \cdot \mathsf{refl}_x{}^{-1}.$$

v) By doing three consecutive path inductions, it suffices to consider when $y \equiv x$ and $p \equiv \mathsf{refl}_x$, $z \equiv x$ and $q \equiv \mathsf{refl}_x$, and $w \equiv x$ and $r \equiv \mathsf{refl}_x$. But then,

$$\mathsf{refl}_x \cdot (\mathsf{refl}_x \cdot \mathsf{refl}_x) \equiv \mathsf{refl}_x \equiv (\mathsf{refl}_x \cdot \mathsf{refl}_x) \cdot \mathsf{refl}_x.$$

□

2.1 Functions are functors and families of types are fibrations

Here is another common property that is expected about the equality. When there are $x, y : A$ being equal, $p : x = y$, and a function $f : A \to B$, it is expected to have $f(x) = f(y)$. From a category theory perspective, it means that the function f behave functorially with the presence of an element "$f(p)$" : $f(x) = f(y)$. While topologically speaking, the function f appears to be continuous at least in the idea that it preserves paths.

Lemma 2.6. *Given a function $f : A \to B$, then for all $x, y : A$ there is a function*

$$\mathsf{ap}_f : (x = y) \to (f(x) = f(y))$$

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

First proof. We want to prove

$$\begin{cases} C : \prod_{x,y:A} (x = y) \to \mathcal{U}, \\ C(x,y,p) :\equiv (f(x) = f(y)). \end{cases}$$

For that, let

$$\begin{cases} c : \left(\prod_{x:A} C(x,x,\mathsf{refl}_x)\right) \equiv \left(\prod_{x:A} f(x) = f(x)\right), \\ c(x) :\equiv \mathsf{refl}_{f(x)}. \end{cases}$$

Then, by path induction we have

$$\mathsf{ap}_f(p) :\equiv \mathsf{ind}_{=_A}(C, c, x, y, p) : (f(x) = f(y))$$

such that

$$\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{ind}_{=_A}(C, c, x, x, \mathsf{refl}_x) \equiv c(x) \equiv \mathsf{refl}_{f(x)}.$$

□

Second proof. By path induction on $p : x = y$, it suffices to consider when $y \equiv x$ and $p \equiv \mathsf{refl}_x$. But then, we can define

$$\mathsf{ap}_f(\mathsf{refl}_x) :\equiv \mathsf{refl}_{f(x)} : f(x) = f(x).$$

□

Notation 2.7. In the same context as in the last lemma, we sometimes write $f(p)$ for $\mathsf{ap}_f(p)$, which highlights the functorial aspect.

The following properties also denote the functorial behaviour.

Lemma 2.8. *Given functions $f : A \to B$, $g : B \to C$, and paths $p : x =_A y$ and $q : y =_A z$, the following hold.*

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

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

Proof. i) By induction on p and q, we have

$$\mathsf{ap}_f(\mathsf{refl}_x \cdot \mathsf{refl}_x) \equiv \mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)} \equiv \mathsf{refl}_{f(x)} \cdot \mathsf{refl}_{f(x)} \equiv \mathsf{ap}_f(\mathsf{refl}_x) \cdot \mathsf{ap}_f(\mathsf{refl}_x).$$

ii) By induction on p, we have

$$\mathsf{ap}_f(\mathsf{refl}_x{}^{-1}) \equiv \mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)} \equiv \mathsf{refl}_{f(x)}{}^{-1} \equiv \mathsf{ap}_f(\mathsf{refl}_x)^{-1}.$$

iii) By induction on p, we have

$$\mathsf{ap}_g \mathsf{ap}_f \mathsf{refl}_x \equiv \mathsf{ap}_g \mathsf{refl}_{fx} \equiv \mathsf{refl}_{gfx} \equiv \mathsf{ap}_{g \circ f} \mathsf{refl}_x.$$

iv) By induction on p, we have

$$\mathsf{ap}_{\mathsf{id}_A} \mathsf{refl}_x \equiv \mathsf{refl}_{\mathsf{id}_A(x)} \equiv \mathsf{refl}_x.$$

□

We have seen how the equality behaves with functions. But to generalise to dependent function, we need to see how equality and type families mix together.

Lemma 2.9 (Transport lemma)**.** *Given a type A, a family $P : A \to \mathcal{U}$, $x, y : A$ and $p : x = y$, there is a function*

$$p_* : P(x) \to P(y)$$

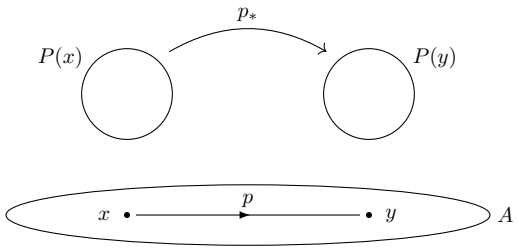
so that $(\mathsf{refl}_x)_* :\equiv \mathsf{id}_{P(x)}$.

Proof. By induction on p, it suffices to consider when $p \equiv \mathsf{refl}_x$. But then, we define $(\mathsf{refl}_x)_* :\equiv \mathsf{id}_{P(x)}$. □

Notation 2.10. In the same context as in the last lemma, we sometimes also denote p_* by

$$\mathsf{transport}^P(p, -) : P(x) \to P(y),$$

in order to keep track of the family P.



Continuing the parallel between type theory and homotopy theory, we can look at a family $P : A \to \mathcal{U}$ as inducing a *fibration*. The *base space* is A and the *fiber* over an element $x : A$ is $P(x)$. The *total space* is then made of all those fibers, thus it consists here of $\sum_{(x:A)} P(x)$ and the explicit fibration is therefore

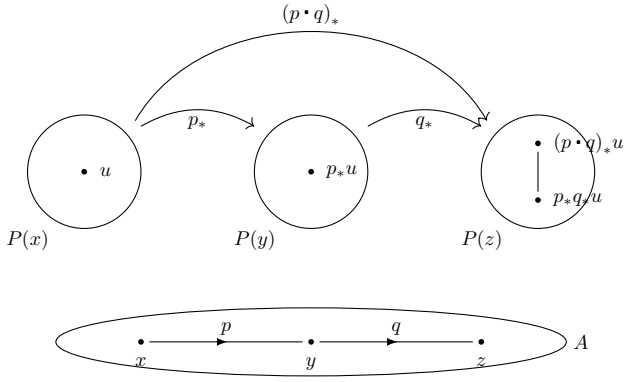
$$\mathsf{pr}_1 : \left(\sum_{x:A} P(x)\right) \to A.$$

The transport function defined above gives us a way to have functions between fibers. This vision of families of types as inducing a fibration will be confirmed with the different properties that we will prove. For instance, we will see that when two elements of a space $x, y : A$ are in the same path component, i.e. we have $x = y$, then their fibers $P(x)$ and $P(y)$ are homotopy equivalent. Note that we have not defined a notion corresponding to homotopy equivalence in HoTT yet, but that will come in the next section.

To better visualise these fibrations, here is a first easy property.

Lemma 2.11. *Given a type A, a family $P : A \to \mathcal{U}$ and two paths $p : x =_A y$, $q : y =_A z$, then for all $u : P(x)$ we have*

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



Proof. By induction on q and then on p, we indeed have

$$(\mathsf{refl}_x)_*(\mathsf{refl}_x)_*(u) \equiv u \equiv (\mathsf{refl}_x \cdot \mathsf{refl}_x)_*(u).$$

□

With this transport function, we can now generalise the functoriality of functions to the dependent cases. The problem is that given $f : \prod_{(x:A)} P(x)$ and $p : x =_A y$, then $f(x) : P(x)$ and $f(y) : P(y)$ are in different fibers. However, we still want to have a path between them, with the desire of having it "lying above" p in our fibration. But thanks to the last lemma, the functions p_* and $(p^{-1})_*$ allow us to go back and forth between our fibers. Therefore, in order to have a link between $f(x)$ and $f(y)$, we can compare $p_*(f(x))$ and $f(y)$ (or equivalently we could have chosen to compare $f(x)$ and ${p^{-1}}_*(f(y))$).

Lemma 2.12. *Given $P : A \to \mathcal{U}$ and $f : \prod_{(x:A)} P(x)$, then for all $x, y : A$ there is a function*

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

such that $\mathsf{apd}_f(\mathsf{refl}_x) :\equiv \mathsf{refl}_{f(x)}$.

Proof. By induction on $p : x = y$, we can define

$$\mathsf{apd}_f(\mathsf{refl}_x) :\equiv \mathsf{refl}_{f(x)}.$$

□

In the case where we have a constant family $P :\equiv x \mapsto B : A \to \mathcal{U}$, then B is our one and only fiber. Thus, we expect transport to behave like the identity.

Lemma 2.13. *Given $P :\equiv (x \mapsto B) : A \to \mathcal{U}$, then for all $p : x =_A y$ and $b : B$ we have*

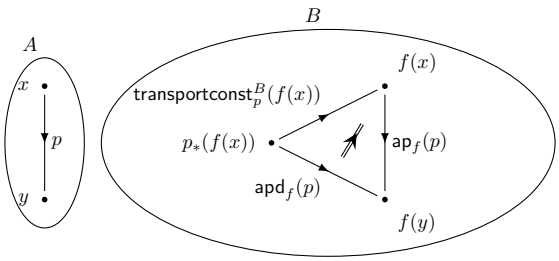
$$\mathsf{transportconst}^B_p(b) : p_*(b) = b.$$

Proof. By induction on p, it suffices to prove $(\mathsf{refl}_x)_*(b) = b$. But both sides are judgmentally equal to b, therefore we have refl_b. □

Note that names were given to the proofs of equality from the last lemma. Naming them help to describe the links between our different paths-proofs. For instance, the next lemma gives us a computational formula between ap_f and apd_f, thanks to transportconst. From a topological point of view, it means that there is an homotopy of paths. But one should not think too much with this point of view here. The only deep meaning that it carries is that we have two formulas for a path between $p_*(f(x))$ and $f(y)$ - one with apd_f, the other with transportconst and ap_f - and that they do not create any new homotopic data between them. This is an idea that we will meet again later, that all of our different type-theoretic formula do behave well together, as they admit practical computational formula and do not mess with the homotopy information of our type.

Lemma 2.14. *Given $f : A \to B$ and $p : x =_A y$, then*

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



Proof. By induction on p, we have

$$\begin{aligned} \mathsf{apd}_f(\mathsf{refl}_x) &\equiv \mathsf{refl}_{f(x)}, \\ \mathsf{transportconst}^B_{\mathsf{refl}_x}(f(x)) &\equiv \mathsf{refl}_{f(x)}, \\ \mathsf{ap}_f(\mathsf{refl}_x) &\equiv \mathsf{refl}_{f(x)}. \end{aligned}$$

Thus, the equality follows from a basic path property (Lemma 2.5). □

2.2 Homotopies, equivalence, function existentionality and univalence

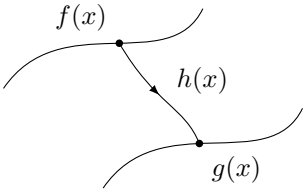
We have seen that taking iterated identity types gives us homotopy of paths or 2-paths, and homotopy of homotopy of paths or 3-paths, etc. But there is still a need for a notion of homotopy between functions and of homotopy equivalence in HoTT.

Definition 2.15. Let $f, g : \prod_{(x:A)} B(x)$ be two (possibly dependent) functions where $B : A \to \mathcal{U}$ is a family. The type of *homotopies* from f to g is defined as

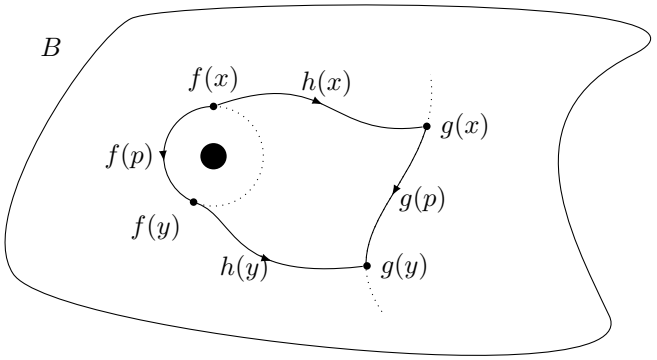
$$f \sim g :\equiv \prod_{x:A} (f(x) =_{B(x)} g(x)).$$

We say that f and g are *homotopic* whenever this type is inhabited.

Looking at those equalities $f(x) = g(x)$ as paths, the usual notion of homotopies is indeed there. A homotopy $h : f \sim g$ can be illustrated as follows.



For $x : A$, the path $h(x) : f(x) = g(x)$ is a "continuous path", topologically speaking. In topology, when we have a homotopy $H : X \times I \to Y$ between two functions $F, G : X \to Y$, i.e., $H(-, 0) = F$ and $H(-, 1) = G$, we have indeed that each $H(x, -)$ is a continuous path from $F(x)$ to $G(x)$. But this derives from the continuity of H itself. So now back to HoTT, is our homotopy h continuous ? Of course, such a question is purely rhetorical without a notion of continuity. But here is a wonderful rule of thumb about HoTT: **everything is always continuous, natural, functorial, etc.** For continuity, this is visible through the properties, which can all be verified. For instance, we had the continuity of functions appearing in Lemma 2.6 in the fact that they preserve paths. For our homotopy h, consider the informal example where $f, g : A \to B$ are non dependent for simplicity, B has a "hole" in it, with f having its image around it but g not. Suppose also that A is path-connected, thus by Lemma 2.6, the images of f and g must too. Then, each $h(x) : f(x) = g(x)$ must choose of which side of the hole it passes, but they cannot all pick the same. Therefore, there are at least two points $x, y : A$ with $h(x)$ and $h(y)$ on different sides. Here, each $h(x)$ is continuous, but h itself is not continuous because of the hole in its image. To see that better, let us name a path $p : x = y$ and give an illustration.



Indeed, this scenario **cannot happen** thanks to the next lemma, which says that the paths on the drawing commute, or equivalently that the homotopy is like a natural transformation if we look at functions as functors.

Lemma 2.16. *Given $f, g : A \to B$, $h : f \sim g$ and $p : x =_A y$, then*

$$h(x) \cdot g(p) = f(p) \cdot h(y).$$

Proof. By induction on p, we have

$$h(x) \cdot g(\mathsf{refl}_x) \equiv h(x) \cdot \mathsf{refl}_{g(x)} = \mathsf{refl}_{f(x)} \cdot h(x) \equiv f(\mathsf{refl}_x) \cdot h(x).$$

□

Corollary 2.17. *Given $f : A \to A$, $h : f \sim \mathsf{id}_A$ and $a : A$, then*

$$h(fx)) = f(hx)).$$

Proof. Observe that

$$\begin{aligned} h(fx)) &= \big(h(fx)) \cdot hx)\big) \cdot (hx)^{-1} \\ &= \big(f(hx)) \cdot hx\big) \cdot (hx)^{-1} && \text{(by Lemma 2.16)} \\ &= f(hx)). \end{aligned}$$

$$\begin{array}{ccc} ffx & \xlongequal{f(hx)} & fx \\ {\scriptstyle h(fx)}\Big\| & & \Big\|{\scriptstyle hx} \\ fx & \xlongequal[hx]{} & x. \end{array}$$

□

From the logical point of view and the equalities with their primary meaning, a homotopy $h : f \sim g$ do tell us that both functions f and g have each of their values $f(x)$ and $g(x)$ being equal. Note that by definition, having $f \sim g$ is not the same as saying that $f = g$. However, we would like both notions to say the same fact, as having two functions being equal or being equal at each value seems to be the same phenomenon. But for that, let us first define precisely a notion of homotopy equivalence in homotopy type theory.

Definition 2.18. Let $f : A \to B$ be a function. The type of *quasi-inverses* of f is defined as

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

An inhabitant of $\mathsf{qinv}f$ is therefore a triplet (g, α, β) where $g : B \to A$ is a function going backwards to f and α, β are homotopies between the compositions of f and g and the identities. It is the literal translation of homotopy equivalence in our framework. However, this definition presents some type-theoretic inconveniences, for a reason that will be detailed later. Because of this, we also employ alternative definitions of the equivalence.

Definition 2.19. Let A, B be types. A function $f : A \to B$ between them is an *bi-invertible map* if the following type is inhabited,

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

In fact, being bi-invertible and admitting a quasi-inverse are synonymous notions, as being in one case implies immediately the other. The proof for this simple fact is even the same in homotopy theory or in homotopy type theory,

Lemma 2.20. *Given $f : A \to B$, the types $\mathsf{qinv}(f)$ and $\mathsf{isbiinv}(f)$ are logically equivalent, i.e., they imply each other:*

$$\begin{aligned} \mathsf{qinv}(f) &\to \mathsf{isbiinv}(f), \ \textit{and} \\ \mathsf{isbiinv}(f) &\to \mathsf{qinv}(f). \end{aligned}$$

Proof. The first function is

$$\begin{cases} \mathsf{qinv}(f) \to \mathsf{isbiinv}(f), \\ (g, \alpha, \beta) \mapsto ((g, \alpha), (g, \beta)). \end{cases}$$

For the second function, $\mathsf{isbiinv}(f) \to \mathsf{qinv}(f)$, take $((g,\alpha),(h,\beta)) : \mathsf{isbiinv}(f)$. We want to keep g and α, but to transform $\beta : h \circ f \sim \mathsf{id}_A$ into another homotopy $\beta' : g \circ f \sim \mathsf{id}_A$. For that, let β' be the composite

$$\begin{aligned} g \circ f &\overset{(\beta \circ g \circ f)^{-1}}{\sim} h \circ f \circ g \circ f \\ &\overset{h \circ \alpha \circ f}{\sim} h \circ g \\ &\overset{\beta}{\sim} \mathsf{id}_A. \end{aligned}$$

□

Here is another alternative definition that we will need at some point.

Definition 2.21. Let A, B be types. A function $f : A \to B$ is a *half adjoint equivalence* if the type $\mathsf{ishae}(f)$ is inhabited, which requires to have a function $g : B \to A$ and homotopies $\eta : g \circ f \sim \mathsf{id}_A$ and $\epsilon : f \circ g \sim \mathsf{id}_B$ such that there exists a homotopy

$$\tau : \prod_{x:A} f(\eta x) = \epsilon(fx).$$

From a category perspective, it recalls the context of an adjunction "$f \dashv g$" especially with η^{-1} and ϵ recalling the unit and counit respectively. The homotopy τ corresponds then to one of the two triangle equalities. We suppose only one, whence the name *half* adjoint equivalence, but the other one can be deduced.

Lemma 2.22. *Let $f : A \to B$, $g : B \to A$, $\eta : g \circ f \sim \mathsf{id}_A$ and $\epsilon : f \circ g \sim \mathsf{id}_B$. Then, the following conditions are logically equivalent:*

- $\prod_{(x:A)} f(\eta x) = \epsilon(fx)$.
- $\prod_{(y:B)} g(\epsilon y) = \eta(gy)$.

The proof will not interest us here, but it can be found in [7] (Lemma 4.2.2). The notions of quasi-inverse and of half-adjoint equivalence here also imply each other.

Lemma 2.23. *Given $f : A \to B$, the types $\mathsf{qinv}(f)$ and $\mathsf{ishae}(f)$ are logically equivalent.*

Proof. One direction is easy, one just has to forget the coherence homotopy that we have as extra structure. I.e., when f is a half adjoint equivalence and we have g, η, ϵ and τ, then $(g, \eta, \epsilon) : \mathsf{qinv}(f)$.

In the other direction, suppose we have $g : B \to A$, $\eta : g \circ f \sim \mathsf{id}_A$ and $\epsilon : f \circ g \sim \mathsf{id}_B$. We want to keep g and η, but we will have to redefine an homotopy ϵ' and a coherence homotopy τ'. We let ϵ' be the composite

$$\begin{aligned} fg &\overset{(\epsilon \circ f \circ g)^{-1}}{\sim} fgfg \\ &\overset{f \circ \eta \circ g}{\sim} fg \\ &\overset{\epsilon}{\sim} \mathsf{id}_B. \end{aligned}$$

and for $a : A$ we let $\tau'(a)$ be the composite

$$\begin{aligned} f\eta(a) &= (\epsilon fgf(a))^{-1} \cdot (\epsilon fgf(a)) \cdot (f\eta(a)) \\ &= (\epsilon fgf(a))^{-1} \cdot (fgf\eta(a)) \cdot (\epsilon f(a)) && \text{(by Lemma 2.16)} \\ &= (\epsilon fgf(a))^{-1} \cdot (f\eta gf(a)) \cdot (\epsilon f(a)) && \text{(by Corollary 2.17)} \\ &\equiv \epsilon' f(a). \end{aligned}$$

□

There is yet another definition also logically equivalent to the other, but we will not have any use of it in this document. To give a small glance of it, the interest is on functions $f : A \to B$ whose homotopy fibers are all contractible. The notions of homotopy fibers and contractibility have not been seen yet, as they are in the upcoming sections and chapters. Similarly as the half-adjoint equivalence, this last definition is close to the category-theoretic viewpoint, as it recalls for instance Quillen's Theorem A.

Now that we have seen multiple ways to express that a function is an equivalence, we choose one of them for our formal definition, purely for convention.

Definition 2.24. We say that a function $f : A \to B$ is an equivalence, provided that the following type is inhabited:

$$\mathsf{isequiv}(f) :\equiv \mathsf{ishae}(f).$$

For two types A and B, the type of *equivalences* from A to B is defined as

$$(A \simeq B) :\equiv \sum_{f:A\to B} \mathsf{isequiv}(f).$$

With the idea of propositions as types, the equivalence $\simeq$ is a generalisation of the logical equivalence $\Leftrightarrow$, because we know it means in particular that we have functions in both directions. In fact, it turns out that if the two types we are comparing are really logical propositions, which is a notion defined in Chapter 3, then logical equivalence and equivalence coincide.

From a more general point of view, having an equivalence $A \simeq B$ can be read as saying that A and B are isomorphic. For instance we can say that they are in bijection if they were both sets, or that they are isomorphic in the algebraic way if they were both groups. Of course we have not defined sets and groups yet, but since we are building a foundation of mathematics, it is no surprise to learn that these notions can also be redefined as part of the theory.

Now that the notion of equivalence has been defined, let us state a simple but important fact. The Lemma 2.11 tells us, given two points $x, y : A$ connected by path $p : x = y$, and a family $P : A \to U$, that p_* and $(p^{-1})_*$ are quasi-inverses. Therefore, the two fibers are equivalent $P(x) \simeq P(y)$. This fact tells us that the dependent constructions of Chapter 1, i.e., the Π-types and the Σ-types are in fact not allowed to construct absolutely everything that one can imagine. Therefore, thanks to the homotopic vision of the types, we see that there are some intrinsic constraints.

With the equivalence defined, we can now also state the relation between $f \sim g$ and $f = g$ for two functions $f, g : \prod_{(x:A)} B(x)$.

Lemma 2.25. *Given two functions $f, g : \prod_{(x:A)} B(x)$, there is a function*

$$\mathsf{happly} : (f = g) \to (f \sim g).$$

Proof. By induction on $p : f = g$, it suffices to define $\mathsf{happly}(\mathsf{refl}_f, x) :\equiv \mathsf{refl}_{f(x)}$. □

Remark. We already mentioned in the remark following Lemma 2.3 that sometimes they are other ways of defining objects, but that in the end they are propositionally equal. For instance, an alternative and more direct definition here of $\mathsf{happly}(p, x)$ given $p : f = g$ and $x : A$, could be

$$\mathsf{ap}_{f\mapsto f(x)}(p) : f(x) = g(x).$$

But since both $\mathsf{happly}(p, x)$ and $\mathsf{ap}_{f\mapsto f(x)}(p)$ were both defined inductively from $\mathsf{refl}_{f(x)}$, we have anyway by induction that

$$\prod_{(p:f=g)} \prod_{(x:A)} \mathsf{happly}(p, x) = \mathsf{ap}_{f\mapsto f(x)}(p).$$

Even though we wish for $f = g$ and $f \sim g$ to be equivalent, it cannot be proven from the basics of our theory and we have to put it in an axiom for now.

