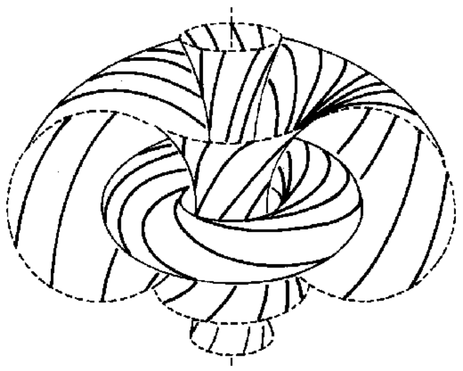
Homotopy Type Theory: Synthetic Homotopy Theory and Proof Verification

Max Blans

July 6, 2019

Bachelor thesis Mathematics and Computer Science

Supervisors: dr. Benno van den Berg, dr. Jasmin Blanchette



Informatics Institute
Korteweg-de Vries Institute for Mathematics
Faculty of Sciences
University of Amsterdam

Abstract

Homotopy type theory is a foundational system for mathematics that lies on the intersection of homotopy theory, logic, category theory, and computer science. Sets are replaced by types, which have the structure of spaces in the sense of homotopy theory, making it possible to use the theory as a language for synthetic homotopy theory.

In the first part of this thesis, we give an introduction to homotopy type theory, introducing concepts such as path induction, the univalence axiom, and higher inductive types. Then we apply the theory to give synthetic proofs of two classic theorems from homotopy theory: $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ and the Freudenthal suspension theorem.

One advantage of synthetic homotopy theory is that it lends itself better to formal proof verification than its classical counterpart. In the second part of the thesis, we explain how proofs from homotopy type theory can be formalized using the proof assistant Agda. We then discuss formalizations we made of the two classical theorems from the first part of the thesis.

Title: Homotopy Type Theory:
Synthetic Homotopy Theory and Proof Verification
Author: Max Blans, maxblans@gmail.com, 10671196
Supervisors: dr. Benno van den Berg, dr. Jasmin Blanchette
Second graders: prof. dr. Yde Venema, dr. Inge Bethke
End date: July 6, 2019

Informatics Institute
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
`http://www.ivi.uva.nl`

Korteweg-de Vries Institute for Mathematics
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
`http://www.kdvi.uva.nl`

Contents

1. Introduction

What does it mean for two mathematical objects to be equal? At first glance, this question seems to have an unambiguous answer: all mathematical objects are encoded as sets, and two sets are equal when they have the same elements. However, there are some mathematical theories for which this is too strong a notion of equality. One such theory is the branch of topology called homotopy theory. Here, the notion of equality is almost completely replaced by that of homotopy. For example, two maps are considered the same if they are homotopic, and two topological spaces are considered the same if there is a map between them which has an inverse up to homotopy. At the same time, homotopy theory is still built on set theoretic foundations, and one must be careful to distinguish between set theoretic equality and equality up to homotopy.

Homotopy type theory is an alternative to set theory in which equality behaves more like equality up to homotopy. Sets are replaced by types. Like sets, these can have elements. But unlike in set theory, when A is a type and x and y are elements of A, the expression $x = y$ again denotes a type. We can think of this type as consisting of the paths from x to y. When p and q are paths from x to y, we can again form the type $p = q$, which we think of as the type of homotopies from p to q. This can be repeated *ad infinitum*. In this way, types get the structure of a space in the sense of homotopy theory.

Since fundamental concepts from homotopy theory are primitives in the foundational system of homotopy type theory, it is possible to use homotopy type theory to give a synthetic development of homotopy theory. Here, the term synthetic is used in the sense of synthetic plane geometry, in which the primitive objects are figures such as lines and circles, as opposed to analytic geometry, where all geometric concepts can be explained in terms of coordinates. It is the goal of the first part of this thesis to introduce homotopy type theory and show how it can be used as a language for synthetic homotopy theory by giving synthetic proofs of two classical theorems from homotopy theory: $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ and the Freudenthal suspension theorem.

The first of these theorems is covered in any introductory course on algebraic topology, and is the most basic result in homotopy theory. In the classical proof, the key ingredient is the universal covering space of the circle. The interesting part of the synthetic proof is the way in which this construction is replaced by an analog from homotopy type theory. The Freudenthal suspension theorem has to do with a certain operation on spaces, called the suspension. It shows that under the right conditions, there is a close connection between homotopy-theoretic properties of a space and those of its suspension. The theorem has applications in the computation of homotopy groups of spaces, which are higher-dimensional generalizations of the fundamental group.

Homotopy type theory also has connections to computer science. A proof assistant is a programming language for writing proofs and verifying their correctness. A proof written in a proof assistant is called a formalization. An advantage of synthetic homotopy theory is that it lends itself better to formalization than its classical counterpart. In the second part of this thesis, we introduce the proof assistant Agda, which can be used to formalize proofs from homotopy type theory. We then discuss formalizations we made of the proofs of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ and the Freudenthal suspension theorem.

The following summary gives a more detailed description of the contents of each chapter of this thesis. In chapter 2, we explain the basics of homotopy type theory, introducing concepts as identity types, path induction, the univalence axiom, and higher inductive types. Our aim is to quickly cover all prerequisites for synthetic homotopy theory. Chapter 3 starts with a rapid treatment of n-types and homotopy groups, after which we give a synthetic proof of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$. Chapter 4 is concerned with the Freudenthal suspension theorem. After introducing the concepts of connectedness and suspensions, the theorem is stated and proved. In the final section of this chapter, we give some applications of this theorem to the computation of homotopy groups of spheres. The second part of the thesis starts with chapter 5, in which we explain how homotopy type theory can be encoded in Agda. We also introduce the HoTT-Agda library, which contains formalizations of many basic definitions and propositions from homtopy type theory. In chapters 6 and 7, we discuss our formalizations of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ and the Freudenthal suspension theorem. The full code of these formalizations is contained in appendices A and B.

2. Homotopy Type Theory

Homotopy type theory is a foundational system for mathematics, just like set theory. This chapter gives an informal introduction to the theory. We develop all that is necessary to begin the study of synthetic homotopy theory, which is the subject of the next two chapters.

2.1. Homotopy type theory versus set theory

Most mathematicians have spent their entire mathematical lives in the world of set theory. It can therefore take some time getting used to homotopy type theory (which we will often refer to as just type theory). To make this transition as seamless as possible, we begin by pointing out the differences between type theory and set theory.

Superficially, type theory and set theory look very similar. The central objects of type theory are called **types**. Just like sets, types can contain **elements**. If we have a type A and a is an element of A, we denote this by $a : A$. But here already we run into a subtle difference between the two theories. In set theory, given any two sets x and y, we can consider the proposition $x \in y$. Moreover, a set can be contained in various different sets. This is not the case in type theory. If $a : A$, the type A is an inherent property of a, and a cannot be contained in any other type.

We now come to the first big difference between type theory and set theory. In set theory, we can consider propositions about sets, which are sentences in first order logic. These propositions can be proved by applying deduction rules to other propositions that have already been proved or were postulated as axioms. So we see that set theory consists of two levels: there are the sets themselves, and there is the language of first order logic which we use to state and prove propositions about sets. In type theory, there is only one level: all propositions are themselves types.

There are only two statements we can make in type theory that are not propositions (and therefore not types). One is the aforementioned $a : A$. The other is called **judgmental equality** and is denoted $b \equiv c$ for elements b, c of some type A. It signifies that b and c are equal by definition. More precisely, it means that if we have expressions E and F for b and c respectively, and we expand all definitions occuring in these expressions, we end up with the same expression. The reason we write $b \equiv c$ instead of $b = c$ is that the latter expression *does* in fact denote a type. This **identity type** and the distinction between judgmental and propositional equality take center stage in homotopy type theory, and we will start exploring this in depth in section 2.5.

These differences between homotopy type theory and set theory lead to different ways of doing mathematics in these foundational systems. In set theory, we prove propositions by applying deduction rules to propositions we already know are true. In type theory we prove propositions by applying rules to construct elements of given types or construct new types from old ones. We will now proceed to introduce a number of basic constructions of types, starting with function types.

2.2. Function types

Given types A and B, we can construct the **function type** $A \to B$ of functions from A to B. An element $f : A \to B$ is called a **function** from A to B. (Note that the notation $f : A \to B$ from set theory gets a slightly different interpretation, even though the intuitive meaning is the same.) Given an element $x : A$ and a function $f : A \to B$, we can apply f to x to obtain an element $f(x) : B$.

One way to construct functions is by **λ-abstraction**. If we have for all $x : A$ an expression $\Phi_x : B$, which may depend on x, we can construct a function in $A \to B$ which we denote by $\lambda x.\Phi_x$. This construction is subject to the following relations:

- For $a : A$, we have $(\lambda x.\Phi_x)(a) \equiv \Phi_a$.
- Given a function $f : A \to B$, we have $(\lambda x.f(x)) \equiv f$.

Let us give some examples. For any $b : B$, we get a constant function $(\lambda x.b) : A \to B$, which sends every element of A to b. For every type A, we also have an identity function $\mathrm{id}_A : A \to A$, defined as $\mathrm{id}_A :\equiv \lambda a.a$. For types A, B, C and functions $f : A \to B$ and $g : B \to C$, we can compose them to obtain $g \circ f : A \to C$, defined as $(g \circ f) :\equiv \lambda x.g(f(x))$.

We will often define a function f of the form $\lambda x.\Phi_x$ by writing $f(x) :\equiv \Phi_x$.

Functions that take multiple arguments can be defined by means of currying. For example, a function that takes an argument from the type A and another argument from the type B and returns an element from type C can be regarded as a function of type $A \to (B \to C)$. We will omit the parentheses. For $a : A$ and $b : B$ and $f : A \to B \to C$, we will often write $f(a, b)$ instead of $f(a)(b)$.

The theory becomes a lot more flexible by also allowing functions that take values in different types, depending on the argument. In order to define such a function, we need the notion of a **type universe**. This is simply a type containing other types. Throughout this thesis, we will assume that all types we care about are contained in a single type universe $\mathcal{U}$. [1] A **family of types** indexed by a type A is simply a function $P : A \to \mathcal{U}$. This means that for every $x : A$ we get a type $P(x)$.

We are now in the position to define the **dependent function type** (also known as **$\prod$-type**) of a type family $P : A \to \mathcal{U}$, which is denoted $\prod_{(x:A)} P(x)$. An element $f : \prod_{(x:A)} P(x)$ is called a **dependent function**. It is a function with the property that $f(a) : P(a)$ for $a : A$. Dependent functions can be constructed just as regular functions: given a formula Φ_x depending on $x : A$ such that $\Phi_a : P(a)$ for all $a : A$, we get a function

[1] We have omitted some details here. If we were to do it properly, we would work with a sequence of nested universes $\mathcal{U}_1 \subset \mathcal{U}_2 \subset \cdots$ to avoid Russell-style paradoxes.

$f : \prod_{(x:A)} P(x)$ with $f(a) \equiv \Phi_a$. We can also use λ-abstraction to define dependent functions, which works exactly the same as for regular functions. Note that if P is a constant type family (which means $P(x) \equiv B$ for all $x : A$, where $B : \mathcal{U}$), the type $\prod_{(x:A)} P(x)$ is just the function type $A \to B$.

2.3. Inductive types

Many types can be constructed by following a certain pattern. Such a type is defined by specifying some canonical elements of the type, giving rules for constructing (dependent) functions out of the type, and stipulating what happens when one applies these functions to the canonical elements. Types constructed according to this scheme are called **inductive types**.

To show what such a definition concretely looks like, we will illustrate it by giving an inductive definition of the type $\mathbb{N}$ of natural numbers.

- The **introduction rules** (also called **constructors**) tell you how to obtain the canonical elements of an inductive type. For $\mathbb{N}$ there are two: $0 : \mathbb{N}$ and $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. These constructors express that all elements of $\mathbb{N}$ should either be 0 or else be obtained by repeatedly taking successors, starting with 0.

- The **recursion principle** makes it possible to construct functions out of an inductive type. The recursion principle for $\mathbb{N}$ says that if we have a type A, an element $a_0 : A$ and a function $a_s : \mathbb{N} \to A \to A$, we get a function $f : \mathbb{N} \to A$, satisfying

$$f(0) :\equiv a_0, \quad f(\mathsf{succ}(n)) :\equiv a_s(n, f(n)).$$

- The **induction principle** gives a way of defining dependent functions out of an inductive type. For $\mathbb{N}$, this principle states that given a type family $A : \mathbb{N} \to \mathcal{U}$, an element $a_0 : A(0)$ and a dependent function $a_s : \prod_{(n:\mathbb{N})} A(n) \to A(\mathsf{succ}(n))$ [2], we obtain a dependent function $f : \prod_{(n:\mathbb{N})} A(n)$, such that

$$f(0) :\equiv a_0, \quad f(\mathsf{succ}(n)) :\equiv a_s(n, f(n)).$$

  Note that the recursion principle is a special case of the induction principle in the case of a constant type family.

- The **computation rule** specifies how functions constructed by means of the recursion or induction principle act when they are applied to elements constructed by using introduction rules. We have already explained this rule for $\mathbb{N}$ by specifying $f(0)$ and $f(\mathsf{succ}(n))$ for functions constructed by recursion or induction.

One should think of an inductive type as a type freely generated by these rules. For example, the rules for $\mathbb{N}$ express that it is the simplest type containing 0 and a successor operator, such that we can define functions out of it by natural induction.

We now proceed to give definitions of a number of basic inductive types.

[2] An expression of the form $\prod_{(a:A)} B \to C$ should always be read as $\prod_{(a:A)}(B \to C)$.

Definition 2.1. The **empty type**, denoted $\mathbf{0}$, is an inductive type without constructors. The recursion principle for the empty type says that for any type $A : \mathcal{U}$, we get a function $f : \mathbf{0} \to A$. The induction principle is similar: given any type family $P : \mathbf{0} \to \mathcal{U}$, we get a dependent function $f : \prod_{(x : \mathbf{0})} P(x)$.