Axiom 2.26 (Function extensionality)**.** The function $\mathsf{happly} : (f = g) \to (f \sim g)$ is an equivalence.

In other words, there is a function, called *function extensionality*,

$$\mathsf{funext} : (f \sim g) \to (f = g),$$

such that for each $p : f = g$ and $h : f \sim g$,

$$\mathsf{happly}(\mathsf{funext}(h)) = h, \text{ and}$$
$$\mathsf{funext}(\mathsf{happly}(p)) = p.$$

In fact, the function extensionality axiom can be proven from the other, major axiom of homotopy type theory, namely Voevodsky's univalence axiom. We first define a function which explains how to go from an equality to an equivalence. The univalence axiom then claims something more about this function.

Lemma 2.27. *Given two types A, B, there is a function*

$$\mathsf{idtoeqv} : (A = B) \to (A \simeq B).$$

Proof. As for the last lemma, we can define $\mathsf{idtoeqv}$ by induction, but this time we prefer the choice of having a more direct definition which makes use of the transport on the family $\mathsf{id}_{\mathcal{U}} : \mathcal{U} \to \mathcal{U}$. We set

$$\mathsf{idtoeqv} :\equiv \mathsf{transport}^{X \mapsto X} : (A = B) \to (A \to B).$$

We only need to check that this function gives an equivalence. But for any $p : A = B$, then $p_* : A \to B$ has the quasi-inverse $(p^{-1})_* : B \to A$ thanks to Lemma 2.11, and is therefore an equivalence by Lemma 2.20. □

Axiom 2.28 (Univalence)**.** The function $\mathsf{idtoeqv} : (A = B) \to (A \simeq B)$ is an equivalence.

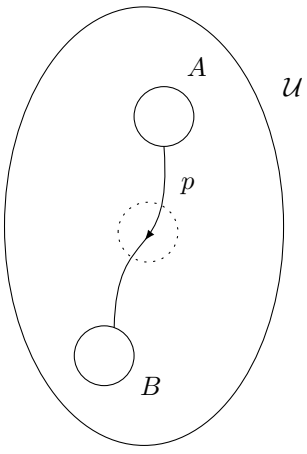
In other words, there is a function

$$\mathsf{ua} : (A \simeq B) \to (A = B),$$

such that for all $p : A = B$ and $f : A \simeq B$,

$$\mathsf{idtoeqv}(\mathsf{ua}(f)) = f, \text{ and}$$
$$\mathsf{ua}(\mathsf{idtoeqv}(p)) = p.$$

The univalence axiom has been an essential part in the progress and development of HoTT. In one direction, if one can continually go from A to B inside $\mathcal{U}$ with a path, then transport offers us an isomorphism between A and B. But in the other direction, we assume that the universe $\mathcal{U}$ is kind enough so that an isomorphism between A and B is enough data to infer the existence of a path between them.



Moreover, the univalence can also be read as saying that A and B are equal if and only if they are isomorphic. This is the common practice in mathematics when we identify for instance bijective sets, or isomorphic groups as the same object, but in HoTT we formalise it.

The univalence axiom will not be employed until the Chapter 3.

Here is another homotopic notion, which is related to equivalence.

Definition 2.29. Given two types A and B, we say that B is a *retract* of A if we have a function $r : A \to B$, called a *retraction*, that admits another function $s : B \to A$, called its *section*, such that there is a homotopy $r \circ s \sim \mathsf{id}_B$.

Remark. Observe that having two equivalent types $A \simeq B$ implies directly from the definitions that X is a retract of Y and that Y is a retract of X.

2.3 Combinations with type formers

Now that we have seen the basics about the homotopic environment of our theory with the paths, the transport inside fibrations, function existentionality and the univalence axiom, it is worth looking at how all of this combines with the different type formers of Chapter 1. Here are a few properties in bulk.

Theorem 2.30. *Given $A, B : \mathcal{U}$ and $x, y : A \times B$, we have the following equivalence*

$$(x =_{A\times B} y) \simeq \big((\mathsf{pr}_1\, x =_A \mathsf{pr}_1\, y) \times (\mathsf{pr}_2\, x =_B \mathsf{pr}_2\, y)\big).$$

Proof. We define two quasi-inverse maps. The first one is

$$\begin{aligned} f : (x = y) &\to \big((\mathsf{pr}_1\, x = \mathsf{pr}_1\, y) \times (\mathsf{pr}_2\, x = \mathsf{pr}_2\, y)\big) \\ r &\mapsto (\mathsf{pr}_1\, r, \mathsf{pr}_2\, r). \end{aligned}$$

The second is denoted

$$\mathsf{pair}^{=} : \big((\mathsf{pr}_1\, x = \mathsf{pr}_1\, y) \times (\mathsf{pr}_2\, x = \mathsf{pr}_2\, y)\big) \to (x = y).$$

To define it, we use the cartesian product induction and suppose that our elements are pairs $x \equiv (a, b)$ and $y \equiv (a', b')$, for some $a, a' : A$ and $b, b' : B$. Let us take $p : a = a'$ and $q : b = b'$. By doing two path inductions, we suppose $p \equiv \mathsf{refl}_a$ and $q \equiv \mathsf{refl}_b$. Then, we let

$$\mathsf{pair}^{=}(\mathsf{refl}_a, \mathsf{refl}_b) :\equiv \mathsf{refl}_{(a,b)}.$$

It remains to prove that

$$\prod_{r:x=y} \mathsf{pair}^{=} f r = r, \quad \text{and}$$

$$\prod_{z:(\mathsf{pr}_1\, x=\mathsf{pr}_1\, y)\times(\mathsf{pr}_2\, x=\mathsf{pr}_2\, y)} f\, \mathsf{pair}^{=} z = z.$$

First, let us take $r : x = y$. We do a path induction to suppose $r \equiv \mathsf{refl}_x$ and a product induction to suppose $x \equiv (a, b)$. But then,

$$\mathsf{pair}^{=}\ f\ \mathsf{refl}_{(a,b)} \equiv \mathsf{pair}^{=}(\mathsf{refl}_a, \mathsf{refl}_b) \equiv \mathsf{refl}_{(a,b)}.$$

Second, let us take $z : (\mathsf{pr}_1\, x = \mathsf{pr}_1\, y) \times (\mathsf{pr}_2\, x = \mathsf{pr}_2\, y)$. We do product inductions to suppose $x \equiv (a, b)$, $y \equiv (a', b')$ and $z \equiv (p, q)$, and path inductions to suppose $p \equiv \mathsf{refl}_a$ and $q \equiv \mathsf{refl}_b$. But then,

$$f\ \mathsf{pair}^{=}(\mathsf{refl}_a, \mathsf{refl}_b) \equiv f\ \mathsf{refl}_{(a,b)} \equiv (\mathsf{refl}_a, \mathsf{refl}_b).$$

□

The functions $\mathsf{ap}_{\mathsf{pr}_1}$, $\mathsf{ap}_{\mathsf{pr}_2}$ and $\mathsf{pair}^{=}$ have a lot of properties that are really useful to characterise paths inside the product type. For instance, given appropriate paths p, p', q and q', here are two formulas that we will need later on:

$$\mathsf{pair}^{=}(p^{-1}, q^{-1}) = \mathsf{pair}^{=}(p, q)^{-1},$$
$$\mathsf{pair}^{=}(p \cdot q, p' \cdot q') = \mathsf{pair}^{=}(p, p') \cdot \mathsf{pair}^{=}(q, q').$$

Proving them is immediate by path induction.

Theorem 2.31. *Given $B : A \to \mathcal{U}$ and $z, z' : \sum_{(x:A)} B(x)$, we have the following equivalence.*

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

Proof. Similarly as in the last theorem, we define two quasi-inverse maps using multiple inductions

$$\begin{cases} f : (z = z') \longleftrightarrow \left(\sum_{p : \mathsf{pr}_1\, z = \mathsf{pr}_1\, z'} p_*(\mathsf{pr}_2\, z) = \mathsf{pr}_2\, z \right) : \mathsf{pair}^{=} \\ \qquad\qquad r \longmapsto (\mathsf{ap}_{\mathsf{pr}_1} r, \mathsf{apd}_{\mathsf{pr}_2} r) \\ \mathsf{refl}_{(a,b)} \longmapsfrom (\mathsf{refl}_a, \mathsf{refl}_b). \end{cases}$$

The proofs that they are quasi-inverses are also done by multiple inductions, as before, and it suffices to see that

$$\mathsf{pair}^{=}\ f\ \mathsf{refl}_{(a,b)} \equiv \mathsf{pair}^{=}(\mathsf{refl}_a, \mathsf{refl}_b) \equiv \mathsf{refl}_{(a,b)},$$

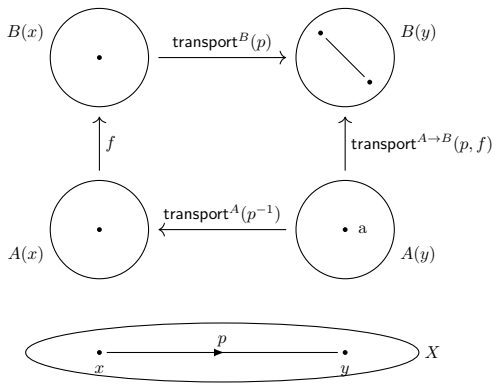
and

$$f\ \mathsf{pair}^{=}(\mathsf{refl}_a, \mathsf{refl}_b) \equiv f\ \mathsf{refl}_{(a,b)} \equiv (\mathsf{refl}_a, \mathsf{refl}_b).$$

□

Lemma 2.32. *Given two families $A, B : X \to U$, we can make a slight abuse of notation and consider the family "$A \to B$" $: X \to \mathcal{U}$ defined by $(A \to B)(x) :\equiv A(x) \to B(x)$ for $x : X$. The transport along this family is, for $p : x =_X y$ and $a : A(y)$,*

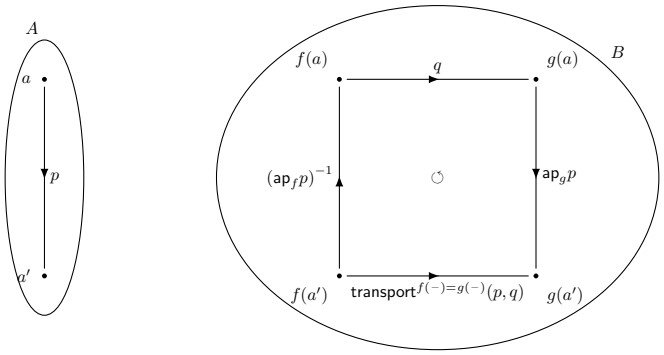
$$\mathsf{transport}^{A \to B}(p, f)(a) = \mathsf{transport}^{B}(p, f(\mathsf{transport}^{A}(p^{-1}, a))).$$



Proof. By induction on p, it suffices to consider when $p \equiv \mathsf{refl}_x$. But then, both sides of our equality are already judgmentally equal to $f(a)$. □

Lemma 2.33. *Given two functions $f, g : A \to B$ and two paths $p : a =_A a'$, $q : f(a) =_B g(a)$, the transport along equality type is*

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



Proof. By induction on p and then on q, it is immediate as

$$\begin{aligned}\mathsf{transport}^{f(-)=g(-)}(\mathsf{refl}_a, \mathsf{refl}_{f(a)}) &\equiv \mathsf{refl}_{f(a)},\\ \mathsf{ap}_f\mathsf{refl}_a &\equiv \mathsf{refl}_{f(a)}.\end{aligned}$$

□

Corollary 2.34. *Given a type A, $a : A$ and $p : x =_A y$, we have*

$$\begin{aligned}\mathsf{transport}^{x\mapsto a=x}(p) &= (-\cdot p),\\ \mathsf{transport}^{x\mapsto x=a}(p) &= (p^{-1}\cdot -),\\ \mathsf{transport}^{x\mapsto x=x}(p) &= (p^{-1}\cdot - \cdot p).\end{aligned}$$

Chapter 3

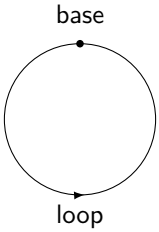
Adapting Homotopy Theory

Now that the basic of HoTT have been established, it is time to come back to more common mathematical considerations. One can wonder about the way to perform standard homotopy theory inside our framework. First, here is a useful terminology.

Definition 3.1. A *higher inductive type* is a type with constructors not only generated by points and functions that give points but also by having paths of some degree.

Indeed, this is a possibility that we have not exploited yet. For instance, $\mathbb{N}$ is generated by a point 0 and a function $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$, but we do not suppose any homotopic data in the constructors. As a basic example, the circle S^1 is a higher inductive type generated by

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



Observe that this definition contains the essence of HoTT with the minimalist abstraction discussed in the introduction, where the object is reduced to its core idea.

The recursion principle for S^1 is quite straightforward. Given a type A, defining a function $f : \mathsf{S}^1 \to A$ requires to have already a "circle" in A, i.e., a point a and a loop at it $p : a = a$. We have then our function f characterised by

$$f(\mathsf{base}) :\equiv a$$
$$\mathsf{ap}_f(\mathsf{loop}) \coloneqq p.$$

Note that the definition for the point is done with a judgmental equality, as done before. However, the question of using also one or not for the path arises. Indeed, the definition of ap_f in Lemma 2.6 is not unique, but all its definitions are propositionally equal. Therefore, it makes sense to consider the path data only up to propositional equality too. The symbol ":=" is used to define something as propositionally equal to something else.

For the induction principle, take a family $B : \mathsf{S}^1 \to \mathcal{U}$. We want to define a function $g : \prod_{(x:\mathsf{S}^1)} B(x)$. We need a point for the image $g(\mathsf{base})$, which lies above base, in its fiber $B(\mathsf{base})$, say $b : B(\mathsf{base})$. But for loop, we want to send it to a path from b to b which "lies above loop" in our fibration B. In the recursion, we used ap_f, and since we are now in the dependent case, we have from Lemma 2.12 the analogous

$$\mathsf{apd}_g : \prod_{p:b=b} (p_*(b) = b).$$

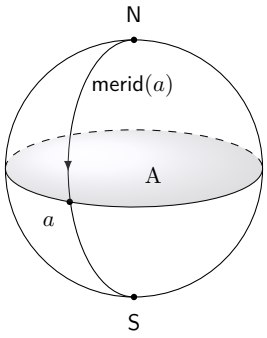


Figure 1: Suspension ΣA

Therefore, this path is lying above loop similarly as what happens in Theorem 2.31. Since it is in fact a recurring idea we give it a name.

Definition 3.2. Given a family $P : X \to \mathcal{U}$, $p : x =_X y$, and $u : P(x), v : P(y)$ in different fibers, the *dependent path* type from u to v is defined as

$$(u =_p^P v) :\equiv (p_*(u) =_{P(y)} v).$$

Then, the induction principle for S^1 can be stated as such. Given

- $b : B(\mathsf{base})$, and
- $l : b =_{\mathsf{loop}}^B b$,

there is a function $g : \prod_{(x:\mathsf{S}^1)} B(x)$ with $g(\mathsf{base}) :\equiv b$ and $g(\mathsf{loop}) \coloneqq l$.

Observe that we met a complication in the induction principle with the need of a dependent path. If we want to keep defining higher spheres $\mathsf{S}^2, \mathsf{S}^3, \ldots$ this way, things become unnecessarily tedious really quickly, with dependent paths of higher degree. Therefore, another approach presented in the next section is generally preferred.

3.1 Suspension

Definition 3.3. A *pointed type* is a pair (A, a) of a type A and a *basepoint* a:A.

Given a pointed type (A, a_0), a common construction in homotopy theory is to take its suspension, which is another pointed type. Here, let us start by defining first the unpointed version in our theory.

Definition 3.4. Given a type A, its *suspension* ΣA is the higher inductive type generated by

- a point $\mathsf{N} : \Sigma A$,
- a point $\mathsf{S} : \Sigma A$, and
- a function $\mathsf{merid} : A \to (\mathsf{N} = \mathsf{S})$.

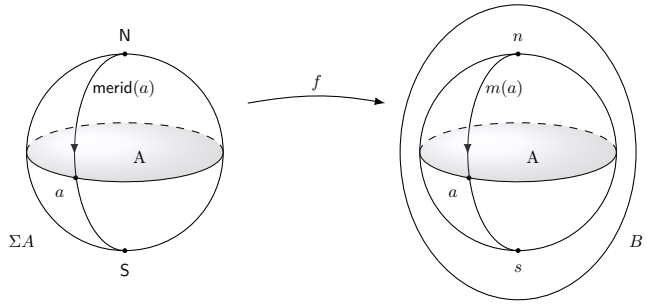
Schematically, one can picture the suspension as creating a globe with a north pole, a south pole and meridians, when starting with only the equator.

The recursion principle for a suspension ΣA is as follows. Given a type B,

- points $n, s : B$, and
- a function $m : A \to (n = s)$,

then there is a function $f : \Sigma A \to B$ characterised by

- $f(\mathsf{N}) :\equiv n$,
- $f(\mathsf{S}) :\equiv s$,
- and for all $a : A$, $\mathsf{ap}_f(\mathsf{merid}(a)) \coloneqq m(a)$.



The induction principle for a suspension ΣA is as follows. Given a family $B : \Sigma A \to \mathcal{U}$,

- a point $n : B(\mathsf{N})$,
- a point $s : B(\mathsf{S})$, and
- a function $m : A \to (n =^P_{\mathsf{merid}(a)} s)$,

then there is a function $g : \prod_{(x:\Sigma A)} B(x)$ characterised by

- $g(\mathsf{N}) :\equiv n$,
- $g(\mathsf{S}) :\equiv s$, and
- $\mathsf{apd}_g(\mathsf{merid}(a)) \coloneqq m(a)$.

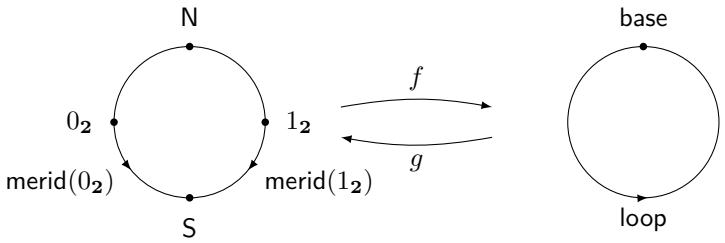
Given a pointed type (A, a_0), we have therefore another pointed type $(\Sigma A, N)$. Observe that we consider the unreduced suspension, and not the reduced suspension as it is commonly done in homotopy theory. The reduced version has both poles N and S being identified by collapsing one meridian, $\mathsf{merid}(a_0)$, the one passing through the original base point a_0. Both constructions are of course equivalent.

Having now the suspension in our toolkit, a more streamlined approach to define higher spheres is possible. Observe first that S^0 consists only of two points, therefore in HoTT it correspond to the type $\mathbf{2}$. Moreover, if we take the suspension of an n-sphere, Figure 1 tells us that we added a new dimension to obtain something corresponding to an $(n+1)$-sphere. To validate this way of looking at our spheres, let us first check that it agrees with our first definition of S^1.

Lemma 3.5. $\Sigma\mathbf{2} \simeq \mathsf{S}^1$.

Proof. We define two functions by recursion, $f : \Sigma\mathbf{2} \to \mathsf{S}^1$ and $g : \mathsf{S}^1 \to \Sigma\mathbf{2}$.

$$\begin{cases} f(\mathsf{N}) :\equiv \mathsf{base}, \\ f(\mathsf{S}) :\equiv \mathsf{base}, \\ \mathsf{ap}_f(\mathsf{merid}(0_{\mathbf{2}})) \coloneqq \mathsf{loop}, \\ \mathsf{ap}_f(\mathsf{merid}(1_{\mathbf{2}})) \coloneqq \mathsf{refl}_{\mathsf{base}}. \end{cases} \qquad \begin{cases} g(\mathsf{base}) :\equiv \mathsf{N}, \\ \mathsf{ap}_g(\mathsf{loop}) \coloneqq \mathsf{merid}(0_{\mathbf{2}}) \cdot \mathsf{merid}(1_{\mathbf{2}})^{-1}. \end{cases}$$



Basically, f contracts the meridian $\mathsf{merid}(\mathbf{1_2})$ and g injects S^1 as the figure suggests it. We need to check that f and g are quasi-inverses.

First, let us show that

$$\prod_{x:\Sigma\mathbf{2}} \underbrace{gf(x) = x}_{\equiv:B(x)}.$$

This is done by $\Sigma\mathbf{2}$-induction, which requires three things.

- An inhabitant of $B(\mathsf{N}) \equiv (\mathsf{N} = \mathsf{N})$, we choose $n :\equiv \mathsf{refl}_{\mathsf{N}}$,
- an inhabitant of $B(\mathsf{S}) \equiv (\mathsf{N} = \mathsf{S})$, we choose $s :\equiv \mathsf{merid}(\mathbf{1_2})$,
- and for all $y : \mathbf{2}$, an inhabitant of

  $$\mathsf{transport}^{x\mapsto(gf(x)=x)}(\mathsf{merid}(y), n) = s.$$

  Lemma 2.33 tells us that it suffices to prove

  $$gf(\mathsf{merid}(y))^{-1} \cdot \mathsf{merid}(y) = \mathsf{merid}(\mathbf{1_2}).$$

  By $\mathbf{2}$-induction, we can look at only two cases.

  – If $y \equiv \mathbf{0_2}$, then

  $$\begin{aligned} gf(\mathsf{merid}(\mathbf{0_2}))^{-1} \cdot \mathsf{merid}(\mathbf{0_2}) &= g(\mathsf{loop})^{-1} \cdot \mathsf{merid}(\mathbf{0_2}) \\ &= \mathsf{merid}(\mathbf{1_2})^{-1} \cdot \mathsf{merid}(\mathbf{0_2})^{-1} \cdot \mathsf{merid}(\mathbf{0_2}) \\ &= \mathsf{merid}(\mathbf{1_2}). \end{aligned}$$

  – If $y \equiv \mathbf{1_2}$, then

  $$\begin{aligned} gf(\mathsf{merid}(\mathbf{1_2}))^{-1} \cdot \mathsf{merid}(\mathbf{1_2}) &= g(\mathsf{refl}_{\mathsf{base}}) \cdot \mathsf{merid}(\mathbf{1_2}) \\ &= \mathsf{refl}_{g(\mathsf{base})} \cdot \mathsf{merid}(\mathbf{1_2}) \\ &= \mathsf{merid}(\mathbf{1_2}). \end{aligned}$$

Second, let us show that

$$\prod_{x:\mathsf{S}^1} \underbrace{fg(x) = x}_{\equiv:C(x)}.$$

This is done by S^1-induction, which requires two things.