There exists also a type which has exactly one element.

Definition 2.2. The **unit type**, denoted $\mathbf{1}$, is an inductive type with one constructor, written $\star : \mathbf{1}$. The recursion principle for the unit type says that for any type $A : \mathcal{U}$, and any element $a : A$, we can construct a function $f : \mathbf{1} \to A$, satisfying $f(\star) \equiv a$. Given a type family $P : \mathbf{1} \to \mathcal{U}$ and an element $a : P(\star)$, the induction principle for the unit type yields a function $f : \prod_{(x : \mathbf{1})} P(x)$ such that $f(\star) \equiv a$.

It is also possible to construct cartesian products of types inductively.

Definition 2.3. Given $A, B : \mathcal{U}$, we can construct the **cartesian product** of A and B, denoted $A \times B$. This is an inductive type with one constructor, $(-, -) : A \to B \to A \times B$. The recursion principle says that given a type C and a function $g : A \to B \to C$, we get a function $f : A \times B \to C$ satisfying $f((a, b)) :\equiv g(a)(b)$. Given $C : A \times B \to \mathcal{U}$, and a dependent function $g : \prod_{(a : A)} \prod_{(b : B)} C((a, b))$, the induction principle allows us to construct a dependent function $f : \prod_{(x : A \times B)} C(x)$, such that $f((a, b)) :\equiv g(a)(b)$.

We will always write $f(a, b) :\equiv f((a, b))$. The recursion and induction principles express that to define a function on $A \times B$, it is enough to define it on the canonical elements (a, b). With the recursion principle, we can define projections $p_1 : A \times B \to A$, $p_2 : A \times B \to B$. For example, to define p_1, we apply the recursion principle to the function $g : A \to B \to A$, defined by $g(a)(b) :\equiv a$.

The next inductive type we define is the type theoretic analog of the disjoint union of two sets.

Definition 2.4. For any $A, B : \mathcal{U}$ there exists an inductive type $A + B$, called the **coproduct** of A and B. It has two constructors, $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$. Given a type C and functions $g : A \to C$ and $h : B \to C$, the recursion principle makes it possible to construct a function $f : A + B \to C$ such that $f(\mathsf{inl}(a)) :\equiv g(a)$ and $f(\mathsf{inr}(b)) :\equiv h(b)$.

There is also an inductive principle: given $C : (A + B) \to \mathcal{U}$ and dependent functions $g_0 : \prod_{(a : A)} C(\mathsf{inl}(a))$ and $g_1 : \prod_{(b : B)} C(\mathsf{inr}(b))$, we get $f : \prod_{(x : A+B)} C(x)$, satisfying $f(\mathsf{inl}(a)) :\equiv g_0(a)$ and $f(\mathsf{inr}(b)) :\equiv g_1(b)$.

We use the coproduct to define the type of booleans $\mathbf{2} :\equiv \mathbf{1} + \mathbf{1}$. We will write $0_\mathbf{2} :\equiv \mathsf{inl}(\star)$ and $1_\mathbf{2} :\equiv \mathsf{inr}(\star)$.

The final inductive type we introduce is a generalization of the cartesian product, in the same way the dependent function type is a generalization of the function type. Its elements are pairs, where the type of the second component depends on the first component.

Definition 2.5. Given a type A and type family $B : A \to \mathcal{U}$, there exists an inductive type called the **dependent pair type** or **$\sum$-type** and written as $\sum_{(a : A)} B(a)$. It has one constructor:

$$(-,-) : \prod_{(a:A)} \Big(B(a) \to \sum_{(x:A)} B(x) \Big).$$

The recursion principle says that given a function $g\colon \prod_{(a:A)} B(a) \to C$, we get a function $f\colon \sum_{(a:A)} B(a) \to C$ satisfying $f((a,b)) :\equiv g(a)(b)$.

The induction principle says that for $C\colon \sum_{(a:A)} B(a) \to \mathcal{U}$, we can define a function $f\colon \prod_{(p:\sum_{(a:A)} B(a))} C(p)$ by providing a function

$$g\colon \prod_{(a:A)} \prod_{(b:B(a))} C((a,b)).$$

The function f will satisfy $f((a,b)) :\equiv g(a)(b)$.

Concretely, the canonical elements of the dependent pair type $\sum_{(a:A)} B(a)$ are pairs (a,b), where $a\colon A$ and $b\colon B(a)$. We will again adopt the convention of writing $f(a,b) :\equiv f((a,b))$ for functions f out of a $\sum$-type. Note that we can recover the cartesian product type by taking the dependent sum type of a constant type family. Again it is possible to define projection functions.

2.4. Propositions as types

Now that we have defined the most basic inductive types, we can explain how propositions are incarnated by types. Any type A can be regarded as a proposition. We consider this propositions to be true if there exists an element $a\colon A$, in which case we call A **inhabited**. If A is not inhabited, we regard the proposition represented by A as false. To prove a proposition A, one has to exhibit an element $a\colon A$.

Quite remarkably, it is possible to mimic all notions from first order logic with type theoretic constructions. This is called the **Curry-Howard correspondence**. To give an example, suppose we have propositions A and B. To prove these propositions is the same as giving inhabitants $a\colon A$ and $b\colon B$. But this is the same as giving the element $(a,b)\colon A \times B$. We find that conjunction of propositions corresponds to the cartesian product of types. Another example is implication. We would like to say that A implies B if we can find an inhabitant of B whenever we have an inhabitant of A. This corresponds to the notion of a function from A to B, so the type $A \to B$ embodies the proposition "A implies B". The entire correspondence is given in table 2.1.

The last three entries in this table require some further explanation. Since a type family $P\colon A \to \mathcal{U}$ associates to each element $a\colon A$ a type $P(a)$, we can consider each $P(x)$ to be some proposition, perhaps expressing a property of x. In this sense, P is the analogon of a predicate. Providing a function $f\colon \prod_{(a:A)} P(a)$ then yields an inhabitant $f(a)\colon P(a)$, for all $a\colon A$, thus proving that $P(a)$ holds for all a. On the other hand, an element $(a,b)\colon \sum_{(x:A)} P(a)$ gives an inhabitant $b\colon P(a)$ for *one* term $a\colon A$, showing there exists *some a* such that $P(a)$.

Since $\forall$-statements correspond to $\prod$-types, proving a statement about all elements in a certain type boils down to constructing a dependent function out of that type. This is

Table 2.1.: The Curry-Howard correspondence

First-order logic	Type theory
proposition ϕ	type A
$\bot$	$\mathbf{0}$
$\top$	$\mathbf{1}$
$\phi \wedge \psi$	$A \times B$
$\phi \vee \psi$	$A + B$
$\phi \implies \psi$	$A \to B$
$\phi \iff \psi$	$(A \to B) \times (B \to A)$
$\neg\phi$	$A \to \mathbf{0}$
predicate p	type family $P : A \to \mathcal{U}$
$\forall x \in S,\ p(x)$	$\prod_{(a:A)} P(a)$
$\exists x \in S,\ p(x)$	$\sum_{(a:A)} P(a)$

then done by invoking an appropriate induction principle. For example, the induction principle for the natural numbers says that we can construct a dependent function $f : \prod_{(n:\mathbb{N})} P(n)$ for some $P : \mathbb{N} \to \mathcal{U}$ by supplying an element $c_0 : P(0)$ and a function $c_s : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}(n))$. Translating into logic, to prove that $P(n)$ holds for all $n : \mathbb{N}$, it is enough to show $P(0)$ holds, and that for all $n : \mathbb{N}$, $P(n)$ implies $P(\mathsf{succ}(n))$. We find that we have recovered the axiom of natural induction.

A salient feature of the logic of homotopy type theory is that it is constructive, which means that the law of the excluded middle (LEM) is not assumed to hold. In type theory, LEM would say that for every type A, the type $A + \neg A$ is inhabited. This is equivalent to saying that $\neg\neg A \to A$ is inhabited. It is possible to add LEM to the theory. However, it can be proved that LEM contradicts the univalence axiom (section 2.9), which would defeat the purpose of homotopy type theory. There is a weaker version of LEM which does not contradict this axiom, but, in general, not assuming LEM is regarded as desirable due to the constructive nature of type theory. In this thesis, we will not assume that any form of LEM holds.

Since a type can have more than one element, a proposition can have multiple inhabitants. This phenomenon has no counterpart in first order logic, but is of primary importance in homotopy type theory. A good way of thinking about it is that we can have multiple different proofs of the same statement, and that these proofs can carry more information than just asserting the veracity of a proposition. But there is more to it, as we will see in the next section.

2.5. Identity types

The most important types in homotopy type theory are arguably the identity types. For any two elements $x, y : A$ of a given type A, we have an identity type, which we write as $x =_A y$ (we will often drop the subscript and leave the type implicit). This type embodies the proposition "x is equal to y". We will refer to this notion of equality as **propositional equality** to distinguish it from judgmental equality. There is one introduction rule for identity types: given a type A and an element $x : A$, we get an element

$$\mathsf{refl}_x : x =_A x,$$

reflecting that every element is "trivially" equal to itself. It also implies that elements which are judgmentally equal are propositionally equal.

In the previous section, we raised the question of how to interpret multiple inhabitants of a proposition. We will now consider this question for identity types. To answer it, we will give a homotopy-theoretic interpretation of type theory (which will explain the name homotopy type theory).

We will think of any type A as a (topological) space. The elements of A are its points. For points $x, y : A$, we will interpret elements of $x =_A y$ as **paths** between the points x and y. So different elements $p, q : x =_A y$ correspond to different paths from x to y, as illustrated in figure 2.1. Since p and q are themselves elements of some type, we also can consider the type $p = q$. Its inhabitants are interpreted as **homotopies** between the paths p and q. We will sometimes call these **2-paths** or **2-dimensional paths**. There is of course no reason to stop here. Given two homotopies $\alpha, \beta : p = q$, we can form $\alpha = \beta$, which is the type of homotopies *between* the homotopies α and β. This can be continued indefinitely, giving rise to the notion of **n-paths** or **n-dimensional paths**.

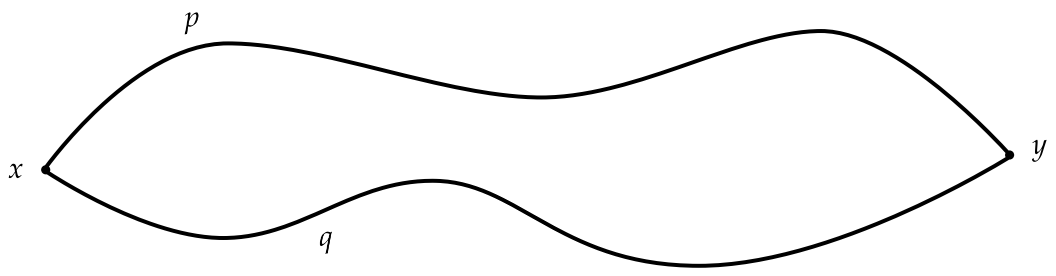


Figure 2.1.: Two paths from x to y.

Note that by now we have given *three* different interpretations of types: set-like objects, propositions, and spaces. These interpretations coexist, but we will later see that there are types that behave more like sets or propositions than other types.

The induction principle for identity types says that to construct a dependent function on all paths in a type, it is enough to specify it on all trivial paths. Or, equivalently, to prove something for all paths in a type, it is enough to prove it for all trivial paths in the type. We now state this more formally. Suppose we are given a type family

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

and a dependent function $g\colon \prod_{(x:A)} P(x, x, \mathsf{refl}_x)$. Then we can construct a dependent function

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

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

There is also a based version of path induction, which we will sometimes need. It says that if we fix $a : A$ and have a type family

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

and an element $c\colon P(A, \mathsf{refl}_a)$, we obtain a dependent function

$$f\colon \prod_{(x:A)} \prod_{(p:a=x)} P(x, p).$$

It turns out that path induction and based path induction are equivalent. (For a proof, see [11, pp. 68–71].)

By definition, propositional equality is reflexive because we always have the element $\mathsf{refl}_x : x = x$. Topologically, we think of $\mathsf{refl}_x : x = x$ as the "trivial" path (standing still at x). We would also like equality to be symmetric and transitive. These notions correspond to inversion and concatenation of paths. We will prove them now as a first application of path induction.

Proposition 2.6. *Given any type A, elements $x, y, z : A$, there are functions*

1. $(x =_A y) \to (y =_A x), \quad p \mapsto p^{-1}$,
2. $(x =_A y) \times (y =_A z) \to (x =_A z), \quad (p, q) \mapsto p \cdot q$,

such that

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

Proof.

1. Consider the type family

$$P\colon \prod_{(x,y:A)} (x =_A y) \to \mathcal{U}, \quad P(x, y, p) :\equiv (y =_A x).$$

We wish to construct a dependent function

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

By path induction, it is enough to give a function $g\colon \prod_{(x:A)} P(x, x, \mathsf{refl}_x)$. But the type $P(x, x, \mathsf{refl}_x)$ is just $x =_A x$, so we set $g(x) :\equiv \mathsf{refl}_x$ for all $x : A$. This completes the construction of f. We will write $p^{-1} :\equiv f(x, y, p)$, keeping the arguments x and y implicit. Note that we have

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

as desired.

2. In this case, the appropriate type family is

$$P\colon \prod_{(x,y,z\colon A)} \prod_{(p\colon x=_A y)} \prod_{(q\colon y=_A z)} \mathcal{U}, \quad P(x,y,z,p,q) :\equiv (x =_A z).$$

   To construct a dependent function f into this type family, we apply path induction twice. It now suffices to construct a dependent function

$$g\colon \prod_{(x\colon A)} P(x,x,x,\mathsf{refl}_x,\mathsf{refl}_x).$$

   But $P(x,x,x,\mathsf{refl}_x,\mathsf{refl}_x) \equiv (x =_A x)$, so we set $g(x) :\equiv \mathsf{refl}_x$, and are done. We will write $p \cdot q :\equiv f(x,y,z,p,q)$, suppressing the arguments x, y, z. Again we have by construction that

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

   This completes the proof.