- An inhabitant of $C(\mathsf{base}) \equiv (\mathsf{base} = \mathsf{base})$, we choose $\mathsf{refl}_{\mathsf{base}}$, and
- a dependent path

  $$\mathsf{transport}^{x\mapsto fg(x)=x}(\mathsf{loop}, \mathsf{refl}_{base}) = \mathsf{refl}_{\mathsf{base}}.$$

  By Lemma 2.33, it suffices to prove

  $$fg(\mathsf{loop})^{-1} \cdot \mathsf{refl}_{\mathsf{base}} \cdot \mathsf{loop} = \mathsf{refl}_{\mathsf{base}},$$

  which holds since

  $$\begin{aligned} fg(\mathsf{loop})^{-1} &= f(\mathsf{merid}(\mathbf{0_2}) \cdot \mathsf{merid}(\mathbf{1_2})^{-1})^{-1} \\ &= f(\mathsf{merid}(\mathbf{0_2}))^{-1} \cdot f(\mathsf{merid}(\mathbf{1_2})) \\ &= \mathsf{loop}^{-1} \cdot \mathsf{refl}_{\mathsf{base}}. \end{aligned}$$

□

Therefore, here is a valid alternative definition of n-spheres.

Definition 3.6. We define the *n-spheres* by induction on $n : \mathbb{N}$ as

$$\begin{aligned} \mathsf{S}^0 &:\equiv \mathbf{2}, \\ \mathsf{S}^{n+1} &:\equiv \Sigma\mathsf{S}^n. \end{aligned}$$

In fact, observe that we can even start with $\mathsf{S}^{-1} :\equiv \mathbf{0}$ because $\Sigma\mathbf{0} \simeq \mathbf{2}$.

3.2 Homotopy n-types

An important notion in homotopy theory is n-connectedness. In HoTT, the dual notion of n-truncatedness is more intuitive to introduce first, so let us start with this one.

Definition 3.7. (i) A type A is *contractible* if the following predicate is inhabited

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

When it is the case, the element a is called a *center of contraction*.

(ii) A type A is a *mere proposition* if the following predicate is inhabited

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

(iii) A type A is a *set* if the following type is inhabited

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

The definitions of sets and of contractible types might seem somewhat disturbing at first, so let us explain them.

Suppose we have a set A. The Definition 3.7(iii) tells us that every two paths of A sharing endpoints are equal. In other words, all the homotopic data of degree one in A is trivial. Intuitively, the notion of set that we expect is something like a discrete collection of points, or of a topological space with discrete topology. But fortunately, since everything in homotopy type theory is functorial, natural and continuous, our definition suffices to imply that all the homotopic structure of degree $\geqslant 1$ is trivial and thus that A is essentially just points. This fact will come again in an upcoming lemma. Therefore, sets are analogous to 0-groupoids in category theory. Because of this, sets are also called 0-types.

Similarly, suppose we have a contractible type B. The Definition 3.7(i) tells us that all points $x : A$ are linked by a path to a specific point $a : A$. This seems closer to the notion of path-connectedness, as nothing seems to prevent B from having "holes". But fortunately, here again it suffices to describe what we want at the bottom homotopic level and the properties of type theory implies the rest. Therefore, we will see for instance in Lemma 3.9 that contractible types are (homotopy) equivalent to the unit type $\mathbf{1}$, as desired.

The definition of mere proposition corresponds to what was announced in Section 1.6. A type C which is a proposition has two choices, either it is empty or it is inhabited but then all its points are equal. Therefore C contains a binary information, similarly to a proposition in logic which has only two states, either true or false. This definition has also a homotopic meaning, hence the reason why we defined it along the two others. This will be detailed in the upcoming lemmas.

With the notion of mere proposition, let us come back to some facts about past definitions.

Lemma 3.8. *Let A be a type. Then, the following types are mere propositions:* $\mathsf{isContr}(A)$, $\mathsf{isProp}(A)$, $\mathsf{isSet}(A)$, $\mathsf{isbiinv}(A)$, $\mathsf{ishae}(A)$, *and* $\mathsf{isequiv}(A)$.

We omit the demonstration as it is not particularly relevant to the purpose of the document. The different proofs can be found in [7]. The implicit convention we use is that predicates of the form "$\mathsf{is}\ldots$" are meant to give mere propositions. This lemma explains for instance why it is preferable to have notions such as $\mathsf{isbiinv}$ or ishae and not only qinv, as we discussed in Section 2.2. Indeed, Theorem 2.31 and Lemma 3.8 implies that two equivalences $(f, e_1), (g, e_2) : A \simeq B$ are equal if and only if their underlying functions $f, g : A \to B$ are equal. Therefore, there is no real need to make the distinction between an equivalence (f, e) and the underlying function f. Hence, it is common to abuse notation and write $f : A \simeq B$ or $f : A \xrightarrow{\simeq}$ to say that we have an equivalence.

To get a better general understanding of Definition 3.7, here are a few lemmas explaining how each definition is related to the others.

Lemma 3.9. *Given a type A, the following are equivalent:*

i) A is contractible.

ii) A is a mere proposition and there is some $a : A$.

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

Proof. ($i \Rightarrow ii$) If A is contractible, then there is $z : \mathsf{isContr}(A)$. By Σ-induction, we can suppose $z \equiv (a, p)$ with $a : A$ and $p : \prod_{(x:A)} a = x$. Therefore, we have indeed $a : A$ and

$$q : \mathsf{isProp}(A) \equiv \prod_{x,y:A} (x = y)$$
$$q(x, y) :\equiv p(x)^{-1} \cdot p(y).$$

($ii \Rightarrow iii$) Suppose $a : A$ and $q : \mathsf{isProp}(A)$. We define two maps, the second one by $\mathbf{1}$-induction.

$$\begin{cases} f : A \to \mathbf{1} \\ \quad x \mapsto \star \end{cases}, \qquad \begin{cases} g : \mathbf{1} \to A \\ \quad \star \mapsto a \end{cases}.$$

Now, for all $x : A$ we have $gf(x) = x$ by $q(a, x)$, since $gf(x) \equiv a$. And for all $y : \mathbf{1}$ we have $fg(y) = y$ by $\mathbf{1}$-induction with $\mathsf{refl}_\star : fg(\star) = \star$, since $fg(\star) \equiv \star$.
($iii \Rightarrow i$) Suppose $A \simeq \mathbf{1}$. So we have quasi-inverse functions, say $f : A \to \mathbf{1}$ and $g : \mathbf{1} \to A$. Let $a :\equiv g(\star)$, which will be our center of contraction. Indeed, for all $x : A$, we have

$$\begin{aligned} x &= gf(x) && \text{(since they are quasi-inverses)} \\ &= g(\star) && \text{(by the uniqueness principle of } \mathbf{1}) \\ &\equiv a. \end{aligned}$$

□

In fact, these definitions are the beginning of a classification of types. To see that better, let us prove two lemmas.

Lemma 3.10. *Let A be a type.*

i) If A is contractible, then A is a mere proposition.

ii) If A is a mere proposition, then A is a set.

Proof. i) This was already proven in the last lemma with ($i \Rightarrow ii$).

ii) Assume we have $f : \mathsf{isProp}(A)$. If we try to prove directly $\mathsf{isSet}(A)$ by path induction, we cannot go very far. Instead, we show that we have an inhabitant g of

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

We define $g(x, y, z, p, q)$ to be the following composite.

$$\begin{aligned} p &= f(x, y)^{-1} \cdot f(x, y) \cdot p && \text{(by Lemma 2.5)} \\ &= f(x, y)^{-1} \cdot \mathsf{transport}^{y \mapsto x=y}(p, f(x, y)) && \text{(by Lemma 2.33)} \\ &= f(x, y)^{-1} \cdot f(x, z) && \text{(by } \mathsf{apd}_{y \mapsto f(x,y)}(p) \text{ from Lemma 2.12)} \\ &= f(x, y)^{-1} \cdot \mathsf{transport}^{y \mapsto x=y}(q, f(x, y)) && \text{(by } \mathsf{apd}_{y \mapsto f(x,y)}(q) \text{ from Lemma 2.12)} \\ &= f(x, y)^{-1} \cdot f(x, y) \cdot q && \text{(by Lemma 2.33)} \\ &= q && \text{(by Lemma 2.5).} \end{aligned}$$

Then, we have

$$h : \mathsf{isSet}(A)$$
$$h(x, y, p, q) :\equiv g(x, x, y, p, q).$$

□

Lemma 3.11. *Let A be a type.*

i) A is a set $\Leftrightarrow$ for all $x, y : A$, the type $x = y$ is a mere proposition.

ii) A is a mere proposition $\Leftrightarrow$ for all $x, y : A$, the type $x = y$ is contractible.

Proof. i) This is immediate by observing that

$$\begin{aligned}\mathsf{isSet}(A) &\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q) \\ &\equiv \prod_{x,y:A} \mathsf{isProp}(x = y).\end{aligned}$$

ii) ($\Rightarrow$) Suppose that A is a mere proposition, with a witness $f : \mathsf{isProp}(A)$. By Lemma 3.10, A is also a set. Thus, by the first part of the lemma, for each $x, y : A$ the type $x = y$ is a mere proposition. But $x = y$ is also inhabited by $f(x, y)$, therefore by Lemma 3.9 it is contractible.

($\Leftarrow$) Suppose that $f : \prod_{(x,y:A)} \mathsf{isContr}(x = y)$. Then, we have

$$\begin{cases} g : \mathsf{isProp}(A) \\ g(x, y) :\equiv \mathsf{pr}_1(f(x, y)) \end{cases},$$

since the first component of $f(x, y)$ is the center of contraction and thus is of type $x = y$.

□

The last lemma gives us a hint about how to keep creating new definitions. Recall that sets are also called 0-types. Therefore, we could say that a type is a 1-type given that all its equality types are sets. We can even keep going by replacing sets with 1-types and by repeating the process. Therefore, it motivates the following definition.

Definition 3.12. Let A be a type. For all $n \geqslant -2$, we say that A is an *n-type* or *n-truncated type* when the predicate $\mathsf{is}\text{-}n\text{-}\mathsf{type}(A) : \mathcal{U}$ is inhabited, which is defined inductively by

$$\mathsf{is}\text{-}n\text{-}\mathsf{type}(A) :\equiv \begin{cases} \mathsf{isContr}(A) & , \text{ if } n = -2, \\ \prod_{(x,y:A)} \mathsf{is}\text{-}n'\text{-}\mathsf{type}(x = y) & , \text{ if } n = n' + 1. \end{cases}$$

Remark. Note first that in the definition, there is an induction on $\mathbb{Z}_{\geqslant -2}$. As we want everything to be internal to our theory, just observe that the type $\mathbb{Z}_{\geqslant -2}$ can be defined with the generators

- $-2 : \mathbb{Z}_{\geqslant -2}$, and
- $\mathsf{succ} : \mathbb{Z}_{\geqslant -2} \to \mathbb{Z}_{\geqslant -2}$.

Therefore, we have a quite obvious equivalence $\mathbb{Z}_{\geqslant -2} \simeq \mathbb{N}$, as we have the quasi-inverse maps which are informally just "shifting every integer by 2". We can thus perform $\mathbb{Z}_{\geqslant -2}$-induction just as $\mathbb{N}$-induction.

Definition 3.13. Given $n \geqslant -2$, the collection of all n-types is denoted

$$n\text{-}\mathsf{Type} :\equiv \sum_{A:\mathcal{U}} \mathsf{is}\text{-}n\text{-}\mathsf{type}(A).$$

Remark. By the last lemma, given a type A we have that

$$\begin{aligned} A \text{ is a set} &\Leftrightarrow A \text{ is a 0-type}, \\ A \text{ is a mere proposition} &\Leftrightarrow A \text{ is a } (-1)\text{-type}, \\ A \text{ is contractible} &\Leftrightarrow A \text{ is a } (-2)\text{-type}. \end{aligned}$$

The reason the classification starts at -2 follows from our earlier observation. That is, a 0-type A has trivial higher homotopy structures, with nothing interesting above degree 0. Then, the levels -1 and -2 are kind of conventions to have a practical way of starting our hierarchy, as it classify types equivalent to $\mathbf{0}$ and $\mathbf{1}$. As in homotopy theory, we consider that the empty type $\mathbf{0}$ has a non trivial (-1)-homotopy group "$\pi_{-1}(\mathbf{0})$", since we can consider the trivial function $\mathsf{S}^{-1} \to \mathbf{0}$, where $\mathsf{S}^{-1} \equiv \mathbf{0}$ as in Definition 3.6.

Going now up with n, the same phenomenon happens. That is, an n-type A will have in fact a trivial homotopy structure at degree $n+1$ and above. This fact is already visible in the following result.

Lemma 3.14. *For all type A and all $n \geqslant -2$,*

$$A \text{ is an } n\text{-type } \Rightarrow A \text{ is an } (n+1)\text{-type.}$$

Proof. We prove the claim by induction on n.

- If $n \equiv -2$, this case was already proven in Lemma 3.10.
- If $n \equiv n' + 1$, suppose that we have an $(n'+1)$-type A. By definition, it means that for all $x, y : A$, the type $x = y$ is an n-type. By induction hypothesis, $x = y$ is thus also an $(n'+1)$-type. Therefore, A is an $(n'+2)$-type.

□

The following theorem is given without proof as it uses intermediate results that are not useful for the goal of the document. A proof can be found in [7].

Theorem 3.15 (Thm 7.1.11 in [7])**.** *Given $n \geqslant -2$, the type n-Type is an $(n+1)$-type.*

One may wonder which operations preserve n-truncatedness. Being a retract turns out to be one of those cases.

Lemma 3.16. *If X is an n-type for some $n \geqslant -2$ and Y is a retract of X, then Y is also an n-type.*

Proof. By hypothesis, we have a retraction $r : X \to Y$, a section $s : Y \to X$ and an homotopy $\epsilon : r \circ s \sim \mathsf{id}_Y$. We proceed by induction on n.

- If $n \equiv -2$, having X contractible means that we can suppose $x_0 : X$ and $f : \prod_{(x:X)}(x_0 = x)$. Then, let $y_0 :\equiv r(x_0) : Y$ and

$$\begin{cases} g : \prod_{y:Y}(y_0 = y) \\ g(y) :\equiv \mathsf{ap}_r(f(sy)) \cdot \epsilon(y). \end{cases}$$

$$y_0 \equiv r(x_0) \xrightarrow{\mathsf{ap}_r(f(sy))} rs(y) \xrightarrow{\epsilon(y)} y$$

  Therefore, $(y_0, g) : \mathsf{isContr}(Y)$.

- Suppose $n \equiv n' + 1$ and take $y, y' : Y$. We want to prove that $y = y'$ is an n'-type. Since X is an $(n'+1)$-type, we know that $s(y) = s(y')$ is an n'-type. Thus, the induction hypothesis will allow us to conclude in case we manage to prove that $y = y'$ is a retract of $ids(y)s(y')$. And indeed, we have a retraction

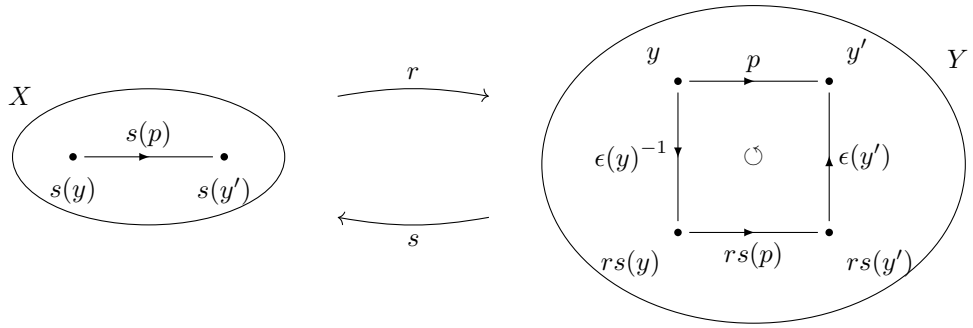
$$\mathsf{ap}_s : (y = y') \to (s(y) = s(y'))$$

  and a section

$$\begin{cases} f : (s(y) = s(y')) \to (y = y') \\ f(q) :\equiv \epsilon(y)^{-1} \cdot rs(p)\epsilon(y'). \end{cases}$$

  The homotopy to verify is given by Lemma 2.16, which tells us that for all $p : y = y'$, then

$$p = \epsilon(y)^{-1} \cdot \mathsf{ap}_r(q) \cdot \epsilon(y').$$



□

Corollary 3.17. *If X is an n-type and $Y \simeq X$, then Y is also an n-type.*

Proof. It follows from Lemma 3.16 and from the remark made after Definition 2.29. □

Let us come back shortly on the notion of contractibility, with the two following lemmas being really powerful results.

Lemma 3.18. *Given $A : \mathcal{U}$ and $a : A$, we have that $\sum_{(x:A)}(a = x)$ is contractible.*

Proof. For the center of contraction, we choose (a, refl_a). Take $z : \sum_{(x:A)}(a = x)$. In order to prove that $(a, \mathsf{refl}_a) = z$, we do a Σ-induction to suppose that $z \equiv (x, p)$. Moreover, by Theorem 2.31 it suffices to prove

$$\sum_{q:a=x} q_*(\mathsf{refl}_a) = p.$$

But we can inhabit it, since we have $p : a = x$ and we know that

$$\begin{aligned} \mathsf{transport}^{a=(-)}(p, \mathsf{refl}_a) &= \mathsf{refl}_a \cdot p && \text{(by Corollary 2.34)} \\ &= p. && \text{(by Lemma 2.5)} \end{aligned}$$

□

Lemma 3.19. *Let $A : \mathcal{U}$ and $B : A \to \mathcal{U}$.*

1. *If for each $x : A$ the type $B(x)$ is contractible, then $\sum_{(x:A)} B(x) \simeq A$.*
2. *If A is contractible with center a, then $\sum_{(x:A)} B(x) \simeq B(a)$.*

Proof. 1. By hypothesis, we have for each $x : A$ a center of contraction b_x of $B(x)$ and also some $f_x : \prod_{(b:B(x))}(b_x = b)$. We prove the claim by showing that the function $\mathsf{pr}_1 : \left(\sum_{(x:A)} B(x)\right) \to A$ has the following quasi-inverse:

$$\begin{aligned} g : A &\to \sum_{x:A} B(x) \\ a &\mapsto (a, b_a). \end{aligned}$$

We immediately have $\mathsf{pr}_1 \circ g(a) \equiv a$ for all $a : A$. Moreover, for $z : \sum_{(x:A)} B(x)$ we can do a Σ-induction to suppose that $z \equiv (x, b)$ and observe that $g \circ \mathsf{pr}_1(x, b) \equiv (x, b_x) = (x, b)$ holds thanks to $\overset{=}{\mathsf{pair}}(\mathsf{refl}_x, f_x(b))$.

2. By hypothesis, we have for each $x : A$ a path $q(x) : a = x$. We define two functions as follows.

$$\begin{cases} f : \left(\sum_{x:A} B(x)\right) \to B(a), \\ f(y) :\equiv \mathsf{transport}^P(q(\mathsf{pr}_1\, y)^{-1}, \mathsf{pr}_2\, y), \end{cases} \qquad \begin{cases} g : B(a) \to \sum_{x:A} B(x), \\ g(p) :\equiv (a, p). \end{cases}$$

To see that they are quasi-inverses, let us first take $y : \sum_{(x:A)} B(x)$. By Σ-induction, we can suppose $y \equiv (x, p)$. Then,

$$\begin{aligned} gf(x,p) &:\equiv (a, \mathsf{transport}^P(q(x)^{-1}, p) \\ &= (x,p). \end{aligned}$$

The propositional equality holds thanks to Theorem 2.31, as we do have $q(x) : a = x$ and

$$\begin{aligned} (q(x))_*(q(x)^{-1})_*(p) &= (q(x)^{-1} \cdot q(x))_*(p) && \text{(by Lemma 2.11)} \\ &= (\mathsf{refl}_x)_*(p) \\ &\equiv p. \end{aligned}$$

In the other direction, let us take $p : P(a)$. Since A is contractible, by Lemma 3.10 it is also a set, which implies $q(a) = \mathsf{refl}_a$, and therefore

$$\begin{aligned} fg(p) &\equiv \mathsf{transport}^P(q(a)^{-1}, p) \\ &= \mathsf{transport}^P(\mathsf{refl}_a, p) \\ &\equiv p \end{aligned}$$

□

3.3 Truncations

Now that the notion of n-type has been established, one may wonder if there is a way to turn an arbitrary type into an n-type. The answer is yes, thanks to the notion of truncations. It is the translation in HoTT of the Postnikov truncations from homotopy theory. First, let us start with a few standard definitions translated into homotopy type theory.

Definition 3.20. Given a pointed type (A, a), its *loop space* is the pointed type

$$\Omega(A, a) :\equiv (\mathsf{refl}_a, a = a).$$

In fact, we can recursively define, for $n : \mathbb{N}$, the *n-fold iterated loop space* as

$$\begin{aligned} \Omega^0(A, a) &:\equiv (A, a), \\ \Omega^{n+1}(A, a) &:\equiv \Omega(\Omega^n(A, a)). \end{aligned}$$

Definition 3.21. Given pointed types (A, a_0) and (B, b_0), the type of *pointed functions* from (A, a_0) to (B, b_0) is defined as

$$\mathsf{Map}_*(A, B) :\equiv \sum_{f:A\to B} f(a_0) = b_0.$$

To motivate the definition of truncations, let us start with a few results.

Lemma 3.22 (Associativity of Σ-types)**.** *Given $A : \mathcal{U}$, $B : A \to \mathcal{U}$ and $C : (\sum_{(x:A)} B(x)) \to \mathcal{U}$, we have*

$$\left(\sum_{(x:A)} \sum_{(y:B(x))} C((x,y))\right) \simeq \left(\sum_{z:\sum_{(x:A)} B(x)} C(z)\right)$$

Proof. The proof is really straightforward. To show the equivalence, observe that there are two quasi-inverse functions, which we define by Σ-induction as follows. In one direction, an element $(a, (b, c))$ is sent to $((a, b), c)$, and in the other direction, an element $((a, b), c)$ is sent to $(a, (b, c))$. The fact that the compositions of both maps gives the identities is immediate, again by Σ-induction, because of the definitions of our functions. □

The next lemma is the classical suspension-loop adjunction formula. It is reassuring to discover that our type theoretical definitions do behave as intended. The proofs of both the lemma and of the following theorem are omitted, as they require some intermediate results that are not of the greatest interest for the goal of the document. The proof can be found in [7].

Lemma 3.23 (Lemma 6.5.3 in [7])**.** *Given pointed type (A, a_0) and (B, b_0), we have*

$$\mathsf{Map}_*(\Sigma A, B) \simeq \mathsf{Map}_*(A, \Omega B).$$

It has an immediate corollary

Corollary 3.24. *Given a pointed type (B, b_0), we have*

$$\mathsf{Map}_*(\mathsf{S}^n, B) \simeq \Omega^n B.$$

Theorem 3.25 (Theorem 7.2.9 in [7])**.** *Given $n \geqslant -1$ and $A : \mathcal{U}$, we have that*

$$A \text{ is an } n\text{-type} \;\Leftrightarrow\; \text{for all } a : A,\ \Omega^{n+1}(A, a) \text{ is contractible.}$$

These results motivate the following definition.