□

Path concatenation and inversion satisfy a number of nice properties.

Proposition 2.7. *Suppose we are given a type $A\colon\mathcal{U}$, elements $x, y, z, w\colon A$, and paths $p\colon x =_A y$, $q\colon y =_A z$, $r\colon z =_A w$. Then we have the following propositional equalities:*

1. $\mathsf{refl}_x \cdot p = p$ *and* $p \cdot \mathsf{refl}_y = p$,
2. $p \cdot p^{-1} = \mathsf{refl}_x$ *and* $p^{-1} \cdot p = \mathsf{refl}_y$,
3. $(p^{-1})^{-1} = p$,
4. $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.

Proof. These are all simple applications of path induction. We will do (3), the others can be found in [11, p. 86].

Consider the type family

$$P\colon \prod_{(x,y\colon A)} (x =_A y) \to \mathcal{U}, \quad P(x,y,p) :\equiv ((p^{-1})^{-1} = p).$$

To construct a dependent function into this type family, it is enough to provide $c_x\colon P(x,x,\mathsf{refl}_x)$ for all $x\colon A$ by path induction. But we have

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

and therefore $P(x,x,\mathsf{refl}_x) \equiv (\mathsf{refl}_x = \mathsf{refl}_x)$. So we set $c_x :\equiv \mathsf{refl}_{\mathsf{refl}_x}$. □

From now on, we will present induction proofs less formally, often leaving the type family implicit.

Topologically, we interpret a function $f : A \to B$ as a continuous map between the spaces A and B. We would like functions to respect equalities, or, from the homotopy-theoretic viewpoint, we would like to be able to push forward paths along continuous maps.

Proposition 2.8. *Suppose $f : A \to B$. For any pair of points $x, y : A$, we get a function*

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

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

Proof. By path induction, we only have to define ap_f on trivial paths, in which case we set $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$. □

We will often write $f(p)$ instead of $\mathsf{ap}_f(p)$. It turns out that ap_f preserves the operations on paths and functions we have seen so far.

Proposition 2.9. *Suppose we are given functions $f : A \to B$ and $g : B \to C$ and paths $p : x =_A y$ and $q : y =_A z$. Then,*

1. $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$,
2. $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$,
3. $\mathsf{ap}_g(\mathsf{ap}_f(p)) = \mathsf{ap}_{g \circ f}(p)$,
4. $\mathsf{ap}_{\mathrm{id}_A}(p) = p$.

Proof.

1. By induction, we can assume p and q are both refl_x. Then $p \cdot q \equiv \mathsf{refl}_x$. We already know that $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$. Combining this, we get

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

   So $\mathsf{ap}_f(p \cdot q)$ and $\mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$ are judgmentally equal. Then they are also propositionally equal.

2. We can assume $p \equiv \mathsf{refl}_x$ by path induction. We then have

$$\begin{aligned} \mathsf{ap}_f(p^{-1}) &\equiv \mathsf{ap}_f(\mathsf{refl}_x^{-1}) \equiv \mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)} \\ &\equiv \mathsf{refl}_{f(x)}^{-1} \equiv \mathsf{ap}_f(\mathsf{refl}_x)^{-1} \equiv \mathsf{ap}_f(p)^{-1}. \end{aligned}$$

3. Again we can assume $p \equiv \mathsf{refl}_x$. Hence,

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

   But $g(f(x)) \equiv (g \circ f)(x)$, which implies

$$\mathsf{ap}_g(\mathsf{ap}_f(p)) \equiv \mathsf{refl}_{(g \circ f)(x)} \equiv \mathsf{ap}_{g \circ f}(\mathsf{refl}_x) \equiv \mathsf{ap}_{g \circ f}(p).$$

4. Again it is enough to prove it for $p \equiv \mathsf{refl}_x$. In this case we have

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

□

2.6. Transport

The homotopy-theoretic interpretation of type theory also gives a new way of looking at type families. We will think of a type family $P\colon A \to \mathcal{U}$ as a space lying over and projecting onto A, such that the fiber over $a\colon A$ is $P(a)$. In classical topology, this corresponds to the notion of a fibration. We think of elements of $\prod_{(a\colon A)} P(x)$ as sections into this space.

If we have a type family $P\colon A \to \mathcal{U}$ and a path $p\colon x =_A y$, we would like to be able to relate the fibers $P(x)$ and $P(y)$ in some way. This is realized by the notion of **transport**.

Proposition 2.10. *Given a type family $P\colon A \to \mathcal{U}$ and a path $p\colon x =_A y$. Then exists a function*

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

such that $\mathsf{transport}^P(\mathsf{refl}_x, -) \equiv id_{P(x)}$.

Proof. We need to construct a dependent function

$$f\colon \prod_{(x,y\colon A)} \prod_{(p\colon x=y)} (P(x) \to P(y)).$$

This is a dependent function of the type family

$$Q\colon \prod_{(x,y\colon A)} (x =_A y) \to \mathcal{U}, \quad Q(x,y,p) \equiv (P(x) \to P(y)).$$

By path induction, it is enough to provide $c\colon \prod_{(x\colon A)} Q(x,x,\mathsf{refl}_x)$. But we have $Q(x,x,\mathsf{refl}_x) \equiv (P(x) \to P(x))$, so we set $c(x) \equiv \mathrm{id}_{P(x)}$.

We now set $\mathsf{transport}^P(p, -) :\equiv f(x,y,p)$, leaving x and y implicit. We then have

$$\mathsf{transport}^P(\mathsf{refl}_x, -) \equiv f(x,x,\mathsf{refl}_x) \equiv c(x) \equiv \mathrm{id}_{P(x)},$$

as desired. □

The function $\mathsf{transport}^P(p, -)$ is called the transport function of the type family P and path p. We will write $p_* :\equiv \mathsf{transport}^P(p, -)$ if the type family is understood.

We now return to the interpretation of a type family $P\colon A \to \mathcal{U}$ as a fibration. Given a path $p\colon x =_A y$ and elements $u\colon P(x)$ and $v\colon P(y)$, we would like to be able to speak of a path between u and v lying over p in the fibration. Of course, this cannot be taken too literally, since u and v inhabit distinct types. However, because of transport, we have a means of comparing the fibers $P(x)$ and $P(y)$. We therefore define a path lying over p between u and v to be a path in $p_*(u) = v$. We will sometimes use the following notation for the type of paths from u to v lying over p:

$$(u =^P_p v) :\equiv (\mathsf{transport}^P(p,\ u) = v).$$

Just like we can push a path forward along a function to obtain a path in the domain, we can push a path forward along a dependent function to obtain a path lying over it.

Proposition 2.11. *Given a dependent function $f\colon \prod_{(a\colon A)} P(a)$, and elements $x, y\colon A$, we get*

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

Proof. By path induction, we may assume $p = \mathsf{refl}_x$. But in this case $f(x) \equiv f(y)$ and $p_* = \mathrm{id}_{P(x)}$, so we need to provide an element of $f(x) = f(x)$. We take $\mathsf{refl}_{f(x)}$. □

We finish this section by proving some properties of the transport function that we will need in the next chapter. First off, transporting in a fibration where all fibers are path spaces is just concatenation of paths.

Proposition 2.12. *Suppose we are given a type A and $a\colon A$. Then for any path $p\colon x_1 =_A x_2$, we have*

$$\begin{aligned}
\mathsf{transport}^{x\mapsto(a=x)}(p,\ q) &= q \cdot p && \textit{where } q\colon a =_A x_1,\\
\mathsf{transport}^{x\mapsto(x=a)}(p,\ q) &= p^{-1} \cdot q && \textit{where } q\colon x_1 =_A a,\\
\mathsf{transport}^{x\mapsto(x=x)}(p,\ q) &= p^{-1} \cdot q \cdot p && \textit{where } q\colon x_1 =_A x_2.
\end{aligned}$$

Proof. Since the proofs of the three identities are similar, we only do the first one.
Let us first parse the statement. We are given a type family

$$P\colon A \to \mathcal{U}, \quad P(x) :\equiv (a =_A x),$$

so $P(x_1) \equiv (a =_A x_1)$, $P(x_2) \equiv (a =_A x_2)$, and $p_*\colon (a =_A x_1) \to (a =_A x_2)$. Since $q\colon a =_A x_1$ and $q \cdot p\colon a = x_2$, the identity

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

is indeed well-typed.

Proving the identity requires a tricky application of path induction. We want to construct a dependent function

$$f\colon \prod_{(p\colon x_1=x_2)} \prod_{(q\colon a=x_1)} \mathsf{transport}^{x\mapsto(a=x)}(p,\ q) = q \cdot p.$$

By first applying path induction, we can assume $p \equiv \mathsf{refl}_{x_1}$. Now we can use based path induction to reduce to the case $q \equiv \mathsf{refl}_a$. But then $x_1 \equiv a$ and $p \equiv \mathsf{refl}_a$ as well. The statement we are trying to prove has now shrunk to

$$\mathsf{transport}^{x\mapsto(a=x)}(\mathsf{refl}_a,\ \mathsf{refl}_a) = \mathsf{refl}_a \cdot \mathsf{refl}_a.$$

The left hand side is just $\mathsf{refl}_{a,*}(\mathsf{refl}_a) \equiv \mathsf{refl}_a$, and on the right hand side we have $\mathsf{refl}_a \cdot \mathsf{refl}_a \equiv \mathsf{refl}_a$. We are now left with a judgmental equality and are therefore done. ☐

Transport also commutes with path concatenation.

Proposition 2.13. *Suppose we have a type family $P\colon A \to \mathcal{U}$ and paths $p\colon x =_A y$ and $q\colon y =_A z$. Then we have for $u\colon P(x)$ that*

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

Proof. By path induction, we can assume that $p \equiv q \equiv \mathsf{refl}_x$. But then we have $(p \cdot q)_*(u) \equiv (\mathsf{refl}_x)_*(u) \equiv \mathrm{id}_{P(x)}(u)$ and

$$q_*(p_*(u)) \equiv (\mathsf{refl}_x)_*((\mathsf{refl}_x)_*(u)) \equiv \mathrm{id}_{P(x)}(\mathrm{id}_{P(x)}(u)).$$

We see that both sides of the equality are reduced to u, and we are done. ☐

We now show that the transport function of any type family P can always be reduced to the transport function of the trivial type family $X \mapsto X : \mathcal{U} \to \mathcal{U}$.

Proposition 2.14. *Let A be a type and $B : A \to \mathcal{U}$ a type family. For any path $p\colon x =_A y$ and $u\colon B(x)$, we have*

$$\mathsf{transport}^B(p,\ u) = \mathsf{transport}^{X \mapsto X}(\mathsf{ap}_B(p),\ u).$$

Proof. By path induction on p, both sides of the equality reduce to u. ☐

2.7. Mere propositions and sets

We mentioned earlier that some types behave more like sets and propositions than others. We will now make this precise.

Definition 2.15. A type A is a **mere proposition** if the following proposition holds

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

We think of a mere proposition as a type that has at most one element (up to propositional equality). This is what we would expect from a proposition: it can either be true or false and contains no extra information.

Those propositions that are inhabited have a special name.

Definition 2.16. A type A is called **contractible** if the following proposition holds

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

The name suggests that we should interpret contractible types as spaces that are homotopy equivalent to a point.

We will now continue our comparison of type theory and set theory by giving a definition of a set from within type theory.

Definition 2.17. A type A is a **set** if for all $x, y \colon A$, $x =_A y$ is a mere proposition.

This definition corresponds to the intuition that a set should be a type in which elements can only be equal in one way.

One question still remains. If A is a mere proposition, we have for every $x, y \colon A$ that $x = y$. But for all we know, it might still be possible that there are $p, q \colon x = y$ that are *unequal*. It would even be conceivable that there are non-trivial higher paths. This would ruin our conception of a mere proposition as a type that can contain no more information than stating that something is true or false. Fortunately, the following theorem tells us that this cannot happen.

Theorem 2.18. *If A is a mere proposition, then A is a set.*

Proof. Since A is a mere proposition, we have $f \colon \mathsf{isProp}(A)$. Concretely, this means that given $x, y \colon A$, we have $f(x, y) \colon x = y$.

Now fix $x \colon A$ and consider $g(y) :\equiv f(x, y)$. This is a dependent function of the type family $y \mapsto (x = y)$. If we now take a path $p \colon y =_A z$, we get $\mathsf{apd}_g(p) \colon \mathsf{transport}^{y \mapsto (x=y)}(p, g(y)) = g(z)$. By Proposition 2.12, the left hand side is equal to $g(y) \cdot p$. Hence, $p = g(y)^{-1} \cdot g(z)$.

So suppose we are given $y, z \colon A$ and paths $p, q \colon y =_A z$. We then have

$$p = g(y)^{-1} \cdot g(z) = q,$$

showing that $y =_A z$ is a mere proposition. □

This theorem tells us that if A is a mere proposition and $x, y \colon A$, then $x = y$ is again a mere proposition. Applying the theorem again, we find that $x = y$ is also a set. By continuing in this way, we see that a mere proposition can have no non-trivial higher paths.

2.8. Quasi-inverses and equivalences

In many branches of mathematics, there is the notion of *isomorphism*, which formalizes the idea that two objects are structurally the same. In this section, we will introduce the notion of **equivalence** of types, which captures this idea in the context of homotopy type theory. To do so, we first need to define what it means for a map to have an inverse. We want such a map to only be an inverse up to homotopy.

Definition 2.19. Two functions $f, g \colon A \to B$ are homotopic if for all $x \colon A$, we have $f(x) = g(x)$. More formally, we write

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

We can now define what it means for a function to have an inverse.