Definition 3.26. Let A be a type and $n \geqslant -1$. The *n-truncation* of A is the higher inductive type $\|A\|_n$ generated by

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

Lemma 3.27. *Given a type A, then $\|A\|_n$ is an n-type.*

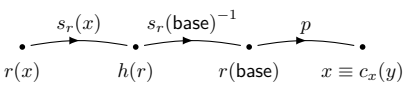
Proof. By Theorem 3.25 and Corollary 3.24, it suffices to prove that for each $x : \|A\|_n$ the type

$$\mathsf{Map}_*(\mathsf{S}^{n+1}, (\|A\|_n, x))$$

is contractible. For the center of contraction, we choose the constant function at x,

$$\begin{cases} c_x : \mathsf{S}^{n+1} \to \|A\|_n, \\ c_x(y) :\equiv x, \end{cases}$$

and the witness $\mathsf{refl}_x : c_x(\mathsf{base}) = x$. Take an arbitrary element of $\mathsf{Map}_*(\mathsf{S}^{n+1}, (\|A\|_n, x))$. By Σ-induction, we can choose it of the form (r, p) where $r : \mathsf{S}^{n+1} \to \|A\|_n$ and $p : r(\mathsf{base}) = x$. By Theorem 2.31, showing $(r, p) = (c_x, \mathsf{refl}_x)$ requires to give a path $q : r = c_x$ and to check that p transported along q gives refl_x. The equality $c_x = r$ is given by function extensionality, since for $y : \mathsf{S}^{n+1}$ we have the following concatenation:



And after the transport in the equality type, it remains to prove that

$$\left(s_r(\mathsf{base}) \cdot s_r(\mathsf{base})^{-1} \cdot p\right)^{-1} \cdot p = \mathsf{refl}_x,$$

which is true by path properties (Lemma 2.5). □

Observe that the (-1)-truncation reduces a type to a mere proposition. Therefore, the notation $\|-\| :\equiv \|-\|_{-1}$ is the one announced in Section 1.6.

The recursion principle for n-truncations is as follows. Let $A, B : \mathcal{U}$. In order to have a function $f : \|A\|_n \to B$, we need three functions

- $\phi : A \to B$,
- $\psi : \prod_{(r:\mathsf{S}^{n+1}\to\|A\|_n)} \prod_{(r':\mathsf{S}^{n+1}\to B)} B$, and
- $\theta : \prod_{(r:\mathsf{S}^{n+1}\to\|A\|_n)} \prod_{(r':\mathsf{S}^{n+1}\to B)} \prod_{(x:\mathsf{S}^{n+1})} r'(x) = \psi(r, r')$.

Then a function $f : \|A\|_n \to B$ exists such that $f(|a|_n) :\equiv \phi(a)$ for all $a : A$.

Making things above dependent gives us the induction principle of n-truncations, which goes as follows. Let $A : \mathcal{U}$ and $B : \|A\|_n \to \mathcal{U}$. In order to have a function $g : \prod_{(x:\|A\|_n)} B(x)$, we need three functions

- $\phi : \prod_{(a:A)} B(|a|_n)$,
- $\psi : \prod_{(r:\mathsf{S}^{n+1}\to\|A\|_n)} \prod_{(r':\prod_{(x:\mathsf{S}^{n+1})} B(r(x)))} B(h(r))$, and
- $\theta : \prod_{(r:\mathsf{S}^{n+1}\to\|A\|_n)} \prod_{(r':\prod_{(x:\mathsf{S}^{n+1})} B(r(x)))} \prod_{(x:\mathsf{S}^{n+1})} r'(x) =^B_{s_r(x)} \psi(r, r')$.

Then a function $g : \prod_{(x:\|A\|_n)} B(x)$ exists such that $g(|a|_n) :\equiv \phi(a)$ for all $a : A$.

The formulation of the induction principle is a bit long and not very telling. Fortunately, the next lemmas tell us that to define a function from an n-truncation to an n-type, it suffices to ignore the truncation. The universal property of truncation will then indeed reflects the result that the truncation is adjoint to the inclusion "$i : n\text{-}\mathsf{Type} \hookrightarrow \mathcal{U}$".

Lemma 3.28. *Let $A : \mathcal{U}$, $B : \|A\|_n \to \mathcal{U}$ and $g : \prod_{(a:A)} B(|a|_n)$, such that each $B(x)$ is an n-type for each $x : \|A\|_n$. Then there exists a function $f : \prod_{(x:\|A\|_n)} B(x)$ such that $f(|a|_n) :\equiv g(a)$ for all $a : A$.*

Notation 3.29. In the same context as in the last lemma, we denote the function f by $\mathsf{ext}(g)$.

Proof. The goal is to use the induction principle of n-truncations. Notice that we already have the first data needed with $\phi :\equiv g$, so let us construct the two others. Take $r : \mathsf{S}^{n+1} \to \|A\|_n$ and $r' : \prod_{(x:\mathsf{S}^{n+1})} B(r(x))$. We want to choose an element $\psi(r, r') : B(h(r))$. In order to already plan for the third data, we need to have

$$s_r(x)_*(r'(x)) = \psi(r, r')$$

for all $x : \mathsf{S}^{n+1}$. If we define

$$\begin{cases} t : \mathsf{S}^{n+1} \to B(h(r)) \\ \qquad x \mapsto s_r(x)_*(r'(x)) \end{cases},$$

it means that $\psi(r, r')$ must be equal to all $t(x)$ for $x : \mathsf{S}^{n+1}$. Since $B(h(r))$ is an n-type, Theorem 3.25 implies that $\Omega^{n+1}(B(h(r)), u)$ is contractible for all $u : B(h(r))$. Let us consider the specific basepoint $u :\equiv t(\mathsf{base})$. By Corollary 3.24, the type $\mathsf{Map}_*(\mathsf{S}^{n+1}, (B(h(r)), u))$ is contractible. For the center of contraction we choose c_u the constant function at u. Since $t(\mathsf{base}) \equiv u$, the function t belong to this pointed map space. Therefore $t = c_u$ and thanks to function extensionality we have a homotopy

$$v : \prod_{x:\mathsf{S}^{n+1}} t(x) = u.$$

We can now set $\psi(r, r') :\equiv u \equiv t(\mathsf{base})$ and the induction principle applies. □

Lemma 3.30 (Universal property of truncations)**.** *Given a type A and an n-type B, we have an equivalence*

$$(\|A\|_n \to B) \simeq (A \to B).$$

Proof. We consider the two following functions:

$$\begin{cases} (\|A\|_n \to B) \longleftrightarrow (A \to B) \\ \qquad \mathsf{ext}(g) \longleftarrow\!\shortmid g, \\ \qquad\quad f \longmapsto f \circ |-|_n. \end{cases}$$

To see that they are quasi-inverses, observe that for $g : A \to B$, $\mathsf{ext}(g) \circ |-|_n \equiv g$ by the computation rule given in Lemma 3.28. Moreover, for $f : \|A\|_n \to B$ and $a : A$, we have $\mathsf{ext}(f \circ |-|_n)(|a|_n) \equiv f(|a|_n)$ by the same computation rule. Hence, we have

$$a \mapsto \mathsf{refl}_{f(|a|_n)} : \prod_{a:A} \mathsf{ext}(f \circ |-|_n)(|a|_n) = f(|a|_n).$$

Since B is an n-type, equality types inside B are $(n-1)$-types, so in particular n-types. Therefore, we can use Lemma 3.28 and have that

$$\prod_{x:\|A\|_n} \mathsf{ext}(f \circ |-|_n)(x) = f(x).$$

□

Definition 3.31. Given $f : A \to B$, the *n-truncation* of f is defined as

$$\|f\|_n :\equiv \mathsf{ext}(|-|_n \circ f) : \|A\|_n \to \|B\|_n.$$

Schematically, the following diagram commute, since for all $a : A$ we have

$$\|f\|_n(|a|_n) \equiv \mathsf{ext}(|-|_n \circ f)(|a|_n) \equiv (|-|_n \circ f)(a) \equiv |f(a)|_n.$$

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ {\scriptstyle |-|_n}\downarrow & \swarrow & \downarrow{\scriptstyle |-|_n} \\ \|A\|_n & \xrightarrow[\|f\|_n]{} & \|B\|_n \end{array}$$

Note that the truncation is now functorial, since given $A \xrightarrow{f} B \xrightarrow{g} C$ we have

$$\begin{aligned} \|g \circ f\|_n &= \|g\|_n \circ \|f\|_n, \text{ and} \\ \|\mathsf{id}_A\|_n &= \mathsf{id}_{\|A\|_n}, \end{aligned}$$

Indeed, by Lemma 3.28 it suffices to check those properties for projected elements $|a|_n$ where $a : A$, in which case everything is trivial.

Lemma 3.32. *Given an equivalence $f : A \xrightarrow{\simeq} B$, then $\|f\|_n : \|A\|_n \xrightarrow{\simeq} \|B\|_n$.*

Proof. By hypothesis, we have a quasi-inverse $g : B \to A$. But then,

$$\|g\|_n \circ \|f\|_n = \|g \circ f\|_n = \|\mathsf{id}_A\|_n = \mathsf{id}_{\|A\|_n},$$

and

$$\|f\|_n \circ \|g\|_n = \|f \circ g\|_n = \|\mathsf{id}_B\|_n = \mathsf{id}_{\|B\|_n}.$$

Therefore $\|f\|_n$ admits $\|g\|_n$ as a quasi-inverse and is an equivalence as desired. □

Theorem 3.33. *Given $A : \mathcal{U}$, $x, y : A$ and $n \geqslant -2$, we have an equivalence*

$$\|x =_A y\|_n \simeq \left(|x|_{n+1} =_{\|A\|_{n+1}} |y|_{n+1}\right).$$

Proof. In one direction, we have by truncation induction the map

$$\begin{cases} f : \|x =_A y\|_n \to (|x|_{n+1} =_{\|A\|_{n+1}} |y|_{n+1}) \\ f(|p|_n) :\equiv \mathsf{ap}_{|-|_n}(p). \end{cases}$$

But in the other direction, there is no way to induct on an equality $|x|_{n+1} =_{\|A\|_{n+1}} |y|_{n+1}$. Hence, it is an application of a method named encode-decode. The idea is to generalise the goal in order to perform induction. Here, what we know instead is how to induct on equalities $u =_{\|A\|_{n+1}} v$, for $u, v : \|A\|_{n+1}$. But then, how do we describe the left hand type $\|x =_A y\|_n$ in terms of u and v ?

We proceed as follows. First, by induction let

$$\begin{cases} P : \|A\|_{n+1} \to \|A\|_{n+1} \to n\text{-}\mathsf{Type}, \\ P(|x|_{n+1}, |y|_{n+1}) :\equiv \|x =_A y\|_n. \end{cases}$$

The induction is valid because n-Type is indeed an $(n+1)$-type, as seen in (Theorem 3.15). The two quasi-inverse maps are defined by induction as follows:

$$\begin{cases} \mathsf{decode} : \prod_{u,v:\|A\|_{n+1}} P(u,v) \to (u =_{\|A\|_{n+1}} v), \\ \mathsf{decode}(|x|_{n+1}, |y|_{n+1}, |p|_n) :\equiv f(|p|_n) \equiv \mathsf{ap}_{|-|_n}(p). \end{cases}$$

$$\begin{cases} \mathsf{encode} : \prod_{u,v:\|A\|_{n+1}} (u =_{\|A\|_{n+1}} v) \to P(u,v), \\ \mathsf{encode}(u, v, q) :\equiv \mathsf{transport}^{P(u,-)}(q, r(u)), \end{cases}$$

where

$$\begin{cases} r : \prod_{u:\|A\|_{n+1}} P(u,u), \\ r(|x|_{n+1}) :\equiv |\mathsf{refl}_x|_n. \end{cases}$$

Let us prove that both compositions of encode and decode give the identity. For notation simplicity, the parameters $u, v : \|A\|_{n+1}$ are omitted as they will always be clear from the context.

In one direction, if we have $q : u =_{\|A\|_{n+1}} v$, we want to prove $\mathsf{decode}(\mathsf{encode}(q)) = q$. By a path induction on q, it suffices to prove that $\mathsf{decode}(r(u)) = \mathsf{refl}_u$. Since both sides of the equality are n-types, it is an $(n-1)$-type so in particular an $(n+1)$-type. By truncation induction, it suffices to consider when $u \equiv |x|_{n+1}$ for some $x : A$. But then, both sides are judgmentally equal to $\mathsf{refl}_{|x|_{n+1}}$.

In the other direction, given $s : P(u,v)$ we want to show that $\mathsf{encode}(\mathsf{decode}(s)) = s$. This an $(n-1)$-type and thus an $(n+1)$-type, as before. Thus, by truncation induction, it suffices to consider when $u \equiv |x|_{n+1}$, $v \equiv |y|_{n+1}$ and $s \equiv |p|_n$ for some $x, y : A$ and $p : x = y$. And last, by a path induction on p, we indeed have

$$\mathsf{encode}(\mathsf{decode}(|\mathsf{refl}_x|_n)) = |\mathsf{refl}_x|_n,$$

as both sides are judgmentally equal by definition. The theorem is then proved as it corresponds to the case where u is $|x|_{n+1}$ and v is $|y|_{n+1}$. □

Lemma 3.34. *Given $A : \mathcal{U}$ and $n, k : \mathbb{N}$ with $k \leqslant n$. Then $\|\|A\|_n\|_k = \|A\|_k$.*

Proof. By univalence, it suffices to show an equivalence. For that, we define by induction two maps:

$$\begin{cases} f : \|\|A\|_n\|_k \to \|A\|_k, \\ f(||x|_n|_k) :\equiv |x|_k, \end{cases} \qquad \begin{cases} g : \|A\|_k \to \|\|A\|_n\|_k, \\ g(|x|_k) :\equiv ||x|_n|_k. \end{cases}$$

The induction for f is valid, since $\|A\|_k$ is a k-type and thus also an n-type. To see that f and g are quasi-inverses, it suffices to check for projected elements, in which case it is trivial by definition. □

3.4 Connected types and maps

Definition 3.35. Given a function $f : A \to B$, its *(homotopy) fiber* over a point $b : B$ is

$$\mathsf{fib}_f(b) :\equiv \sum_{a:A} (f(a) = b).$$

Definition 3.36. Let $A, B : \mathcal{U}$ be types and $f : A \to B$ be a function.

- We say that A is *n-connected* if $\|A\|_n$ is contractible.
- We say that f is *n-connected* if all its fibers are n-connected. In other words,

$$\mathsf{isConn}_n(f) :\equiv \prod_{b:B} \mathsf{isContr}(\|\mathsf{fib}_f(b)\|_n).$$

Remark. Observe that a type A is n-connected if and only if the unique function $A \to \mathbf{1}$ is n-connected.

$$\begin{aligned}
\mathsf{isConn}_n(a \mapsto \star : A \to \mathbf{1}) &\equiv \prod_{x:1} \mathsf{isContr}\|\mathsf{fib}_{a\mapsto\star}(x)\|_n \\
&\simeq \mathsf{isContr}\|\mathsf{fib}_{a\mapsto\star}(\star)\|_n && \text{By the uniqueness principle of } \mathbf{1}. \\
&\equiv \mathsf{isContr}\left\|\sum_{a:A} (\star = \star)\right\|_n \\
&\simeq \mathsf{isContr}\|A\|_n. && \text{By Lemma 3.19.}
\end{aligned}$$

Lemma 3.37. *Let $f : A \to B$ and $n \geqslant -2$. For a family $P : B \to \mathcal{U}$, consider the function*

$$(- \circ f) : \left(\prod_{b:B} P(b)\right) \to \left(\prod_{a:A} P(f(a))\right)$$

Then, the following are equivalent:

i) f is n-connected.

ii) For every $P : B \to n\text{-}\mathsf{Type}$, $- \circ f$ is an equivalence.

iii) For every $P : B \to n\text{-}\mathsf{Type}$, $- \circ f$ admits a section.

Proof. ($i \Rightarrow ii$) Suppose f is n-connected and take $P : B \to n\text{-}\mathsf{Type}$. Observe that

$$\begin{aligned}
\prod_{b:B} P(b) &\simeq \prod_{b:B} (\|\mathsf{fib}_f(b)\|_n \to P(b)) && \text{(since } \|\mathsf{fib}_f(b)\|_n \text{ is contractible)} \\
&\simeq \prod_{b:B} (\mathsf{fib}_f(b) \to P(b)) && \text{(since } P(b) \text{ is an } n\text{-type)} \\
&\simeq \prod_{(b:B)} \prod_{(a:A)} \prod_{(p:fa=b)} P(b) && \text{(universal property of } \Sigma\text{-types)} \\
&\simeq \prod_{a:A} P(fa) && \text{(universal property of } =)
\end{aligned}$$

which is explicitly given by

$$\begin{aligned}
s &\rightsquigarrow \lambda b.\, (\star \mapsto s(b)) \\
&\rightsquigarrow \lambda b.\, (x \mapsto s(b)) \\
&\rightsquigarrow \lambda b.\, \lambda a.\, \lambda p.\, s(b) \\
&\rightsquigarrow (a \mapsto s(f(a))).
\end{aligned}$$

In other words, $s \mapsto s \circ f$ is an equivalence as desired.

($ii \Rightarrow iii$) This is immediate as an equivalence is in particular a retraction.

($iii \Rightarrow i$) To show this one, let us make use of a specific family $P : B \to n\text{-}\mathsf{Type}$, defined as

$$P(b) :\equiv \|\mathsf{fib}_f(b)\|_n.$$

Thus, to show that f is n-connected, it suffices to show the contractibility of each type $P(b)$ for $b : B$. To exhibit centers of contraction, recall that by hypothesis there is a section to $(- \circ f)$, say

$$g : \left(\prod_{a:A} P(fa)\right) \to \left(\prod_{b:B} P(b)\right).$$

Hence, for

$$\begin{cases} h : \prod_{a:A} P(fa), \\ h(a) :\equiv |(a, \mathsf{refl}_{fa})|_n, \end{cases}$$

we have that $h = g(h) \circ f$. Let us call $c :\equiv g(h) : \prod_{(b:B)} P(b)$, which will give the centers of contraction. Observe that for all $a : A$ we have

$$c(f(a)) = h(a) \equiv \left|(a, \mathsf{refl}_{f(a)})\right|_n.$$

What is left to be proven is that for $b : B$, each element of $P(b)$ is equal to the center $c(b)$, i.e.,

$$\prod_{(b:B)} \prod_{(z:\|\mathsf{fib}_f(b)\|_n)} c(b) = z.$$

Since this equality takes place in an n-type, it is also in particular an n-type. Hence, by truncation induction (Lemma 3.28) and Σ-induction, it suffices to inhabit

$$\prod_{(b:B)} \prod_{(a:A)} \prod_{(p:fa=b)} c(b) = |(a, p)|_n.$$

Be reordering variables and path induction on p, it suffices to show

$$\prod_{a:A} c(f(a)) = \left|(a, \mathsf{refl}_{f(a)})\right|_n,$$

which has already been seen above. □

The following result is stated without proof as it requires some intermediate results that are of no use for the goal of the document.

Lemma 3.38. *Let A be a type, $a_0 : \mathbf{1} \to A$ be a basepoint and $n \geqslant -1$. Then, we have*

$$A \text{ is } n\text{-connected} \Leftrightarrow \text{the function } a_0 \text{ is } (n-1)\text{-connected.}$$

Definition 3.39. Given $B, C : A \to \mathcal{U}$, a function $f : \prod_{(x:A)}(B(x) \to C(x))$ is called a *fiberwise map*. And such an f induces another function, which we define by induction:

$$\begin{aligned} \mathsf{tot}(f) : \left(\sum_{x:A} B(x)\right) &\to \left(\sum_{x:A} C(x)\right) \\ (a, b) &\mapsto (a, f(a, b)). \end{aligned}$$

The following theorem is the HoTT version of Puppe's theorem, that one can find for instance in [2].

Lemma 3.40. *Given a fiberwise map $f : \prod_{(x:A)}(B(x) \to C(x))$, $a : A$ and $c : C(a)$. Then, we have an equivalence*

$$\mathsf{fib\_total\_map} : \mathsf{fib}_{fa}(c) \xrightarrow{\simeq} \mathsf{fib}_{\mathsf{tot}(f)}(a, c).$$

Proof. It follows from the following sequence of equivalences.

$$\begin{aligned}
\mathsf{fib}_{\mathsf{tot}(f)}(a,c) &\equiv \sum_{z:\sum_{(x:A)} B(x)} (\mathsf{pr}_1\, z, f(\mathsf{pr}_1\, z, \mathsf{pr}_2\, z)) = (a,c) \\
&\simeq \sum_{(x:A)} \sum_{(b:B(x))} (x, f(x,b)) = (a,c) && \text{(by Lemma 3.22)} \\
&\simeq \sum_{(x:A)} \sum_{(b:B(x))} \sum_{(p:x=a)} p_*(f(x,b)) = c && \text{(by Theorem 2.31)} \\
&\simeq \sum_{(x:A)} \sum_{(p:x=a)} \sum_{(b:B(x))} p_*(f(x,b)) = c \\
&\simeq \sum_{(w:\sum_{(x:A)} x=a)} \sum_{(b:B(\mathsf{pr}_1\, w))} (\mathsf{pr}_2\, w)_*(f(\mathsf{pr}_1\, w, b)) = c && \text{(by Lemma 3.22)} \\
&\simeq \sum_{b:B(a)} \mathsf{refl}_{a*}(f(a,b)) = c && \text{(by Lemma 3.19 and Lemma 3.18)} \\
&\equiv \mathsf{fib}_{fa}(c).
\end{aligned}$$

□

Corollary 3.41. *Let $B, C : A \to \mathcal{U}$, $f : \prod_{(x:A)}(B(x) \to C(x))$ and $n \geqslant -2$. Then,*

$$\mathsf{tot}(f) \text{ is } n\text{-connected} \Leftrightarrow \text{each } f(a) \text{ is } n\text{-connected, for any } a : A$$

Proof. It immediately follows from Lemmas 3.32 and 3.40. □

Lemma 3.42. *Let $f : A \to B$ be n-connected. Then, $\|A\|_n \simeq \|B\|_n$.*

Proof. By hypothesis, there is a witness

$$c : \mathsf{isConn_n}(f) \equiv \prod_{b:B} \left(\mathsf{isContr}\|\mathsf{fib}_f(b)\|_n\right).$$

It induces a function that specifies the centers of contraction:

$$\begin{cases} d : \prod_{b:B} \|\mathsf{fib}_f(b)\|_n, \\ d(b) :\equiv \mathsf{pr}_1\, c(b). \end{cases}$$

The goal is to exhibit two quasi-inverse maps between $\|A\|_n$ and $\|B\|_n$. The first one is

$$\|f\|_n : \|A\|_n \to \|B\|_n.$$

The second one is defined by truncation induction (Lemma 3.28) as

$$\begin{cases} h : \|B\|_n \to \|A\|_n, \\ h(|b|_n) :\equiv \|\mathsf{pr}_1\|_n(d(b)). \end{cases}$$

To see that they are quasi-inverses, the equalities to be proven are in n-types and thus are n-types too. So by truncation induction (Lemma 3.28), it suffices to check for projected elements.