Definition 2.20. Let $f : A \to B$ be a function. A **quasi-inverse** to f is a function $g : B \to A$ such that $f \circ g \sim \mathrm{id}_B$ and $g \circ f \sim \mathrm{id}_A$. The type of quasi-inverses to f is denoted by

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

It is tempting to define a function $f : A \to B$ to be an equivalence if it has a quasi-inverse $g : B \to A$. It turns out that it is better to make a slightly different definition. The reason for this is that f might have multiple different quasi-inverses. What we are looking for is a type $\mathsf{isequiv}(f)$, with the following properties:

- The type $\mathsf{isequiv}(f)$ is a mere proposition.
- The types $\mathsf{isequiv}(f)$ and $\mathsf{qinv}(f)$ are logically equivalent, that is, we have maps $\mathsf{isequiv}(f) \to \mathsf{qinv}(f)$ and $\mathsf{qinv}(f) \to \mathsf{isequiv}(f)$.

Assuming for now that such a type exists, we will write

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

if A and B are equivalent. We will often make a slight abuse of notation by writing $f : A \simeq B$ if f is an equivalence from A to B. Note that to show f is an equivalence, it is enough to provide a quasi-inverse by the second property. The simplest example of an equivalence is $\mathrm{id}_A : A \to A$, which is its own quasi-inverse.

It follows from the properties listed above that any two definitions of $\mathsf{isequiv}$ are equivalent. We will give a definition which depends on the concept of a fiber of a map.

Definition 2.21. If $f : A \to B$ and $y : B$, the type

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

is called the **fiber** of f over y.

The idea is that f is an equivalence if all fibers contain exactly one element, up to homotopy.

Definition 2.22. Given $f : A \to B$, we define

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

For proofs that $\mathsf{isequiv}$ indeed satisfies the properties listed above, we refer to [11, pp. 179–180].

It is not hard to prove that type equivalence is an equivalence relation on $\mathcal{U}$. In particular, if $f : A \simeq B$, the type $B \simeq A$ also has an inhabitant, which we will call f^{-1}, and which is a quasi-inverse to f.

2.9. The univalence axiom

With the notion of equivalence, we now have two ways of expressing that types A and B are "the same". On the one hand, we can say they are equivalent. But since $A, B : \mathcal{U}$, we can also form the type $A =_{\mathcal{U}} B$, and say they are propositionally equal. The univalence declares these notions to be equivalent.

A first application of the univalence axiom will be to show that there are in fact types that are not sets. As by now it has become clear, the possibility of having multiple paths between elements of a type is a central aspect of homotopy type theory. But we have not yet demonstrated a single type with non-homotopic paths between two points.

We start preparing the ground for the axiom by showing that a propositional equality can be turned into an equivalence.

Lemma 2.23. *Given types $A, B : \mathcal{U}$, we have a function*

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

such that a path $p : A =_{\mathcal{U}} B$ is sent to the function

$$\mathsf{transport}^{X \mapsto X}(p, -).$$

Proof. Given a path $p : A =_{\mathcal{U}} B$, we can transport along this path in the type family $X \mapsto X$, which corresponds to a function $p_* : A \to B$. We now only have to proof that this is an equivalence. By path induction, we can assume $p \equiv \mathsf{refl}_A$. But

$$(\mathsf{refl}_A)_* \equiv \mathrm{id}_A,$$

which is an equivalence. □

We can now state the univalence axiom.

Axiom 2.24 (Univalence)**.** The function $\mathsf{idtoequiv}$ is an equivalence. Hence we have

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

In particular, the univalence axiom gives us a quasi-inverse to $\mathsf{idetoequiv}$, which we call

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

Since

$$\mathsf{idtoequiv} \equiv \mathsf{transport}^{X \mapsto X} : (A =_{\mathcal{U}} B) \to (A \simeq B),$$

we have

$$\mathsf{ua}(\mathsf{transport}^{X \mapsto X}(p, -)) = p, \qquad \mathsf{transport}^{X \mapsto X}(\mathsf{ua}(f), x) = f(x).$$

Using the univalence axiom, it is possible to prove that propositional equality of functions is equivalent to homotopy of functions.

Theorem 2.25 (Function extensionality). *Given functions $f, g\colon A \to B$, we have the following equivalence of types:*

$$(f = g) \simeq (f \sim g).$$

Proof. See [11, pp. 188–190]. $\square$

In particular, we have functions

$$\mathsf{happly}\colon (f = g) \to (f \sim g), \quad \mathsf{funext}\colon (f \sim g) \to (f = g),$$

which are quasi-inverses to each other.

Proposition 2.26. *Let $f : A \simeq B$ and $g : B \simeq C$. The function* ua *has the following properties:*

1. $\mathsf{refl}_A = \mathsf{ua}(\mathrm{id}_A)$,
2. $\mathsf{ua}(f) \cdot \mathsf{ua}(g) = \mathsf{ua}(g \circ f)$,
3. $\mathsf{ua}(f)^{-1} = \mathsf{ua}(f^{-1})$.

Proof.

1. We have $\mathrm{id}_A \equiv \mathsf{transport}^{X \mapsto X}(\mathsf{refl}_A, -,)$ by definition of transport. So

$$\mathsf{ua}(\mathrm{id}_A) = \mathsf{ua}(\mathsf{idtoequiv}(\mathsf{refl}_A)) = \mathsf{refl}_A.$$

2. By Proposition 2.13 we have that $\mathsf{idtoequiv}(q) \circ \mathsf{idtoequiv}(p) = \mathsf{idtoequiv}(p \cdot q)$. Therefore

$$\begin{aligned}\mathsf{ua}(g \circ f) &= \mathsf{ua}(\mathsf{idtoequiv}(\mathsf{ua}(g)) \circ \mathsf{idtoequiv}(\mathsf{ua}(f))) \\ &= \mathsf{ua}(\mathsf{idtoequiv}(\mathsf{ua}(f) \cdot \mathsf{ua}(g))) = \mathsf{ua}(f) \cdot \mathsf{ua}(g).\end{aligned}$$

3. By the previous identity, we have that

$$\mathsf{ua}(f^{-1}) \cdot \mathsf{ua}(f) = \mathsf{ua}(f^{-1} \circ f).$$

It follows from function extensionality that $f^{-1} \circ f = \mathrm{id}_B$, so by applying $\mathsf{ap}_{\mathsf{ua}}$, we get

$$\mathsf{ua}(f^{-1}) \cdot \mathsf{ua}(f) = \mathsf{ua}(\mathrm{id}_B) = \mathsf{refl}_B.$$

By cancelling $\mathsf{ua}(f)$ on the right, we obtain the desired equality.

$\square$

We will now use the univalence axiom to prove that not every type is a set.