In one direction, take $a : A$ and observe that

$$\begin{aligned}
h \circ \|f\|_n(|a|_n) &\equiv h(|f(a)|_n) \\
&\equiv \|\mathsf{pr}_1\|_n(d(f(a))) \\
&= \|\mathsf{pr}_1\|_n(\left|(a, \mathsf{refl}_{f(a)})\right|_n) && \text{(since } d(f(a)) : \|\mathsf{fib}_f(f(a))\|_n \text{ which is contractible)} \\
&\equiv \left|\mathsf{pr}_1(a, \mathsf{refl}_{f(a)})\right|_n \\
&= |a|_n.
\end{aligned}$$

In the other direction, take $b : B$. Since we have

$$\|f\|_n \circ h\ (|b|_n) \equiv \|f\|_n \circ \|\mathsf{pr}_1\|_n(d(b)),$$

the goal is now to prove that

$$\|f\|_n \circ \|\mathsf{pr}_1\|_n(d(b)) = |b|_n.$$

Observe that $d(b) : \|\mathsf{fib}_f(b)\|_n \equiv \left\|\sum_{(a:A)} fa = b\right\|_n$. Therefore, what we want is essentially in the second component of $d(b)$, except that truncations are preventing us for manipulating them easily. The idea is to generalise the claim to an arbitrary element z:

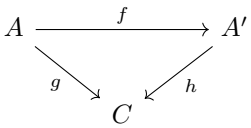
$$\prod_{z:\|\mathsf{fib}_f(b)\|_n} \|f\|_n \circ \|\mathsf{pr}_1\|_n(z) = |b|_n.$$

By truncation induction (Lemma 3.28), it is sufficient to assume that $z \equiv |(a, p : fa = b)|_n$, but then

$$\begin{aligned} \|f\|_n \circ \|\mathsf{pr}_1\|_n(|(a,p)|_n) &\equiv \|f\|_n(|a|_n) \\ &\equiv |f(a)|_n \\ &= |b|_n \qquad (\text{by } \mathsf{ap}_{|-|_n}(p)) \end{aligned}$$

□

Lemma 3.43. *Let f, g, h be functions as on the diagram.*



If f is n-connected and the triangle commute, with some $\gamma : hf \sim g$, then there is a family of equivalences:

$$\mathsf{fib\_\_trunc\_\_eqv} : \prod_{b:B} \left(\|\mathsf{fib}_g(b)\|_n \simeq \|\mathsf{fib}_h(b)\|_n\right).$$

Proof. For each $b : B$, we define a function

$$\begin{cases} \phi_b : \mathsf{fib}_g(b) \to \mathsf{fib}_h(b) \\ \phi_b(a, p : ga = b) \mapsto (fa, \gamma(a) \cdot p : hfa = b). \end{cases}$$

The lemma will be proved thanks to Lemma 3.42, provided that all ϕ_b are n-connected. Take $b : B$, $a' : A'$ and $q : ha' = b$ and observe that

$$\begin{aligned} \mathsf{fib}_{\phi_b}(a', q) &\equiv \sum_{(a,p):\sum_{(a:A)} ga=b} (fa, \gamma(a) \cdot p) = (a', q) \\ &\overset{\Sigma \text{ associativity}}{\simeq} \sum_{(a:A)} \sum_{(p:ga=b)} (fa, \gamma(a) \cdot p) = (a', q) \\ &\overset{\text{equality in } \Sigma}{\simeq} \sum_{(a:A)} \sum_{(p:ga=b)} \sum_{(r:fa=a')} \mathsf{transport}^{h(-)=b}(r, \gamma(a) \cdot p) = q \\ &\overset{\text{transport in } =}{\simeq} \sum_{(a:A)} \sum_{(p:ga=b)} \sum_{(r:fa=a')} \mathsf{ap}_h(r)^{-1} \cdot \gamma(a) \cdot p = q \\ &\simeq \sum_{(a:A)} \sum_{(r:fa=a')} \underbrace{\sum_{p:ga=b} p = \gamma(a)^{-1} \cdot \mathsf{ap}_h(r) \cdot q}_{\text{contractible by Lemma 3.18 for all } r} \\ &\overset{\text{Lemma 3.19}}{\simeq} \sum_{a:A} fa = a' \\ &\equiv \mathsf{fib}_f(a'). \end{aligned}$$

Therefore, for $b : B$ the function ϕ_b is n-connected because all its fibers are, and the proof is finished. □

3.5 Pushouts & Pullbacks

Definition 3.44. A span is a 5-tuple $\mathscr{D} :\equiv (A, B, C, f, g)$ with $f : C \to A$ and $g : C \to B$.

$$\mathscr{D} \equiv \quad \begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & & \\ A & & \end{array}$$

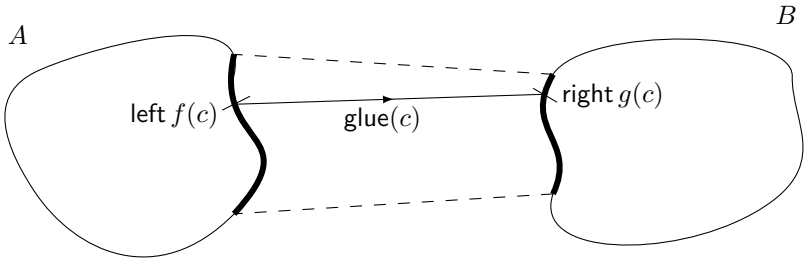
Definition 3.45. Given a span $\mathscr{D} :\equiv (A, B, C, f, g)$, its *(homotopy) pushout* is the higher inductive type $A \sqcup^C B$ generated by

- a function $\mathsf{left} : A \to A \sqcup^C B$,
- a function $\mathsf{right} : B \to A \sqcup^C B$, and
- for each $c : C$, a path $\mathsf{glue}(c) : \mathsf{left}\, f(c) = \mathsf{right}\, g(c)$.

$$\begin{array}{ccc} C & \xrightarrow{g} & B \\ {\scriptstyle f}\downarrow & \swarrow_{\mathsf{glue}} & \downarrow{\scriptstyle \mathsf{right}} \\ A & \xrightarrow[\mathsf{left}]{} & A \sqcup^C B \end{array}$$

The pushout is, as in homotopy theory, the idea of a disjoint union of A and B where we glue together the images of C in both spaces. However, the gluing is done with equalities, i.e., paths. Therefore it is not a simple pushout, but a homotopy pushout. Visualising with topological spaces, it corresponds to having a cylinder between the parts to be glued, which allows to glue corresponding points with paths in order for the pushout to be homotopy invariant.

This phenomenon is the same as with higher categories, where the strict construction also ends up being homotopy invariant.

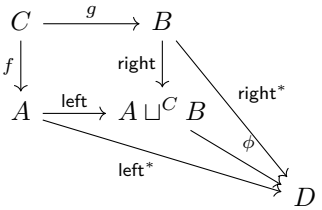


The recursion principle is that given $D : \mathcal{U}$ and functions

- $\mathsf{left}^* : A \to D$,
- $\mathsf{right}^* : B \to D$, and
- $\mathsf{glue}^* : \prod_{(c:C)} \mathsf{left}^*\, f(c) = \mathsf{right}^*\, g(c)$,

then there exists $\phi : A \sqcup^C B \to D$ such that

$$\begin{aligned} \phi\ \mathsf{left}(a) &:\equiv \mathsf{left}^*\, a && \text{for each } a : A, \\ \phi\ \mathsf{right}(b) &:\equiv \mathsf{right}^*\, b && \text{for each } b : B, \text{ and} \\ \mathsf{ap}_\phi(\mathsf{glue}(c)) &:= \mathsf{glue}^*\, c && \text{for each } c : C. \end{aligned}$$



The induction principle is that given $D : A \sqcup^C B \to \mathcal{U}$ and functions

- $\mathsf{left}^* : \prod_{(a:A)} D(\mathsf{left}(a))$,
- $\mathsf{right}^* : \prod_{(b:B)} D(\mathsf{right}(b))$, and
- $\mathsf{glue}^* : \prod_{(c:C)} \mathsf{left}^*\, f(c) =^D_{\mathsf{glue}(c)} \mathsf{right}^*\, g(c)$,

then there exists $\phi : \prod_{(p:A\sqcup^C B)} D(p)$ such that

$$\begin{aligned} \phi\ \mathsf{left}(a) &:\equiv \mathsf{left}^*\, a && \text{for each } a : A, \\ \phi\ \mathsf{right}(b) &:\equiv \mathsf{right}^*\, b && \text{for each } b : B, \text{ and} \\ \mathsf{apd}_\phi(\mathsf{glue}(c)) &\coloneqq \mathsf{glue}^*\, c && \text{for each } c : C. \end{aligned}$$

An alternative viewpoint of the pushout is to replace the space C and the two functions f and g with a predicate $P : A \to B \to \mathcal{U}$. Right now, two points $a : A$ and $b : B$ are glued when they are related by an element $c : C$, i.e., we have $a \equiv f(c)$ and $b \equiv g(c)$. This can change by choosing to glue a and b together when they are related by the predicate P, i.e., if there is some element $p : P(a, b)$. Quite logically, the function glue shall then be a bit different to translate this change.

Definition 3.46. A *span with predicate* is a 3-tuple $\mathscr{D} :\equiv (A, B, P)$ with $P : A \to B \to \mathcal{U}$. The *(homotopy) pushout with predicate* of $\mathscr{D}$ is the higher inductive type $A \sqcup^P B$ generated by three functions

- $\mathsf{left} : A \to A \sqcup^P B$,
- $\mathsf{right} : B \to A \sqcup^P B$, and
- $\mathsf{glue} : \prod_{(a:A)} \prod_{(b:B)} P(a, b) \to (\mathsf{left}\ \ a = \mathsf{right}\ \ b)$.

The recursion and induction principle are easy to adapt from the first case.

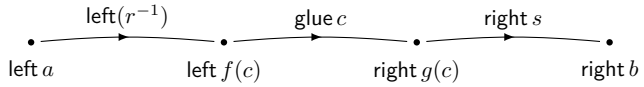
Remark. Observe that having a pushout or a pushout with predicate is equivalent.

- First, imagine having $\mathscr{D} :\equiv (A, B, C, f, g)$ and $A \sqcup^C B$. Then, we can set

$$\begin{aligned} &P : A \to B \to \mathcal{U} \\ &P(a, b) :\equiv \sum_{c:C} \left((fc = a) \times (gc = b)\right), \end{aligned}$$

  and by induction

$$\begin{aligned} &\mathsf{glue}^P : \prod_{(a:A)} \prod_{(b:B)} P(a, b) \to (\mathsf{left}\ \ a = \mathsf{right}\ \ b) \\ &\mathsf{glue}^P(a, b, (c, r, s)) :\equiv \mathsf{left}(r^{-1}) \centerdot \mathsf{glue}\, c \centerdot \mathsf{right}\ \ s. \end{aligned} \tag{1}$$



- Second, imagine having $\mathscr{D} :\equiv (A, B, P)$ and $A \sqcup^P B$. Then, we can set

$$\begin{aligned} C &:\equiv \sum_{(a,b):A\times B} P(a, b), \\ f &:\equiv \mathsf{pr}_1\, \mathsf{pr}_1 : C \to A, \text{ and} \\ g &:\equiv \mathsf{pr}_2\, \mathsf{pr}_1 : C \to B. \end{aligned}$$

  The slight abuse of notation with the dependent sum indexed directly over pairs $(a, b) : A \times B$ is voluntary. This is for better readability as it avoids having to write $z : A\times B$ and then $P(\mathsf{pr}_1\, z, \mathsf{pr}_2\, z)$, and it also announces directly the notations used for the projections.

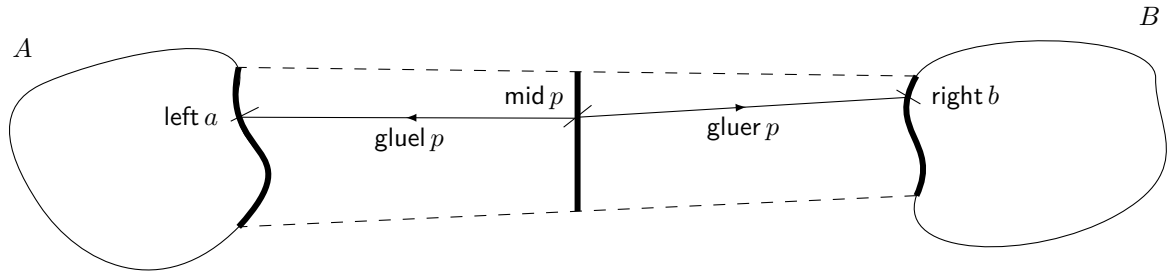
Note that the correspondences we just gave do fit well together. For instance if have (A, B, C, f, g) and define the predicate P as above, then $\sum_{((a,b):A\times B)} P(a,b)$ is equivalent to C quite directly thanks to Lemma 3.18. Therefore, there is no problem in going back and forth between looking at the pushout with a predicate or not.

Now yet another point of view, which will be needed later, is the notion of "fat" pushout, in opposition with our precedent pushout which is sometimes called "thin". Let us come back to the visualisation of our homotopy pushout with our two topological spaces linked with a cylinder. We now construct instead two cylinders glued one behind the other. This construction is sometimes used in practice to define with more ease the connections of the gluing.

Definition 3.47. Given a span with predicate $\mathscr{D} :\equiv (A, B, P)$, the *(homotopy) fat pushout (with predicate)* is the higher inductive type $A \sqcup_f^P B$ generated by the functions

- $\mathsf{left} : A \to A \sqcup_f^P B$,
- $\mathsf{right} : B \to A \sqcup_f^P B$,
- $\mathsf{mid} : \prod_{(a:A)} \prod_{(b:B)} P(a,b) \to A \sqcup_f^P B$,
- $\mathsf{gluel} : \prod_{(a:A)} \prod_{(b:B)} \prod_{(p:P(a,b))} (\mathsf{mid}\, p = \mathsf{left}\, a)$, and
- $\mathsf{gluer} : \prod_{(a:A)} \prod_{(b:B)} \prod_{(p:P(a,b))} (\mathsf{mid}\, p = \mathsf{right}\, b)$.

When dealing with a pushout with predicate we sometimes simplify notations for readability purposes. This is done by omitting some parameters and writing for instance $\mathsf{glue}\, p$ instead of $\mathsf{glue}(a, b, p)$ or similarly $\mathsf{gluel}\, p$ instead of $\mathsf{gluel}(a, b, p)$ when we have $a : A$, $b : B$, and $p : P(a, b)$, since it can be clear what the missing parameters are when knowing that $p : P(a, b)$.



Remark. Here again, observe that this vision of the pushout is equivalent to the others.

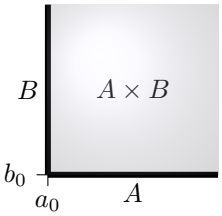
- First, suppose we have a fat pushout with predicate $A \sqcup_f^P B$. To come back to a thin one, we can define

$$\mathsf{glue}^P : \prod_{(a:A)} \prod_{(b:B)} P(a,b) \to (\mathsf{left}\, a = \mathsf{right}\, b),$$
$$\mathsf{glue}^P(a, b, p) :\equiv (\mathsf{gluel}\, p)^{-1} \cdot \mathsf{gluer}\, p.$$

- And the other way around, suppose we have a (thin) pushout with predicate $A \sqcup^P B$. Then we can set the middle of the cylinders to be the image of the right injection, with gluel having the role of glue, and gluer being trivial.

$$\begin{cases} \mathsf{mid} : \prod_{(a:A)} \prod_{(b:B)} P(a,b) \to A \sqcup_f^P B, \\ \mathsf{mid}(a,b,p) :\equiv \mathsf{right}\, a, \end{cases}$$
$$\begin{cases} \mathsf{gluel} : \prod_{(a:A)} \prod_{(b:B)} \prod_{(p:P(a,b))} (\mathsf{mid}\, p = \mathsf{left}\, a), \\ \mathsf{gluel}(a,b,p) :\equiv (\mathsf{glue}^P(a,b,p))^{-1}, \end{cases}$$
$$\begin{cases} \mathsf{gluer} : \prod_{(a:A)} \prod_{(b:B)} \prod_{(p:P(a,b))} (\mathsf{mid}\, p = \mathsf{right}\, b), \\ \mathsf{gluer}(a,b,p) :\equiv \mathsf{refl}_{\mathsf{right}\, b}. \end{cases}$$

Example 3.48. Given two pointed types (A, a_0) and (B, b_0), their *wedge*, denoted $A \vee B$, is the pushout of $A \xleftarrow{a_0} \mathbf{1} \xrightarrow{b_0} B$. The idea of the wedge is to have almost the disjoint union of A and B, but with two points, one in each type, glued together. Informally, if A and B are illustrated with segments then the product $A \times B$ corresponds to the surface created, while $A \vee B$ corresponds to the "axis system", with the origin being the point shared between both axes.



Therefore, it would make sense to have an inclusion from the wedge to the product. Indeed, such a function exists,

$$\mathsf{wtp} : A \vee B \to A \times B,$$

defined by wedge induction as follows:

$$\begin{cases} \mathsf{wtp}(\mathsf{left}\, a) :\equiv (a, b_0), \\ \mathsf{wtp}(\mathsf{right}\, b) :\equiv (a_0, b), \\ \mathsf{ap}_{\mathsf{wtp}}(\mathsf{glue}\, \star) \coloneqq \mathsf{refl}_{(a_0,b_0)}, \end{cases}$$

Definition 3.49. Dually to the notion of span and pushout, we have notion of *cospan* and of *pullback*. A *cospan* is a 5-tuple (A, B, D, h, k), where $h : A \to D$ and $k : B \to D$.

$$\begin{array}{ccc} & & B \\ & & \downarrow{\scriptstyle k} \\ A & \xrightarrow[h]{} & D \end{array}$$

A *pullback* of this cospan, denoted $A \times^D B$, can be defined analogously to the way we defined a pushout with a span, but this time we choose instead to give an explicit construction. We let

$$A \times^D B :\equiv \sum_{(a:A)} \sum_{(b:B)} (h\, a = k\, b).$$

Observe that thanks to the projections, we have indeed a commutative square.

$$\begin{array}{ccc} A \times^D B & \xrightarrow{\mathsf{pr}_1\, \mathsf{pr}_2} & B \\ {\scriptstyle \mathsf{pr}_1}\downarrow & \swarrow{\scriptstyle \mathsf{pr}_2\, \mathsf{pr}_2} & \downarrow{\scriptstyle k} \\ A & \xrightarrow[h]{} & D \end{array}$$

3.6 Wedge connectivity lemma

A lemma which is essential for the Blakers-Massey theorem is the wedge connectivity lemma. It is a powerful statement on the connectivity of the map wtp defined in Example 3.48. First, we need the following lemma, which is a generalisation of Lemma 3.37.

Lemma 3.50 (Lemma 8.6.1 in [7])**.** *Given an n-connected function $f : A \to B$, a family $P : B \to k\text{-}\mathsf{Type}$ where $k \geqslant n$, then the fibers of the induced function*

$$(- \circ f) : \left(\prod_{b:B} P(b)\right) \to \left(\prod_{a:A} P(f(a))\right)$$

are all $(k - n - 2)$-types.

The proof is very similar to the one of Lemma 3.37. However, it requires small intermediate results that do not interest us in this document, so the proof will be omitted. The proof can be found in [7].

The second lemma needed contains in fact already the essence of the wedge connectivity lemma. The idea is that to define a function $f : \prod_{(a:A)} \prod_{(b:B)} P(a, b)$ where all types $P(a, b)$ are $(i + j)$-connected, it suffices to define the function in the case where the first parameter is fixed at some $a_0 : A$, then in the case where the second parameter is fixed at some $b_0 : B$ and then to check that everything do coincide nicely when considering the pair (a_0, b_0). And indeed, this procedure recalls the induction required to define a function from the wedge $A \vee B$.

Lemma 3.51. *Let (A, a_0) and (B, b_0) be i- and j- connected pointed types, respectively, where $i, j \geqslant 0$, and let $P : A \to B \to (i + j)\text{-}\mathsf{Type}$. Given any functions*

$$f_{a_0} : \prod_{b:B} P(a_0, b) \text{ and}$$
$$f_{b_0} : \prod_{a:A} P(a, b_0),$$

with $p : f_{b_0}(a_0) = f_{a_0}(b_0)$, then there exists a function $f : \prod_{(a:A)} \prod_{(b:B)} P(a, b)$ and two homotopies

$$\alpha : \prod_{a:A} f(a, b_0) = f_{b_0}(a)$$
$$\beta : \prod_{b:B} f(a_0, b) = f_{a_0}(b)$$

such that $p = \alpha(a_0)^{-1} \cdot \beta(b_0)$.

Proof. First, by Lemma 3.38, we have that $b_0 : \mathbf{1} \to B$ is $(j - 1)$-connected. Moreover, each $P(a, b)$ is $(i + j)$-connected. Therefore, by Lemma 3.50 the function

$$(- \circ b_0) : \left(\prod_{b:B} P(a, b)\right) \to P(a, b_0)$$

has its fibers which are $(i - 1)$-truncated.

By Lemma 3.38 again, we have that $a_0 : \mathbf{1} \to A$ is $(i - 1)$-connected. Hence, by Lemma 3.37, given a family $Q : A \to (i - 1)\text{-}\mathsf{Type}$, the function

$$(- \circ a_0) : \left(\prod_{a:A} Q(a)\right) \to Q(a_0)$$

admits a section, say g, which comes with an homotopy $\epsilon : g(-)(a_0) \sim \mathsf{id}_{Q(a_0)}$. We apply this fact with the following specific family:

$$\begin{cases} Q : A \to (i - 1)\text{-}\mathsf{Type}, \\ \\ Q(a) :\equiv \mathsf{fib}_{(- \circ b_0)}(f_{b_0}(a)) \equiv \left(\displaystyle\sum_{k : \prod_{(b:B)} P(a,b)} f_{b_0}(a) = k(b_0)\right). \end{cases}$$

Since we noticed before that the fibers of $(- \circ b_0)$ are $(i-1)$-truncated, the family Q is well-defined. Observe that we have an element $(f_{a_0}, p) : Q(a_0)$. Therefore, if we let $l :\equiv g((f_{a_0}, p)) : \prod_{(a:A)} Q(a)$, we have that $\epsilon((f_{a_0}, p)) : l(a_0) = (f_{a_0}, p)$. We define the function

$$\begin{cases} f : \displaystyle\prod_{(a:A)} \prod_{(b:B)} P(a,b), \\ f(a,b) :\equiv (\mathsf{pr}_1(l(a)))(b). \end{cases}$$

We still have to exhibit the homotopies, which is very computational. For $a : A$ and $b : B$, let

$$\begin{cases} \alpha(a) : \Big(f(a,b_0) = f_{b_0}(a)\Big), \\ \alpha(a) :\equiv \mathsf{pr}_2(l(a_0))^{-1} \end{cases}$$

and

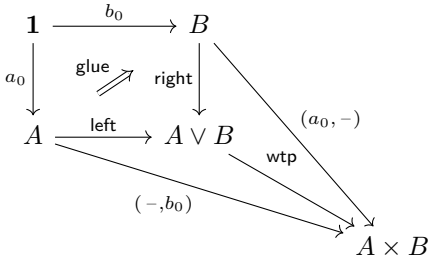
$$\begin{cases} \beta(b) : \Big(f(a_0,b) = f_{a_0}(b)\Big) \equiv \Big(\mathsf{pr}_1(l(a_0))(b) = f_{a_0}(b)\Big), \\ \alpha(a) :\equiv \mathsf{ap}_{\mathsf{pr}_1(-)(b_0)}(\epsilon(f_{a_0}, p)), \end{cases}$$

Finally, the equality $p = \alpha(a_0)^{-1} \cdot \beta(b_0)$ holds as it suffices to explicit the definition of ϵ which comes from Lemma 3.37 and to develop everything. □

Lemma 3.52 (Wedge Connectivity Lemma)**.** *Let (A, a_0) and (B, b_0) be i- and j- connected pointed types, respectively, where $i, j \geqslant 0$. Then the map*

$$\mathsf{wtp} : A \vee B \to A \times B$$

as defined in Example 3.48 is $(i+j)$-connected.



Proof. To prove that wtp is $(i+j)$-connected, we use the characterisation given in Lemma 3.37. Take an arbitrary family $P : (A \times B) \to (i+j)\text{-}\mathsf{Type}$. The goal is to define a function

$$\theta : \left(\prod_{u:A\vee B} P(\mathsf{wtp}\, u)\right) \to \left(\prod_{v:A\times B} P(v)\right)$$

so that $(- \circ \mathsf{wtp}) \circ \theta \sim \mathsf{id}$. Let $g : \prod_{(u:A\vee B)} P(\mathsf{wtp}\, u)$. To define $\theta(g) : \prod_{(v:A\times B)} P(v)$, we construct first an uncurried function

$$f : \prod_{(a:A)} \prod_{(b:B)} P(a,b)$$

with the help of Lemma 3.51. Let

$$\begin{cases} f_{a_0} : \displaystyle\prod_{b:B} P(a_0, b), \\ f_{a_0}(b) :\equiv g(\mathsf{right}\, b), \end{cases}$$
$$\begin{cases} f_{b_0} : \displaystyle\prod_{a:A} P(a, b_0), \\ f_{b_0}(a) :\equiv g(\mathsf{left}\, a), \end{cases}$$
$$\begin{cases} p : (f_{b_0}(a_0) = f_{a_0}(b_0)) \equiv (g\, \mathsf{left}\, a_0 = g\, \mathsf{right}\, b_0), \\ p :\equiv \mathsf{ap}_g(\mathsf{glue} \star). \end{cases}$$

Therefore the function f exists. Moreover, Lemma 3.51 also gives two homotopies $\alpha : f(-, b_0) \sim f_{b_0}$ and $\beta : f(a_0, -) \sim f_{a_0}$, such that $p = \alpha(a_0)^{-1} \cdot \beta(b_0)$. Then, for $v : A \times B$ let

$$\theta(g, v) :\equiv f(\mathsf{pr}_1\, v, \mathsf{pr}_2\, v).$$

To finalise the proof, we have to show that

$$\prod_{u:A\vee B} (\theta(g) \circ \mathsf{wtp})(u) = g(u).$$

Given $a : A$ and $b : B$, this is done by wedge induction:

$$\begin{aligned}
(\theta(g) \circ \mathsf{wtp})(\mathsf{left}\, a) &\equiv \theta(g, (a, b_0)) \\
&\equiv f(a, b_0) \\
&= f_{b_0}(a) && \text{(by } \alpha(a)\text{)} \\
&\equiv g(\mathsf{left}\, a).
\end{aligned}$$

And similarly, we have $(\theta(g) \circ \mathsf{wtp})(\mathsf{right}\, b) = g(\mathsf{right}\, b)$, thanks to $\beta(b)$. For the third data of the induction, there is indeed a coherence path

$$\begin{aligned}
\mathsf{transport}^{(\theta(g)\circ\mathsf{wtp})(-)=g(-)}(\mathsf{glue}\,\star, \alpha(a_0)) &= \mathsf{ap}_{\theta(g)\circ\mathsf{wtp}}(\mathsf{glue}\,\star)^{-1} \cdot \alpha(a_0) \cdot \mathsf{ap}_g(\mathsf{glue}\,\star) && \text{(by Lemma 2.33)} \\
&= \mathsf{ap}_{\theta(g)}(\mathsf{refl}_{(a_0,b_0)}) \cdot \alpha(a_0) \cdot p && \text{(by functoriality)} \\
&= \alpha(a_0) \cdot p \\
&= \beta(b_0),
\end{aligned}$$

as desired. Therefore, θ is a section to $(- \circ \mathsf{wtp})$, which proves that wtp is $(i + j)$-connected.

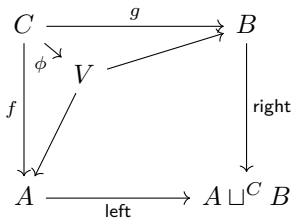
□

Chapter 4

Blakers-Massey theorem

The aim of this chapter is to prove the Blakers-Massey Connectivity Theorem. It follows a paper from Hou, Finster, Licata, and Lumsdaine [4]. The statement is the following:

Theorem 4.1 (Blakers-Massey Connectivity Theorem)**.** *Suppose we have a pushout and pullback as on the diagram*



where $V :\equiv A \times^{A \sqcup^C B} B$ *and*

$$\begin{cases} \phi : C \to V, \\ c \mapsto (fc, gc, \mathsf{glue}\, c). \end{cases}$$

If f *and* g *are* i*- and* j*-connected respectively, then* ϕ *is* $(i+j)$*-connected.*

To prove it, some additional lemmas are needed.

Lemma 4.2. *Given* $f : A \to B'$*,* $e : B \xrightarrow{\simeq} B'$ *and* $b : B$*, we have an equivalence*

$$\mathsf{fib\_at\_eqv}_{f,e} : \mathsf{fib}_f(eb) \xrightarrow{\simeq} \mathsf{fib}_{e^{-1} \circ f}(b).$$

Proof. By hypothesis, e is an equivalence. Here, we choose to look at it as an half adjoint equivalence (recall Definition 2.21) with indeed a reverse map $e^{-1} : B' \to B$, homotopies $\alpha : e^{-1}e \sim \mathsf{id}_B$ and $\beta : ee^{-1} \sim \mathsf{id}_{B'}$ and the two equivalent properties

$$\begin{aligned} &\prod_{b:B} \mathsf{ap}_e(\alpha b) = \beta(eb), \\ &\prod_{b':B'} \mathsf{ap}_{e^{-1}}(\beta b') = \alpha(e^{-1}b'). \end{aligned} \tag{1}$$

The quasi-inverse maps we are looking for can be defined as follows.

$$\begin{aligned} \left(\sum_{a:A} fa = eb\right) &\longleftrightarrow \left(\sum_{a:A} e^{-1}fa = b\right) \\ (a, p) &\longmapsto (a, \mathsf{ap}_{e^{-1}}(p) \centerdot \alpha(b)) \\ (a, (\beta(fa))^{-1} \centerdot \mathsf{ap}_e(q)) &\longmapsfrom (a, q) \end{aligned}$$

We have to prove two things:

$$\prod_{p:fa=eb} \beta(fa)^{-1} \cdot \mathsf{ap}_e(\mathsf{ap}_{e^{-1}} p \cdot \alpha b) = p,$$

and

$$\prod_{q:e^{-1}fa=b} \mathsf{ap}_{e^{-1}}(\beta(fa)^{-1} \cdot \mathsf{ap}_e q) \cdot \alpha b = q.$$

Let us prove the second one, the other one being analogous. By path induction, we can suppose $b \equiv e^{-1}fa$ and show the claim only for $q \equiv \mathsf{refl}_{e^{-1}fa}$. And indeed, we have

$$\begin{aligned}\mathsf{ap}_{e^{-1}}(\beta(fa)^{-1} \cdot \mathsf{ap}_e \mathsf{refl}_{e^{-1}fa}) \cdot \alpha(e^{-1}fa) &= \mathsf{ap}_{e^{-1}}\beta(fa)^{-1} \cdot \alpha(e^{-1}fa) \\ &\overset{(1)}{=} \alpha(e^{-1}fa)^{-1} \cdot \alpha(e^{-1}fa) \\ &= \mathsf{refl}_{e^{-1}fa}.\end{aligned}$$

□

Corollary 4.3. *Given $f : A \to B$, $e_1 : B \xrightarrow{\simeq} B'$, $e_2 : A' \xrightarrow{\simeq} A$ and $n : \mathbb{Z}_{\geqslant -2}$.*

i) f is n-connected $\Rightarrow e_1 \circ f$ is n-connected.

ii) f is n-connected $\Rightarrow f \circ e_2$ is n-connected.

The next lemma is a very technical computation formula. It follows from the same reasoning as in the observations made before Lemma 2.14. That is, when there are two different type-theoretic formulas that gives a path between the same endpoints or a function between the same very specific types, they often do not create any new homotopic data between them.

Lemma 4.4. *Let $A : \mathcal{U}$, $B : A \to \mathcal{U}$, and $C : \prod_{(a:A)} B(a) \to \mathcal{U}$, and also $a_1, a_2 : A$ with $m : a_1 = a_2$ and $b : B(a_2)$. Let $C' : (\sum_{(a:A)} B(a)) \to \mathcal{U}$ denote the uncurried form of C and $t : \mathsf{transport}^B(m, \mathsf{transport}^B(m^{-1}, b)) = b$ be the obvious coherence path. Then, the function*

$$\mathsf{transport}^{C'}(\mathsf{pair}^{=}(m, t), \_) : C(a_1, \mathsf{transport}^B(m^{-1}, b)) \to C(a_2, b),$$

is equal to the equivalence obtained from univalence from the composite

$$\begin{aligned}C(a_1, \mathsf{transport}^B(m^{-1}, b)) &= \mathsf{transport}^{B(\_) \to \mathcal{U}}(m, C(a_1))(b) && \text{(by Lem./Cor. 2.13, 2.32, and 2.34)} \\ &= C(a_2, b). && \text{(by } \mathsf{happly}(\mathsf{apd}_C(m), b))\end{aligned}$$

Proof. By path induction on m, we can assume $a_2 \equiv a_1$ and $m = \mathsf{refl}_{a_1}$. But then, everything simplifies and both functions are in fact the identity. □

The next two lemmas will be helpful to reformulate the hypotheses and the conclusion of the Blakers-Massey theorem when changing the point of view of the pushout for a pushout with predicate.

Lemma 4.5. *Suppose we have a span*

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

As seen in Section 3.5, we can without loss of generality assume it is span with a predicate $P : A \to B \to \mathcal{U}$. Then, for every $a : A$ we have

$$\mathsf{fib}_f(a) \simeq \sum_{b:B} P(a, b)$$

and for all $b : B$ we have

$$\mathsf{fib}_g(b) \simeq \sum_{a:A} P(a, b).$$

Proof. Recall that the predicate is defined as

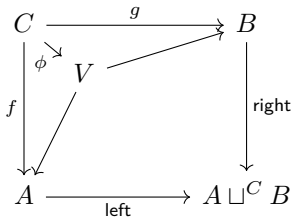
$$P : A \to B \to \mathcal{U},$$
$$P(a,b) :\equiv \sum_{c:C} ((fc = a) \times (gc = b)).$$

Observe that for each $a : A$,

$$\begin{aligned}
\mathsf{fib}_f(a) &\equiv \sum_{c:C} (fc = a) \\
&\simeq \sum_{c:C} \left((fc = a) \times \left(\sum_{b:B} (gc = b)\right)\right) && \text{(by Lemma 3.18)} \\
&\simeq \sum_{(b:B)} \sum_{(c:C)} ((fc = a) \times (gc = b)) && \text{(by reordering)} \\
&\equiv \sum_{b:B} P(a,b).
\end{aligned}$$

This observation makes sense, because every element $c : C$ in the fiber over a induces an element $b :\equiv g(c)$ which is related to a, i.e., $P(a,b)$ holds, and vice-versa. The other equivalence is obtain similarly. □

Lemma 4.6. *Suppose we have a pushout and pullback as on the diagram*



where $V :\equiv A \times^{A \sqcup^C B} B$ *and*

$$\begin{cases} \phi : C \to V, \\ c \mapsto (fc, gc, \mathsf{glue}\, c). \end{cases}$$

As seen in Section 3.5, we can without loss of generality assume it is a pushout with a predicate $P : A \to B \to \mathcal{U}$*. For all* $v : V$ *we have by uniqueness in* Σ*-types that* $v = (a, b, r)$ *for some* $a : A$*,* $b : B$ *and* $r : \mathsf{left}\, a = \mathsf{right}\, b$*. Then,*

$$\mathsf{fib}_\phi(v) \simeq \mathsf{fib}_{\mathsf{glue}^P(a,b)}(r),$$

Proof. Let $v : V$, and a, b, r be its projections so that $v = (a, b, r)$. Then, we have

$$
\begin{aligned}
\mathsf{fib}_\phi(v) \quad &\equiv \quad \sum_{c:C} (\phi c = v) \\
&\overset{\text{uniqueness in } \Sigma}{\simeq} \sum_{c:C} (fc, gc, \mathsf{glue}\, c) = (a, b, r) \\
&\overset{\text{equality in } \Sigma}{\simeq} \sum_{(c:C)} \sum_{(p_1 : fc = a)} \mathsf{transport}^{\sum_{(b':B)} \mathsf{left}(-) = \mathsf{right}\, b'}(p_1, (gc, \mathsf{glue}\, c)) \\
&\overset{\text{transport in } \Sigma}{\simeq} \sum_{(c:C)} \sum_{(p_1 : fc = a)} \left(gc, \mathsf{transport}^{(a',b') \mapsto \mathsf{left}\, a' = \mathsf{right}\, b'}(\mathsf{pair}^{=}(p_1, \mathsf{refl}_{gc}), \mathsf{glue}\, c)\right) = (b, r) \\
&\overset{\text{transport in } =}{\simeq} \sum_{(c:C)} \sum_{(p_1 : fc = a)} \left(gc, \mathsf{ap}_{\mathsf{left} \circ \mathsf{pr}_1} \mathsf{pair}^{=}(p_1, \mathsf{refl}_{gc})^{-1} \cdot \mathsf{glue}\, c \cdot \mathsf{ap}_{\mathsf{right} \circ \mathsf{pr}_2} \mathsf{pair}^{=}(p_1, \mathsf{refl}_{gc})\right) = (b, r) \\
&\overset{\text{functoriality of } \mathsf{ap}}{\simeq} \sum_{(c:C)} \sum_{(p_1 : fc = a)} (gc, \mathsf{ap}_{\mathsf{left}} {p_1}^{-1} \cdot \mathsf{glue}\, c) = (b, r) \\
&\overset{\text{equality in } \Sigma}{\simeq} \sum_{(c:C)} \sum_{(p_1 : fc = a)} \sum_{(p_2 : gc = b)} \mathsf{transport}^{\mathsf{left}\, a = \mathsf{right}(-)}(p_2, \mathsf{ap}_{\mathsf{left}} {p_1}^{-1} \cdot \mathsf{glue}\, c) = (b, r) \\
&\overset{\text{transport in } =}{\simeq} \sum_{(c:C)} \sum_{(p_1 : fc = a)} \sum_{(p_2 : gc = b)} \mathsf{ap}_{\mathsf{left}} {p_1}^{-1} \cdot \mathsf{glue}\, c \cdot \mathsf{ap}_{\mathsf{right}} p_2 = r \\
&\overset{\text{def. } \mathsf{glue}^P}{\equiv} \sum_{(c:C)} \sum_{(p_1 : fc = a)} \sum_{(p_2 : gc = b)} \mathsf{glue}^P(a, b, (c, p_1, p_2)) = r \\
&\overset{\Sigma \text{ associativity}}{\simeq} \sum_{p : P(a,b)} \mathsf{glue}^P(a, b, p) = r \\
&\equiv \quad \mathsf{fib}_{\mathsf{glue}^P(a,b)}(r).
\end{aligned}
$$

□

With all those lemmas behind us, we have now all the cards in hand to prove Blakers-Massey theorem.

Proof of Theorem 4.1. In the very first place, recall that the point of view can be modified into having a fat pushout with predicate, as seen in Section 3.5. This is done by defining a predicate

$$P : A \to B \to \mathcal{U},$$

and some functions:

$$
\begin{aligned}
\mathsf{glue}^P &: \prod_{(a:A)} \prod_{(b:B)} P(a, b) \to (\mathsf{left}\, a = \mathsf{right}\, b), \\
\mathsf{gluel} &: \prod_{(a:A)} \prod_{(b:B)} \prod_{(p:P(a,b))} (\mathsf{mid}\, p = \mathsf{left}\, a), \quad \text{and} \\
\mathsf{gluer} &: \prod_{(a:A)} \prod_{(b:B)} \prod_{(p:P(a,b))} (\mathsf{mid}\, p = \mathsf{right}\, b).
\end{aligned}
$$

To start well, let us also introduce two notations:

$$
\begin{aligned}
W &:\equiv A \sqcup^P_f B, \quad \text{and} \\
Z &:\equiv \sum_{(a,b):A \times B} P(a, b).
\end{aligned}
$$

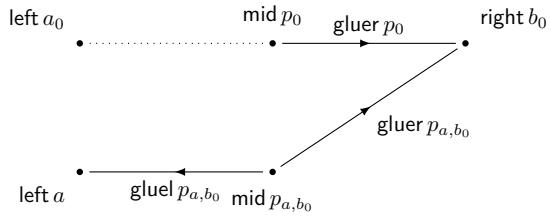
In other words, W is our (fat) pushout and Z is the type equivalent to C but in the vision with the predicate P. To help us, we suppose the existence of some basepoints $a_0 : A$, $b_0 : B$ and $p_0 : P(a_0, b_0)$, which by the way gives an element (a_0, b_0, p_0) of Z. The whole proof is made with them and at the end, we will see how we can discard those extra assumptions.

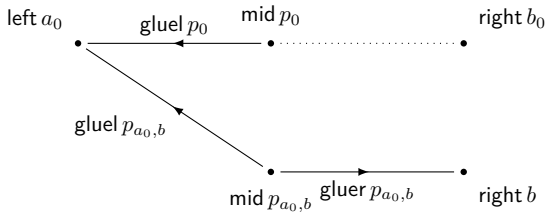
Thanks to Lemma 4.5 and Lemma 4.6, the hypotheses and the conclusion can be reformulated as such:

$$f \text{ is } i\text{-connected} \Leftrightarrow \text{for all } a : A, \text{ the type } \sum_{b:B} P(a,b) \text{ is } i\text{-connected},$$
$$g \text{ is } j\text{-connected} \Leftrightarrow \text{for all } b : A, \text{ the type } \sum_{a:B} P(a,b) \text{ is } j\text{-connected},$$
$$\phi \text{ is } (i+j)\text{-connected} \Leftrightarrow \text{for all } a : A \text{ and } b : B, \text{ the function } \mathsf{glue}^P(a,b) \text{ is } (i+j)\text{-connected}.$$

Given $a : A$ and $b : B$, the goal is now to prove the connectivity of $\mathsf{glue}^P(a,b)$. Thanks to Corollary 4.3, it can be done by proving that $\mathsf{glue}^P(a,b)$ composed with an equivalence is $(i+j)$-connected. Note that a function which takes paths as arguments and pre- or post-concatenates them with a fixed path is an equivalence, since it admits the obvious quasi-inverse which concatenates with the inverse of said fixed path. For this reason, let us define two functions for both symmetric cases.

$$\begin{cases} \mathsf{gluel}_0 : \prod_{a:A} P(a,b_0) \to (\mathsf{mid}\, p_0 = \mathsf{left}\, a) \\ \mathsf{gluel}_0(a, p_{a,b_0}) :\equiv \mathsf{gluer}\, p_0 \cdot (\mathsf{glue}^P\, p_{a,b_0})^{-1} \end{cases} \qquad \begin{cases} \mathsf{gluer}_0 : \prod_{b:B} P(a_0,b) \to (\mathsf{mid}\, p_0 = \mathsf{right}\, b) \\ \mathsf{gluer}_0(b, p_{a_0,b}) :\equiv \mathsf{gluel}\, p_0 \cdot \mathsf{glue}^P\, p_{a_0,b} \end{cases}$$





(Observe that the figures respect voluntarily the vision of the left, middle and right part respectively of the "cylinders" used in the gluing of the pushout.) Therefore, it suffices to prove the $(i+j)$-connectivity of either each $\mathsf{gluel}_0(a)$ or each $\mathsf{gluer}_0(b)$. This will be done by proving that the truncations of their fibers are contractible. To do this, we employ a method of encoding. We take a step back in our point of view and construct a more generic function

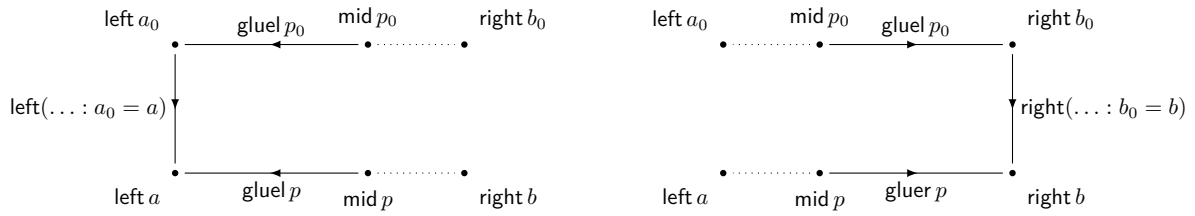
$$\mathsf{code} : \prod_{(w:W)} \prod_{(\alpha:\mathsf{mid}\, p_0 = w)} \mathcal{U}.$$

We want code to correspond with the fibers of all $\mathsf{gluel}_0(a)$ and $\mathsf{gluer}_0(b)$. For instance, in the case where w is of the form $\mathsf{right}\, b$ for some $b : B$ and we have some $\alpha : \mathsf{mid}\, p_0 = \mathsf{right}\, b$, we want to set $\mathsf{code}(w,\alpha)$ as $\left\|\mathsf{fib}_{\mathsf{gluer}_0(b)}(\alpha)\right\|_{i+j}$. Therefore, to prove the contractibility of the truncations of each $\mathsf{fib}_{\mathsf{gluel}_0(a)}(\alpha)$ or $\mathsf{fib}_{\mathsf{gluer}_0(b)}(\alpha)$, we can construct this code function and then prove more globally that each $\mathsf{code}(w,\alpha)$ is contractible.

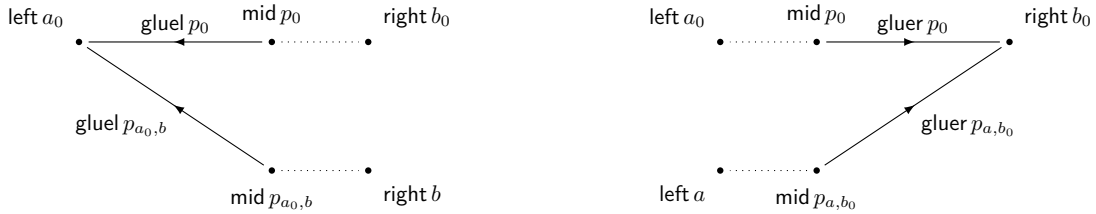
The definition of code is done by a fat pushout induction which requires five data. The first two are the ones that we just discussed above, i.e., for $a : A$ and $b : B$ we let

$$\mathsf{code}(\mathsf{left}\, a, \alpha : \mathsf{mid}\, p_0 = \mathsf{left}\, a) \quad :\equiv \left\|\mathsf{fib}_{\mathsf{gluel}_0(a)}(\alpha)\right\|_{i+j} \equiv \left\| \sum_{p_{a,b_0}:P(a,b_0)} \mathsf{gluel}_0(a, p_{a,b_0}) = \alpha \right\|_{i+j},$$
$$\mathsf{code}(\mathsf{right}\, b, \alpha : \mathsf{mid}\, p_0 = \mathsf{right}\, b) \quad :\equiv \left\|\mathsf{fib}_{\mathsf{gluer}_0(b)}(\alpha)\right\|_{i+j} \equiv \left\| \sum_{p_{a_0,b}:P(a_0,b)} \mathsf{gluer}_0(b, p_{a_0,b}) = \alpha \right\|_{i+j}.$$

Given $(a,b,p) : Z$, we want now to define $\mathsf{code}(\mathsf{mid}\, p)$. To stay coherent with the beginning of our definition, we also aim to define it as the truncation of the fibers of some map, which this time gives us path of the form $(\mathsf{mid}\, p_0 = \mathsf{mid}\, p)$. Observe that to have such a map, there are two ways. Either we have a connection $a_0 = a$ or another connection $b_0 = b$.



In the case where $a_0 = a$, then having $p : P(a, b)$ is equivalent thanks to transport as having some $p_{a_0,b} : P(a_0, b)$. Thus, having the connection $a_0 = a$ which gives us a path $\mathsf{left}\, a_0 = \mathsf{left}\, a$ is the same as having a path $\mathsf{left}\, a_0 = \mathsf{mid}\, p_{a_0,b}$. Then, the information that we have with our supposed elements (a, b, p) can be chosen in fact inside $\sum_{(b:B)} P(a_0, b)$. Similarly, when $b_0 = b$, the information we want can be formulated with $\sum_{(a:A)} P(a, b_0)$.



To express that we can have one case or the other, a disjoint union seems to do the job, except that (a_0, b_0, p_0) can fit in both cases. Thus, we need a disjoint union with one intersection of cases, i.e., a wedge product. We define it as follows:

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{(a_0,p_0)} & \sum_{(a:A)} P(a, b_0) \\ {\scriptstyle (b_0,p_0)}\downarrow & \overset{\mathsf{glue}}{\nearrow} & \downarrow{\scriptstyle \mathsf{right}} \\ \sum_{(b:B)} P(a_0, b) & \xrightarrow[\mathsf{left}]{} & \mathsf{P}\vee_{(a_0,b_0,p_0)} \end{array}$$

Observe that we put a subscript to $\mathsf{P}\vee$. Its purpose is to specify what our fixed element are, as later on we will work on only arbitrary variables. With this type in hand, we can now define a function, with a first initial parameter to specify the use of (a_0, b_0, p_0):

$$\mathsf{gluem}(a_0, b_0, p_0) : \mathsf{P}\vee_{(a_0,b_0,p_0)} \to \sum_{(a,b,p):Z} (\mathsf{mid}\, p_0 = \mathsf{mid}\, p)$$

obtained by induction with

$$\begin{cases} \mathsf{gluem}((a_0, b_0, p_0), \mathsf{left}(b, p_{a_0,b})) :\equiv ((a_0, b, p_{a_0,b}), \mathsf{gluel}\, p_0 \centerdot (\mathsf{gluel}\, p_{a_0,b})^{-1}), \\ \mathsf{gluem}((a_0, b_0, p_0), \mathsf{right}(a, p_{a,b_0})) :\equiv ((a, b_0, p_{a,b_0}), \mathsf{gluer}\, p_0 \centerdot (\mathsf{gluer}\, p_{a,b_0})^{-1}), \\ \qquad \mathsf{ap}_{\mathsf{gluem}(a_0,b_0,p_0)}(\mathsf{glue}\star) : ((a_0, b_0, p_0), \mathsf{gluel}\, p_0 \centerdot (\mathsf{gluel}\, p_0)^{-1}) = ((a_0, b_0, p_0), (\mathsf{gluer}\, p_0 \centerdot (\mathsf{gluer}\, p_0)^{-1})), \\ \qquad \mathsf{ap}_{\mathsf{gluem}(a_0,b_0,p_0)}(\mathsf{glue}\star) :\equiv \overset{=}{\mathsf{pair}}(\mathsf{refl}_{(a_0,b_0,p_0)}, \mathsf{rinv}(\mathsf{gluel}\, p_0) \centerdot \mathsf{rinv}(\mathsf{gluer}\, p_0)^{-1}). \end{cases}$$

We recall from Lemma 2.5 that for a path r, then $\mathsf{rinv}(r) : r \centerdot r^{-1} = \mathsf{refl}$. Now, for $a : A$, $b : B$ and $p : P(a, b)$ we let

$$\begin{aligned} \mathsf{code}(\mathsf{mid}\, p, \alpha : \mathsf{mid}\, p_0 = \mathsf{mid}\, p) &:\equiv \left\| \mathsf{fib}_{\mathsf{gluem}(a_0,b_0,p_0)}((a, b, p), \alpha) \right\|_{i+j} \\ &\equiv \left\| \sum_{q:\mathsf{P}\vee_{(a_0,b_0,p_0)}} \mathsf{gluem}((a_0, b_0, p_0), q) = ((a, b, p), \alpha) \right\|_{i+j}. \end{aligned}$$

Take arbitrary $a : A$, $b : B$ and $p : P(a, b)$. The last two steps needed for the definition of code are the coherence paths $\mathsf{apd}_{\mathsf{code}}(\mathsf{gluel}\, p)$ and $\mathsf{apd}_{\mathsf{code}}(\mathsf{gluer}\, p)$. This part of the proof is where actions start to

have a less obvious homotopic meaning, with considerations becoming more abstract and type-theoretic. The types needed are the following,

$$\begin{aligned}
&\mathsf{apd}_{\mathsf{code}}(\mathsf{gluel}\,p) && : \mathsf{transport}^{(\mathsf{mid}\,p_0=(-))\to\mathcal{U}}(\mathsf{gluel}\,p, \mathsf{code}(\mathsf{mid}\,p)) = \mathsf{code}(\mathsf{left}\,a), \quad \text{and}\\
&\mathsf{apd}_{\mathsf{code}}(\mathsf{gluer}\,p) && : \mathsf{transport}^{(\mathsf{mid}\,p_0=(-))\to\mathcal{U}}(\mathsf{gluer}\,p, \mathsf{code}(\mathsf{mid}\,p)) = \mathsf{code}(\mathsf{right}\,b).
\end{aligned}$$

By different properties about transport, here namely Lemma 2.32, Corollary 2.34 and Lemma 2.13, it suffices to prove that

$$\begin{aligned}
\Big(\alpha \mapsto \mathsf{code}(\mathsf{mid}\,p, \alpha \centerdot (\mathsf{gluel}\,p)^{-1})\Big) &= \Big(\alpha \mapsto \mathsf{code}(\mathsf{left}\,a, \alpha)\Big), \quad \text{and}\\
\Big(\beta \mapsto \mathsf{code}(\mathsf{mid}\,p, \beta \centerdot (\mathsf{gluer}\,p)^{-1})\Big) &= \Big(\beta \mapsto \mathsf{code}(\mathsf{right}\,b, \beta)\Big),
\end{aligned}$$

for $\alpha : \mathsf{mid}\,p_0 = \mathsf{left}\,a$ and $\beta : \mathsf{mid}\,p_0 = \mathsf{right}\,b$. Therefore, by function extensionality and univalence, it suffices to show for all such α and β that we have the equivalences

$$\mathsf{code}(\mathsf{mid}\,p, \alpha \centerdot (\mathsf{gluel}\,p)^{-1}) \simeq \mathsf{code}(\mathsf{left}\,a, \alpha), \text{ and} \tag{1}$$

$$\mathsf{code}(\mathsf{mid}\,p, \beta \centerdot (\mathsf{gluer}\,p)^{-1}) \simeq \mathsf{code}(\mathsf{right}\,b, \beta). \tag{2}$$

Both cases are symmetrical, thus we need to focus only on one of them, say the first one. In summary, with the definitions of code so far, the aim is to prove

$$\left\|\mathsf{fib}_{\mathsf{gluem}(a_0,b_0,p_0)}((a,b,p),\alpha_m)\right\|_{i+j} \simeq \left\|\mathsf{fib}_{\mathsf{gluel}_0(a)}(\alpha_l)\right\|_{i+j},$$

where

$$\begin{aligned}
\alpha_m &:\equiv \alpha \centerdot (\mathsf{gluel}\,p)^{-1} && : \mathsf{mid}\,p_0 = \mathsf{mid}\,p,\\
\alpha_l &:\equiv \alpha && : \mathsf{mid}\,p_0 = \mathsf{left}\,a.
\end{aligned}$$

Note that $\alpha_l = \alpha_m \centerdot \mathsf{gluel}\,p$ holds, and let us call s a witness of this equality.

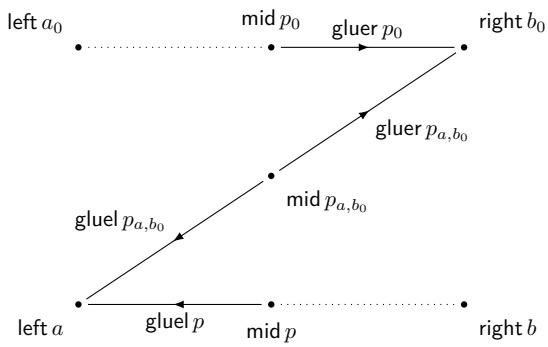
We split the desired equivalence into three easier ones,

$$\begin{aligned}
\left\|\mathsf{fib}_{\mathsf{gluel}_0\,a}(\alpha_l)\right\|_{i+j} &\overset{①}{\simeq} \left\|\mathsf{fib}_{\mathsf{gluel}_0\,a}(\alpha_m \centerdot \mathsf{gluel}\,p)\right\|_{i+j}\\
&\overset{②}{\simeq} \left\|\mathsf{fib}_{\mathsf{glueml}((a_0,b_0,p_0),(a,b,p))}(\alpha_m)\right\|_{i+j}\\
&\overset{③}{\simeq} \left\|\mathsf{fib}_{\mathsf{gluem}(a_0,b_0,p_0)}((a,b,p),\alpha_m)\right\|_{i+j},
\end{aligned}$$

where $\mathsf{glueml}(a_0,b_0,p_0)$ is an intermediate function defined as follows,

$$\begin{cases}
\mathsf{glueml}(a_0,b_0,p_0) : \displaystyle\prod_{(a,b,p):Z} P(a,b_0) \to (\mathsf{mid}\,p_0 = \mathsf{mid}\,p),\\
\mathsf{glueml}((a_0,b_0,p_0),(a,b,p),p_{a,b_0}) :\equiv \mathsf{gluer}\,p_0 \centerdot (\mathsf{glue}^P\,p_{a,b_0})^{-1} \centerdot (\mathsf{gluel}\,p)^{-1}.
\end{cases}$$

Note here again the initial argument (a_0,b_0,p_0).



The first one (1) follows from Lemma 3.32 which tells us that truncations preserve equivalences, because there is the following function which is an obvious equivalence:

$$\begin{cases} \mathsf{fib}_{\mathsf{gluel}_0\, a}(\alpha_l) \xrightarrow{\simeq} \mathsf{fib}_{\mathsf{gluel}_0\, a}(\alpha_m \centerdot \mathsf{gluel}\, p), \\ (p_{a,b_0}, \gamma : \mathsf{gluel}_0(a, p_{a,b_0}) = \alpha_l) \longmapsto (p_{a,b_0}, \gamma \centerdot s : \mathsf{gluel}_0(a, p_{a,b_0}) = \alpha_m \centerdot \mathsf{gluel}\, p). \end{cases}$$

The second one (2) follows from Lemma 3.32 again. Indeed, by Lemma 4.2, we have an equivalence with

$$\mathsf{fib\_at\_eqv}_{\mathsf{gluel}_0\, a, (-) \centerdot \mathsf{gluel}\, p} : \mathsf{fib}_{\mathsf{gluel}_0\, a}(\alpha_m \centerdot \mathsf{gluel}\, p) \xrightarrow{\simeq} \mathsf{fib}_{\mathsf{glueml}((a_0,b_0,p_0),(a,b,p))}(\alpha_m),$$

since $e :\equiv (-) \centerdot \mathsf{gluel}\, p$ is an equivalence and by definition we have

$$e^{-1} \circ \mathsf{gluel}_0(a) \equiv \mathsf{glueml}((a_0, b_0, p_0), (a, b, p)).$$

The third and last equivalence (3) is obtained as follows. The idea is to make use of Lemma 3.40, and thus to go to total functions. For that, let us first generalise both our concerned functions $\mathsf{glueml}(a_0, b_0, p_0)$ and $\mathsf{gluem}(a_0, b_0, p_0)$ by removing the initial arguments in order to have all variables being arbitrary. We let

$$\begin{aligned} \mathsf{glueml} &: \prod_{((a,b,p),(a',b',p')):Z\times Z} (P(a', b) \to (\mathsf{mid}\, p = \mathsf{mid}\, p')), \quad \text{and} \\ \mathsf{gluem} &: \prod_{(a,b,p):Z} \Big(\mathsf{P}\vee_{(a,b,p)} \to \sum_{(a',b',p'):Z} (\mathsf{mid}\, p = \mathsf{mid}\, p') \Big). \end{aligned}$$

Now that we generalised them to fiberwise functions, we consider the total functions they induce.

$$\begin{aligned} \mathsf{tot}(\mathsf{glueml}) &: \Big(\sum_{((a,b,p),(a',b',p')):Z\times Z} P(a', b) \Big) \to \Big(\sum_{((a,b,p),(a',b',p')):Z\times Z} (\mathsf{mid}\, p = \mathsf{mid}\, p') \Big), \\ \mathsf{tot}(\mathsf{gluem}) &: \Big(\sum_{(a,b,p):Z} \mathsf{P}\vee_{(a,b,p)} \Big) \to \Big(\sum_{((a,b,p):Z)} \sum_{((a',b',p'):Z)} (\mathsf{mid}\, p = \mathsf{mid}\, p') \Big). \end{aligned}$$

Observe that we can without loss of generality split the two dependent sums over $Z \times Z$ inside $\mathsf{tot}(\mathsf{glueml})$ each into two dependent sums on Z, as it is one equivalence away of what we have. With those two total functions in hand, we can invoke Lemma 3.40, which tells us that

$$\mathsf{fib}_{\mathsf{glueml}((a_0,b_0,p_0),(a,b,p))}(\alpha_m) \simeq \mathsf{fib}_{\mathsf{tot}(\mathsf{glueml})}((a_0, b_0, p_0), (a, b, p), \alpha_m),$$

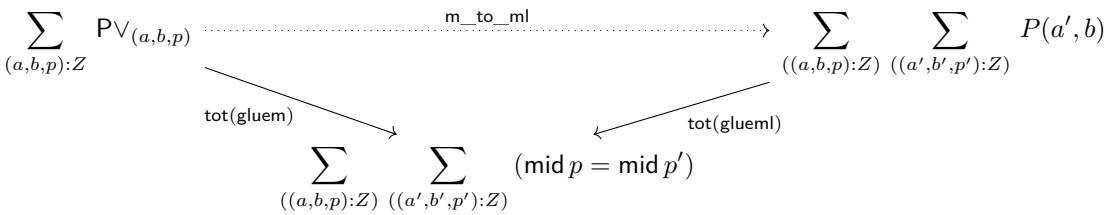
and

$$\mathsf{fib}_{\mathsf{gluem}(a_0,b_0,p_0)}((a, b, p), \alpha_m) \simeq \mathsf{fib}_{\mathsf{tot}(\mathsf{gluem})}((a_0, b_0, p_0), (a, b, p), \alpha_m).$$

As before, Lemma 3.32 tells us that equivalences are preserved when truncations are added. Therefore our goal is now reduced into proving that for all $\alpha : \mathsf{mid}\, p_0 = \mathsf{mid}\, p$, we have

$$\left\| \mathsf{fib}_{\mathsf{tot}(\mathsf{glueml})}((a_0, b_0, p_0), (a, b, p), \alpha) \right\|_{i+j} \simeq \left\| \mathsf{fib}_{\mathsf{tot}(\mathsf{gluem})}((a_0, b_0, p_0), (a, b, p), \alpha) \right\|_{i+j}.$$

The advantage behind this change of perspective with total functions is that it is now easier to employ Lemma 3.43. The only criterion needed is a third function, that we will call $\mathsf{m\_to\_ml}$ as on the diagram, and we need to prove it is $(i + j)$-connected and that it makes the triangle commute.



To define m_to_ml, a function from a wedge to a product as seen in Example 3.48 is of great use:

$$\mathsf{wtp} : \prod_{(a,b,p):Z} \left(\left(\left(\sum_{b':B} P(a,b') \right) \vee \left(\sum_{a':A} P(a',b) \right) \right) \to \left(\left(\sum_{b':B} P(a,b') \right) \times \left(\sum_{a':A} P(a',b) \right) \right) \right).$$

Indeed, as it induces a total function:

$$\mathsf{tot}(\mathsf{wtp}) : \left(\sum_{(a,b,p):Z} \mathsf{P}\vee_{(a,b,p)} \right) \to \left(\sum_{(a,b,p):Z} \left(\sum_{b':B} P(a,b') \right) \times \left(\sum_{a':A} P(a',b) \right) \right).$$

Observe that to get to the right codomain type, we need a slight rearrangement. For that, we define a function reordering and while we are at it, let us exhibit a quasi-inverse to reordering in order to already see that is is an equivalence.

$$\begin{cases} \mathsf{reordering} : \left(\sum_{(a,b,p):Z} \left(\sum_{b':B} P(a,b') \right) \times \left(\sum_{a':A} P(a',b) \right) \right) \longrightarrow \left(\sum_{((a,b,p):Z)} \sum_{((a',b',p'):Z)} P(a',b) \right), \\ \qquad ((a,b,p),(b',p_{a,b'}),(a',p_{a',b})) \longmapsto ((a',b,p_{a',b}),(a,b',p_{a,b'}),p), \\ \qquad ((a',b,p_{a',b}),(b',p'),(a,p)) \longmapsto\!\shortmid\ ((a,b,p),(a',b',p'),p_{a',b}). \end{cases}$$

One may see that already on the first component, the triangle will not commute if we set m_to_ml to be the composition of tot(wtp) and reordering. Thus, a third function is needed, say

$$\mathsf{switchr} : \left(\sum_{z:Z} \mathsf{P}\vee_z \right) \to \left(\sum_{z:Z} \mathsf{P}\vee_z \right),$$

that we define by Σ-induction and wedge induction as follows:

$$\begin{cases} \mathsf{switchr}((a,b,p),\mathsf{left}(b',p_{a,b'})) :\equiv ((a,b,p),\mathsf{left}(b',p_{a,b'})), \\ \mathsf{switchr}((a,b,p),\mathsf{right}(a',p_{a',b})) :\equiv ((a',b,p_{a',b}),\mathsf{right}(a,p)), \\ \qquad \mathsf{ap}_{\mathsf{switchr}(a,b,p)}(\mathsf{glue}\star) : ((a,b,p),\mathsf{left}(b,p)) = ((a,b,p),\mathsf{right}(a,p)), \\ \qquad \mathsf{ap}_{\mathsf{switchr}(a,b,p)}(\mathsf{glue}\star) :\equiv \mathsf{pair}^{=}(\mathsf{refl}_{(a,b,p)},\mathsf{glue}\star). \end{cases}$$

Now, we can finally set

$$\begin{cases} \mathsf{m\_to\_ml} : \left(\sum_{z:Z} \mathsf{P}\vee_z \right) \to \left(\sum_{((a,b,p):Z)} \sum_{((a',b',p'):Z)} P(a',b) \right), \\ \mathsf{m\_to\_ml} :\equiv \mathsf{reordering} \circ \mathsf{tot}(\mathsf{wtp}) \circ \mathsf{switchr}\,. \end{cases}$$

It is time to prove that m_to_ml is $(i+j)$-connected. By hypothesis, for each $a : A$ the type $\sum_{(b':B)} P(a,b')$ is i-connected and for each $b : B$ the type $\sum_{(a':A)} P(a',b)$ is j-connected. Therefore, the wedge connectivity lemma (Lemma 3.52) implies that each function $\mathsf{wtp}(a,b,p)$ is $(i+j)$-connected. Thus, by Corollary 3.41 the function tot(wtp) is also $(i+j)$-connected.

It was noted earlier, that reordering is an equivalence. But switchr is also one, as it is its own inverse. To see that, let

$$\mathsf{lemma} : \prod_{x:\sum_{(z:Z)} \mathsf{P}\vee_z} \mathsf{switchr}\,\mathsf{switchr}(x) = x$$

be defined by Σ-induction and wedge induction:

$$\begin{cases} \mathsf{lemma}((a,b,p),\mathsf{left}(b',p_{a,b'})) :\equiv \mathsf{refl}_{((a,b,p),\mathsf{left}(b',p_{a,b'}))}, \\ \mathsf{lemma}((a,b,p),\mathsf{right}(a',p_{a',b})) :\equiv \mathsf{refl}_{((a,b,p),\mathsf{right}(a',p_{a',b}))}, \\ \qquad \mathsf{ap}_{\mathsf{lemma}(a,b,p)}(\mathsf{glue}\star) : \mathsf{refl}_{((a,b,p),\mathsf{left}(b,p))} = \mathsf{refl}_{((a,b,p),\mathsf{right}(a,p))}, \\ \qquad \mathsf{ap}_{\mathsf{lemma}(a,b,p)}(\mathsf{glue}\star) :\equiv \mathsf{ap}_{\mathsf{refl}}\,\mathsf{ap}_{\mathsf{switchr}(a,b,p)}(\mathsf{glue}\star). \end{cases}$$

Therefore m_to_ml is indeed $(i+j)$-connected, since Corollary 4.3 tells us that composing with equivalences does not change the connectivity.

The last thing to prove in order to invoke Lemma 3.43 is that the triangle commute, i.e., that we have some witness

$$\mathsf{tr\_commute} : \prod_{x:\sum_{(z:Z)} \mathsf{P}\vee_z} \mathsf{tot}(\mathsf{gluem})(x) = \mathsf{tot}(\mathsf{glueml})(\mathsf{m\_to\_ml}(x)).$$

Here is an instance where the computability aspect is prevailing. In the Agda code, all functions that have ever been defined are stored, including all elementary path properties, as seen for instance in the beginning of Chapter 2. Here, we have to do it by hand, which is tedious as it for instance requires to be overly explicit in the name of all witnesses. The definition of $\mathsf{tr\_commute}$ is in fact quite elementary, as it consists of only some quite basic paths properties, but the problem lies in the complexity of some computation rules of our theory. We will still do it in order to at least once realise what we are dealing with. But in order to avoid taking too much space for this computation, this proof that $\mathsf{tr\_commute}$ exists is left in the appendix.

Now that code is finally defined it is time to prove its contractibility. Recall that

$$\mathsf{code} : \prod_{(w:W)} \prod_{(\alpha:\mathsf{mid}\, p_0 = w)} \mathcal{U}.$$

To prove that for each $w : W$ and $\alpha : \mathsf{mid}\, p_0 = w$ the type $\mathsf{code}(w, \alpha)$ is contractible, we need a center of contraction, i.e., a function

$$\mathsf{center} : \prod_{(w:W)} \prod_{(\alpha:\mathsf{mid}\, p_0 = w)} \mathsf{code}(w, \alpha).$$

We begin by defining an element center_0 for $\mathsf{center}(\mathsf{mid}\, p_0, \mathsf{refl}_{\mathsf{mid}\, p_0})$, which must be of type

$$\mathsf{code}(\mathsf{mid}\, p_0, \mathsf{refl}_{\mathsf{mid}\, p_0}) \equiv \left\| \sum_{q:\mathsf{P}\vee} \mathsf{gluem}((a_0, b_0, p_0), q) = ((a_0, b_0, p_0), \mathsf{refl}_{\mathsf{mid}\, p_0}) \right\|_{i+j}.$$

We let $\mathsf{center}_0 :\equiv |(\mathsf{left}(b_0, p_0), \mathsf{centercoh})|_{i+j}$, where $\mathsf{centercoh}$ is the following coherence path:

$$\begin{cases} \mathsf{centercoh} : \Big(\mathsf{gluem}((a_0, b_0, p_0), \mathsf{left}(b_0, p_0)) = ((a_0, b_0, p_0), \mathsf{refl}_{\mathsf{mid}\, p_0}) \Big) \\ \qquad\qquad \equiv \Big(((a_0, b_0, p_0), \mathsf{gluel}\, p_0 \cdot (\mathsf{gluel}\, p_0)^{-1}) = ((a_0, b_0, p_0), \mathsf{refl}_{\mathsf{mid}\, p_0}) \Big), \\ \mathsf{centercoh} :\equiv \mathsf{pair}^{=}(\mathsf{refl}_{(a_0, b_0, p_0)}, \mathsf{rinv}(\mathsf{gluel}\, p_0)). \end{cases}$$

This allows us to define every center of contraction by transporting from the case $(\mathsf{mid}\, p_0, \mathsf{refl}_{\mathsf{mid}\, p_0})$. For practical reasons, we will do transport with an uncurried form of code, which is

$$\begin{cases} \mathsf{code}' : \prod_{y:\sum_{(w:W)} \mathsf{mid}\, p_0 = w} \mathcal{U}, \\ \mathsf{code}'(y) :\equiv \mathsf{code}(\mathsf{pr}_1\, y, \mathsf{pr}_2\, y). \end{cases}$$

Then, we can define

$$\begin{cases} \mathsf{center} : \prod_{(w:W)} \prod_{(\alpha:\mathsf{mid}\, p_0 = w)} \mathsf{code}(w, \alpha), \\ \mathsf{center}(w, \alpha) :\equiv \mathsf{transport}^{\mathsf{code}'}(\mathsf{pair}^{=}(\alpha, \mathsf{connOver}), \mathsf{center}_0), \end{cases}$$

but note that for the definition to be complete, we still need another coherence path that we call $\mathsf{connOver}$. What we want is

$$\mathsf{transport}^{\mathsf{code}'}(\mathsf{pair}^{=}(\alpha, \mathsf{connOver}), -) : \mathsf{code}(\mathsf{mid}\, p_0, \mathsf{refl}_{\mathsf{mid}\, p_0}) \to \mathsf{code}(w, \alpha).$$

Therefore, if we go back to the definition of $\mathsf{pair}^{=}$ in Theorem 2.31, we see that we need a dependent path

$$\mathsf{connOver} : \mathsf{refl}_{\mathsf{mid}\, p_0} =_{\alpha}^{\mathsf{mid}\, p_0 = (-)} \alpha.$$

Thus, we can set $\mathsf{connOver}$ to be the following composite:

$$\begin{aligned}\mathsf{transport}^{\mathsf{mid}\,p_0=(-)}(\alpha, \mathsf{refl}_{\mathsf{mid}\,p_0}) &= \mathsf{refl}_{\mathsf{mid}\,p_0} \centerdot \alpha && \text{(by Corollary 2.34)}\\ &= \alpha. && \text{(by Lemma 2.5)}\end{aligned}$$

Now that we have centers of contraction, the goal is to prove

$$\prod_{((w,\alpha):\sum_{(w:W)} \mathsf{mid}\,p_0=w)} \underbrace{\prod_{(c:\mathsf{code}(w,\alpha))} \mathsf{center}(w,\alpha) = c}_{\equiv:R(w,\alpha)}.$$

Thanks to Lemma 3.18, we know that the type $\sum_{(w:W)} \mathsf{mid}\,p_0 = w$ is contractible, and thus it suffices to prove $R(w,\alpha)$ in a specific case and then transport will assures us that every $R(w,\alpha)$ hold. We choose the situation where $w \equiv \mathsf{right}\,b$ for some $b : B$ but with $\alpha : (\mathsf{mid}\,p_0 = \mathsf{right}\,b)$ still arbitrary. Considering the more general case with α not fixed allows us to perform a very practical path induction in just a moment. Let us take

$$c : \mathsf{code}(\mathsf{right}\,b, \alpha) \equiv \left\| \sum_{p_{a_0,b}:P(a_0,b)} \mathsf{gluel}\,p_0 \centerdot \mathsf{glue}^P\,p_{a_0,b} = \alpha \right\|_{i+j}.$$

Since the equality $\mathsf{center}(\mathsf{right}\,b, \alpha) = c$ takes place in an $(i+j)$-type, the equality type itself is an $(i+j-1)$-type so in particular an $(i+j)$-type. Therefore, by truncation induction we can suppose that c is of the form $\left|(p_{a_0,b}, r : \mathsf{gluel}\,p_0 \centerdot \mathsf{glue}^P\,p_{a_0,b} = \alpha)\right|_{i+j}$. By a path induction that we teased just before, we can suppose that $\alpha \equiv \mathsf{gluel}\,p_0 \centerdot \mathsf{glue}^P\,p_{a_0,b}$ and that r is the reflexive path. In summary, we now aim to show

$$\begin{aligned}\mathsf{transport}^{\mathsf{code}'}(\mathsf{pair}^{=}(\mathsf{gluel}\,p_0 \centerdot \mathsf{glue}^P\,p_{a_0,b}, \mathsf{connOver}), |(\mathsf{left}(b_0,p_0), \mathsf{centercoh})|_{i+j})\\ = \left|(p_{a_0,b}, \mathsf{refl}_{\mathsf{gluel}\,p_0 \centerdot \mathsf{glue}^P\,p_{a_0,b}})\right|_{i+j}.\end{aligned}$$

We just arrived at the other part heavy in computations of the proof. The idea is to apply Lemma 4.4 with the following correspondence of notations:

$$\begin{aligned}&\text{“}A\text{”} :\equiv \text{our pushout } W,\\ &\text{“}B\text{”} :\equiv \mathsf{mid}\,p_0 = (-) : W \to \mathcal{U},\\ &\text{“}C\text{”} :\equiv \mathsf{code},\\ &\text{“}m\text{”} :\equiv \alpha : \mathsf{mid}\,p_0 = \mathsf{right}\,b, \text{ and}\\ &\text{“}b\text{”} :\equiv \alpha : \text{“}B(\mathsf{right}\,b)''.\end{aligned}$$

Note that in the lemma, we consider the function obtained from univalence, function extensionality, Lemmas 2.13 and 2.32 and Corollary 2.34. These operations were also the one we performed at the beginning of the definitions of both cases $\mathsf{ap}_{\mathsf{code}}(\mathsf{gluel}\,p)$ and $\mathsf{ap}_{\mathsf{code}}(\mathsf{gluer}\,p)$. Recall also that $\alpha \equiv \mathsf{gluel}\,p_0 \centerdot (\mathsf{gluel}\,p_{a_0,b})^{-1} \centerdot \mathsf{gluer}\,p_{a_0,b}$ and thus $\mathsf{ap}_{\mathsf{code}}(\alpha)$ is a combination of $\mathsf{ap}_{\mathsf{code}}$ in both cases with gluel and gluer. Therefore, what Lemma 4.4 implies is that the above transport in code' is essentially a combination of both equivalences (1) and (2). This is therefore only a very long and tedious computation, which make use of the precise definitions of all intermediate equivalences we employed. It also uses $\mathsf{ap}_{\mathsf{code}}$ with a gluer input, which is the case we did not explicitly describe, as it is symmetric to the one we did. Hence, we let the full details of those calculations to the reader, knowing that all of these details have been coded and computer-checked, and are available online [3]. It consists in the observation that the first component inside the $(i+j)$-projection, $\mathsf{left}(b_0,p_0)$, is sent to $p_{a_0,b}$ and that on the second component, all coherence paths eventually cancel to leave only a reflexive path at the very end.

Now that we have the contractibility of each $\mathsf{code}(w,\alpha)$, let us come back to what we discussed at the start of the proof. Take arbitrary $a : A$ and $b : B$ and recall that we have

$$\begin{aligned}&\mathsf{code}(\mathsf{left}\,a, \alpha : \mathsf{mid}\,p_0 = \mathsf{left}\,a) \equiv \left\|\mathsf{fib}_{\mathsf{gluel}_0(a)}(\alpha)\right\|_{i+j}, \text{ and}\\ &\mathsf{code}(\mathsf{right}\,b, \alpha : \mathsf{mid}\,p_0 = \mathsf{right}\,b) \equiv \left\|\mathsf{fib}_{\mathsf{gluer}_0(b)}(\alpha)\right\|_{i+j}.\end{aligned}$$

Since they are all contractible, it means the functions

$$\begin{aligned}&\mathsf{gluel}_0(a) : P(a, b_0) \to (\mathsf{mid}\, p_0 = \mathsf{left}\, a), \quad \text{and}\\ &\mathsf{gluer}_0(b) : P(a_0, b) \to (\mathsf{mid}\, p_0 = \mathsf{right}\, b),\end{aligned}$$

are $(i+j)$-connected. The definition of gluer_0 directly implies that

$$\mathsf{glue}^P(a_0, b, p_{a_0,b}) = (\mathsf{gluel}\, p_0)^{-1} \cdot \mathsf{gluer}_0(b, p_{a_0,b}).$$

Hence, by function extensionality, we have

$$\mathsf{glue}^P(a_0, b, -) = (\mathsf{gluel}\, p_0)^{-1} \cdot \mathsf{gluer}_0(b, -).$$

The function that concatenates with $(\mathsf{gluel}\, p_0)^{-1}$ is an equivalence, and since composing with an equivalence preserve connectivity as seen in Corollary 4.3, we have that each function $\mathsf{glue}^P(a_0, b)$ is $(i+j)$-connected, which we have seen to be an equivalent formulation of our goal. But there is a very last thing to do, which is getting rid of the extra assumptions that we have. To summarize, we proved that

$$\forall a_0 : A, \forall b, b_0 : B, \forall p_0 : P(a_0, b_0)(\mathsf{glue}^P(a_0, b) \text{ is } (i+j)\text{-connected}).$$

If we show that the hypotheses of having b_0 and p_0 are not necessary, then we will have

$$\forall a_0 : A, \forall b : B, (\mathsf{glue}^P(a_0, b) \text{ is } (i+j)\text{-connected}),$$

as desired. The reasoning we make is that for each $a_0 : A$, we have:

$$\begin{aligned}\sum_{b_0:B} P(a_0, b_0) \text{ is } i\text{-connected} &\Leftrightarrow \left\| \sum_{b_0:B} P(a_0, b_0) \right\|_i \text{ is contractible}\\ &\Rightarrow \left\| \sum_{b_0:B} P(a_0, b_0) \right\|_i \text{ has an element (the center of contraction)}\\ &\Rightarrow \left\| \left\| \sum_{b_0:B} P(a_0, b_0) \right\|_i \right\| \overset{\text{Lemma 3.34}}{=} \left\| \sum_{b_0:B} P(a_0, b_0) \right\| \text{ is inhabited.}\end{aligned}$$

What we proved at the end is the connectivity of $\mathsf{glue}^P(a_0, b)$, which is a mere proposition (as discussed after Lemma 3.8), i.e., a (-1)-type. Therefore, by the universal property of the truncations (Lemma 3.30), we can drop the (-1)-truncation above and suppose that $\sum_{(b_0:B)} P(a_0, b_0)$ is inhabited, i.e., that we have some (b_0, p_0). This proves that the supposition of having b_0 and p_0 may be dropped and the proof is finished.

□

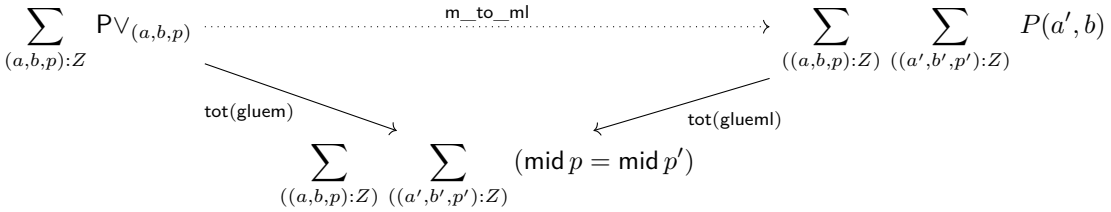
Appendix

Detail in the proof of Blakers-Massey

The following claim is a part of the proof of Blakers-Massey Connectivity Theorem (Theorem 4.1). It is voluntarily written outside of the proof of Chapter 4 as it is only a minor detail, and that clashes with the length that it takes.

Claim. In the context of the proof of Theorem 4.1, we have a witness that the following triangle diagram commutes.

$$\mathsf{tr\_commute} : \prod_{x:\sum_{(z:Z)} \mathsf{P}\vee_z} \mathsf{tot}(\mathsf{gluem})(x) = \mathsf{tot}(\mathsf{glueml})(\mathsf{m\_to\_ml}(x)).$$



Proof. Let S denote the family of equalities that we want to prove, i.e., for $x : \sum_{(z:Z)} \mathsf{P}\vee_z$, let

$$S(x) :\equiv \mathsf{tot}(\mathsf{gluem})(x) = \mathsf{tot}(\mathsf{glueml})(\mathsf{m\_to\_ml}(x)).$$

By Σ-induction and wedge induction, we need to look at three cases. For $a, a' : A$, $b, b' : B$, $p : P(a, b)$, $p_{a',b} : P(a', b)$ and $p_{a,b'} : P(a, b')$, we require:

1. $\mathsf{tr\_commute}((a, b, p), \mathsf{left}(b', p_{a,b'}))$,
2. $\mathsf{tr\_commute}((a, b, p), \mathsf{right}(a', p_{a',b}))$,
3. and the coherence path $\mathsf{ap}_{\mathsf{tr\_commute}(a,b,p)}(\mathsf{glue}\,\star)$.

Let us proceed in the same order.

1. When x is of the form $((a, b, p), \mathsf{left}(b', p_{a,b'}))$, observe that

$$\begin{aligned}\mathsf{tot}(\mathsf{gluem})((a, b, p), \mathsf{left}(b', p_{a,b'})) &\equiv ((a, b, p), \mathsf{gluem}((a, b, p), \mathsf{left}(b', p_{a,b'})))\\ &\equiv ((a, b, p), (a, b', p_{a,b'}), \mathsf{gluel}\, p \cdot (\mathsf{gluel}\, p_{a,b'})^{-1})\end{aligned}$$

and

$$\begin{aligned}&\mathsf{tot}(\mathsf{glueml}) \circ \mathsf{m\_to\_ml}((a, b, p), \mathsf{left}(b', p_{a,b'}))\\ &\qquad\equiv \mathsf{tot}(\mathsf{glueml}) \circ \mathsf{reordering} \circ \mathsf{tot}(\mathsf{wtp})((a, b, p), \mathsf{left}(b', p_{a,b'}))\\ &\qquad\equiv \mathsf{tot}(\mathsf{glueml}) \circ \mathsf{reordering}((a, b, p), (b', p_{a,b'}), (a, p))\\ &\qquad\equiv \mathsf{tot}(\mathsf{glueml})((a, b, p), (a, b', p_{a,b'}), p)\\ &\qquad\equiv ((a, b, p), (a, b', p_{a,b'}), \mathsf{gluem}l((a, b, p), a, b', p_{a,b'}, p))\\ &\qquad\equiv ((a, b, p), (a, b', p_{a,b'}), \mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p)^{-1} \cdot \mathsf{gluel}\, p \cdot (\mathsf{gluel}\, p_{a,b'})^{-1}).\end{aligned}$$

Therefore, there is a need for the reflexive paths on the first two components, and on the third component we need the concatenation with $\mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p)^{-1}$ to appear. Formally, this is written as follows.

$$\mathsf{tr\_commute}((a,b,p),\mathsf{left}(b',p_{a,b'})) :\equiv \mathsf{pair}^{=}(\mathsf{refl}_{(a,b,p)},\mathsf{pair}^{=}(\mathsf{refl}_{(a,b',p_{a,b'})},\delta_{b',p_{a,b'}})),$$

where

$$\begin{cases} \delta_{b',p_{a,b'}} : \mathsf{gluel}\, p \cdot (\mathsf{gluel}\, p_{a,b'})^{-1} = \mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p)^{-1} \cdot \mathsf{gluel}\, p \cdot (\mathsf{gluel}\, p_{a,b'})^{-1}, \\ \delta_{b',p_{a,b'}} :\equiv \mathsf{left\_refl}(\mathsf{gluel}\, p \cdot (\mathsf{gluel}\, p_{a,b'})^{-1}) \cdot \mathsf{ap}_{(-)\cdot \mathsf{gluel}\, p \cdot (\mathsf{gluel}\, p_{a,b'})^{-1}}(\mathsf{rinv}(\mathsf{gluer}\, p)^{-1}). \end{cases}$$

About the notations, recall that for an arbitrary path r, we have $\mathsf{left\_refl}(r) : r = \mathsf{refl} \cdot r$ and $\mathsf{rinv}(r) : r \cdot r^{-1} = \mathsf{refl}$.

2. Similarly, we have

$$\begin{aligned} \mathsf{tot}(\mathsf{gluem})((a,b,p),\mathsf{right}(a',p_{a',b})) &\equiv ((a,b,p),\mathsf{gluem}((a,b,p),\mathsf{right}(a',p_{a',b}))) \\ &\equiv ((a,b,p),(a',b,p_{a',b}),\mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p_{a',b})^{-1}) \end{aligned}$$

and

$$\begin{aligned} &\mathsf{tot}(\mathsf{glueml}) \circ \mathsf{m\_to\_ml}((a,b,p),\mathsf{right}(a',p_{a',b})) \\ &\quad\equiv \mathsf{tot}(\mathsf{glueml}) \circ \mathsf{reordering} \circ \mathsf{tot}(\mathsf{wtp})((a',b,p_{a',b}),\mathsf{right}(a,p)) \\ &\quad\equiv \mathsf{tot}(\mathsf{glueml}) \circ \mathsf{reordering}((a',b,p_{a',b}),(b,p_{a',b}),(a,p)) \\ &\quad\equiv \mathsf{tot}(\mathsf{glueml})((a,b,p),(a',b,p_{a',b}),p_{a',b}) \\ &\quad\equiv ((a,b,p),(a',b,p_{a',b}),\mathsf{glueml}((a,b,p),a',b,p_{a',b},p_{a',b})) \\ &\quad\equiv ((a,b,p),(a',b,p_{a',b}),\mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p_{a',b})^{-1} \cdot \mathsf{gluel}\, p_{a',b} \cdot (\mathsf{gluel}\, p_{a',b})^{-1}). \end{aligned}$$

Therefore, there is a need for the reflexive paths on the first two components, and on the third component we need the concatenation with $\mathsf{gluel}\, p_{a',b} \cdot (\mathsf{gluel}\, p_{a',b})^{-1}$ to appear. Formally, this is written as follows.

$$\mathsf{tr\_commute}((a,b,p),\mathsf{right}(a',p_{a',b})) :\equiv \mathsf{pair}^{=}(\mathsf{refl}_{(a,b,p)},\mathsf{pair}^{=}(\mathsf{refl}_{(a,b',p_{a,b'})},\epsilon_{a',p_{a',b}})),$$

where

$$\begin{cases} \epsilon_{a',p_{a',b}} : \mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p_{a',b})^{-1} = \mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p_{a',b})^{-1} \cdot \mathsf{gluel}\, p_{a',b} \cdot (\mathsf{gluel}\, p_{a',b})^{-1}, \\ \epsilon_{a',p_{a',b}} :\equiv \mathsf{right\_refl}(\mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p_{a',b})^{-1}) \cdot \mathsf{ap}_{\mathsf{gluer}\, p \cdot (\mathsf{gluer}\, p_{a',b})^{-1} \cdot (-)}(\mathsf{rinv}(\mathsf{gluer}\, p)^{-1}). \end{cases}$$

3. Now, we need the coherence path $\mathsf{ap}_{\mathsf{tr\_commute}(a,b,p)}(\mathsf{glue}\,\star)$ which must be of type

$$\mathsf{pair}^{=}(\mathsf{refl}_{(a,b,p)},\mathsf{pair}^{=}(\mathsf{refl}_{(a,b,p)},\delta_{b,p})) =^{S}_{\mathsf{glue}\,\star} \mathsf{pair}^{=}(\mathsf{refl}_{(a,b,p)},\mathsf{pair}^{=}(\mathsf{refl}_{(a,b,p)},\epsilon_{a,p})).$$

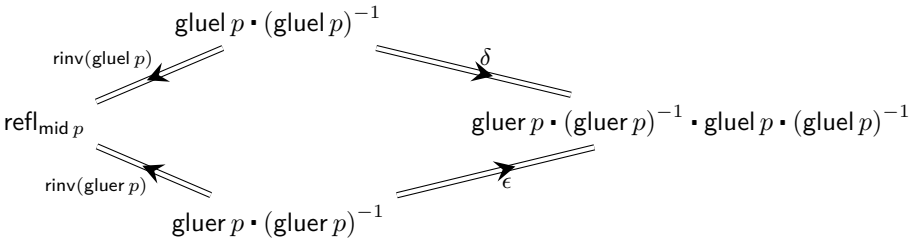
Since S is an equality type, we can use Lemma 2.33 to simplify the goal. Let us simplify a bit the notations, in particular by denoting $\mathsf{tot}(\mathsf{gluem})$ with h and $\mathsf{tot}(\mathsf{glueml}) \circ \mathsf{m\_to\_ml}$ with k. What Lemma 2.33 tells us is that it suffices to inhabit

$$\mathsf{ap}_h(\mathsf{glue}\,\star)^{-1} \cdot \mathsf{pair}^{=}(\mathsf{refl},\mathsf{pair}^{=}(\mathsf{refl},\delta)) \cdot \mathsf{ap}_k(\mathsf{glue}\,\star) = \mathsf{pair}^{=}(\mathsf{refl},\mathsf{pair}^{=}(\mathsf{refl},\epsilon)).$$

If we go back to the definitions of h and k and use the functoriality of ap (Lemma 2.8), we compute that

$$\begin{aligned} \mathsf{ap}_h(\mathsf{glue}\,\star)^{-1} &= \mathsf{pair}^{=}(\mathsf{refl},\mathsf{pair}^{=}(\mathsf{refl},\mathsf{rinv}(\mathsf{gluer}\, p) \cdot \mathsf{rinv}(\mathsf{gluel}\, p)^{-1})), \quad \text{and} \\ \mathsf{ap}_k(\mathsf{glue}\,\star) &= \mathsf{refl}. \end{aligned}$$

We also observe that the property $\mathsf{rinv}(\mathsf{gluer}\, p) \cdot \mathsf{rinv}(\mathsf{gluel}\, p)^{-1} = \epsilon \cdot \delta^{-1}$ holds, as it is immediate by path induction. It makes sense homotopically that everything contracts, as we are only manipulating paths doing round trips.



Therefore, using the properties of $\mathsf{pair}^{=}$ (such as on page 27), everything simplifies and we are left with proving that

$$\mathsf{pair}^{=}(\mathsf{refl}, \mathsf{pair}^{=}(\mathsf{refl}, \epsilon)) = \mathsf{pair}^{=}(\mathsf{refl}, \mathsf{pair}^{=}(\mathsf{refl}, \epsilon))$$

holds. But both sides are judgmentally equal, thus we can choose the reflexive path, and proof is finished.

□

Bibliography

[1] Mathieu Anel, Georg Biedermann, Eric Finster, and André Joyal. "A Generalized Blakers-Massey Theorem". In: *arXiv e-prints* (Mar. 2017). arXiv: 1703.09050.

[2] Emmanuel D. Farjoun. *Cellular Spaces, Null Spaces and Homotopy Localization*. Lecture Notes in Mathematics. Springer-Verlag Berlin Heidelberg, p. 179.

[3] Kuen-Bang Hou. *BlakersMassey.agda*. https://github.com/HoTT/HoTT-Agda/blob/1.0/Homotopy/BlakersMassey.agda. Accessed: 2019-06-17.

[4] Kuen-Bang Hou, Eric Finster, Dan Licata, and Peter LeFanu Lumsdaine. "A mechanization of the Blakers-Massey connectivity theorem in Homotopy Type Theory". In: *arXiv e-prints* (May 2016). arXiv: 1605.03227.

[5] Charles Rekz. *Proof of the Blakers-Massey theorem*. 2014.

[6] Charles Rezk. *Toposes and homotopy toposes (version 1.05)*. 2010.

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