Theorem 2.27. *The type $\mathcal{U}$ is not a set.*

Proof. Consider $\mathbf{2} : \mathcal{U}$. We construct the function

$$f : \mathbf{2} \to \mathbf{2}, \quad 0_{\mathbf{2}} \mapsto 1_{\mathbf{2}}, \quad 1_{\mathbf{2}} \mapsto 0_{\mathbf{2}}.$$

This function is clearly its own quasi-inverse, so it is an equivalence. Moreover, f is not equal to $\mathsf{id}_{\mathbf{2}}$. Now consider $\mathsf{ua}(f) : \mathbf{2} =_{\mathcal{U}} \mathbf{2}$. I claim this is not equal to $\mathsf{refl}_{\mathbf{2}}$. For if this were the case, we would have

$$f = \mathsf{idtoequiv}(\mathsf{ua}(f)) = \mathsf{idtoequiv}(\mathsf{refl}_{\mathbf{2}}) = \mathsf{id}_{\mathbf{2}},$$

which is a contradiction. Hence $\mathbf{2} =_{\mathcal{U}} \mathbf{2}$ is not a mere proposition. We conclude that $\mathcal{U}$ is not a set. $\square$

2.10. Higher inductive types

It is possible to construct elements of inductive types by means of introduction rules. We will now extend this idea by also allowing for inductive types which have introduction rules for paths and higher paths. These types will be called **higher inductive types**. For such a type, we refer to the introduction rules for elements as **point constructors** and to the introduction rules for (higher) paths as **path constructors**.

Higher inductive types are indispensable for synthetic homotopy theory, because they allow for simple constructions of interesting spaces. These constructions are similar to those of CW-complexes in classical homotopy theory.

As an example, we will define the **circle** $\mathbb{S}^1$. This is a higher inductive type with one point constructor $\mathsf{base} : \mathbb{S}^1$ and one path constructor $\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$. The recursion principle for $\mathbb{S}^1$ is simple: given a type A, an element $a : A$ and a path $\ell : a = a$, we get a function $f : \mathbb{S}^1 \to A$ satisfying $f(\mathsf{base}) \equiv a$ and $\mathsf{ap}_f(\mathsf{loop}) = \ell$.

Given $P : \mathbb{S}^1 \to \mathcal{U}$, an element $b : P(\mathsf{base})$ and a path $\ell : b =^P_{\mathsf{loop}} b$ lying over loop, the induction principle of $\mathbb{S}^1$ allows for defining a function $f : \prod_{(x:\mathbb{S}^1)} P(x)$ such that

$$f(\mathsf{base}) \equiv b, \quad \mathsf{apd}_f(\mathsf{loop}) = \ell.$$

How do we know that we have defined a type that is actually interesting? To put it in another way, how do we know that $\mathbb{S}^1$ is not a set, in which case we would have $\mathsf{loop} = \mathsf{refl}_{\mathsf{base}}$? This is provided for by the univalence axiom. For if $\mathsf{loop} = \mathsf{refl}_{\mathsf{base}}$, we can show there exist no types with non-trivial loops: take a type A, an element $a : A$ and a loop $\ell : a = a$. The recursion principle for $\mathbb{S}^1$ then gives a function $f : \mathbb{S}^1 \to A$ such that $f(\mathsf{base}) \equiv a$ and $f(\mathsf{loop}) = \ell$. But then

$$\ell = f(\mathsf{loop}) = f(\mathsf{refl}_{\mathsf{base}}) \equiv \mathsf{refl}_a.$$

However, we have seen that the univalence axiom implies $\mathcal{U}$ has a non-trivial loop at $\mathbf{2}$, so this cannot be the case.

We will see many more examples of higher inductive types in the next chapter.

3. The Fundamental Group of $\mathbb{S}^1$

In this chapter we will take up the study of synthetic homotopy theory by proving that $\pi_1(\mathbb{S}^1) = \mathbb{Z}$.

3.1. n-types and truncations

Roughly speaking, an n-type is a type that has no non-trivial higher paths above dimension n. It is a generalization of the concepts of mere proposition and set that were introduced in the previous chapter. We give a recursive definition.

Definition 3.1. Let $n \geq 2$ be an integer. For $X : \mathcal{U}$, let

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

We call X an **n-type** or **n-truncated** if $\mathsf{is}{-}n{-}\mathsf{type}(X)$ holds.

We see that a -2-type is the same as a contractible type, a -1-type is the same as a mere proposition and a 0-type is the same as a set. We already proved that every proposition is a set. This holds in general for n-types: any n-type is an $(n+1)$-type. The proof is similar to the case $n = -1$, and we omit it. It can furthermore be shown that n-truncatedness is stable under equivalence (see [11, p. 287]).

Given a type X, there is a canonical way to turn it into an n-type. This type should be thought of as being obtained from X by throwing away all higher paths above dimension n.

Theorem 3.2. *Let X be a type. There exists an n-type $\|X\|_n$, called the **n-truncation** of X, together with a map $|\cdot| \colon X \to \|X\|_n$ such that for any n-type Y and map $f \colon X \to Y$, there exists a unique map $\mathsf{ext}(f) \colon \|X\|_n \to Y$ (up to propositional equality), such that $f = \mathsf{ext}(f) \circ |\cdot|$.*

$$\begin{tikzcd} X \arrow[rr, "f"] \arrow[dr, "|\cdot|"'] & & Y \\ & \|X\|_n \arrow[ur, dashed, "\exists!\,\mathsf{ext}(f)"'] & \end{tikzcd}$$

Proof. The type $\|X\|_n$ can be constructed as a certain higher inductive type which has $|\cdot| \colon X \to \|X\|_n$ as a point constructor. See [11, pp. 294–5] for details. The proof of the universal property can be found on pp. 295–6. □

Given types A, B and a map $f\colon A \to B$, we obtain a map $\|f\|_n : \|A\|_n \to \|B\|_n$ by applying the universal property to the map $|\cdot| \circ f\colon A \to \|B\|_n$. It is easy to check that $\|f \circ g\|_n = \|f\|_n \circ \|g\|_n$ and $\|\mathrm{id}_A\|_n = \mathrm{id}_{\|A\|_n}$. It follows that if $A \simeq B$, then $\|A\|_n \simeq \|B\|_n$ for all n. We also have the following:

Proposition 3.3. *A type A is an n-type if and only if $|\cdot| : A \to \|A\|_n$ is an equivalence.*

Proof. The implication from right to left is immediate. So suppose A is an n-type. The identity map id_A then induces a map $\mathrm{ext}(\mathrm{id}_A)\colon \|A\|_n \to A$. It follows immediately from the universal property of truncation that $\mathrm{ext}(\mathrm{id}_A) \circ |\cdot| = \mathrm{id}_A$. On the other hand, we have

$$\mathrm{id}_{\|A\|_n} \circ |\cdot| = |\cdot| \circ \mathrm{id}_A = |\cdot| \circ \mathrm{ext}(\mathrm{id}_A) \circ |\cdot|.$$

This implies that both $\mathrm{id}_{\|A\|_n}$ and $|\cdot| \circ \mathrm{ext}(\mathrm{id}_A)$ make the diagram

$$\begin{tikzcd} A \arrow[rr, "|\cdot|"] \arrow[dr, "|\cdot|"'] & & \|A\|_n \\ & \|A\|_n \arrow[ur, dashed] & \end{tikzcd}$$

commute. By uniqueness, we must have $|\cdot| \circ \mathrm{ext}(\mathrm{id}_A) = \mathrm{id}_{\|A\|_n}$. □

We write $P : A \to n\text{−}\mathsf{Type}$ if P is a type family and $P(a)$ is an n-type for all $a\colon A$. We call such P a family of n-types.

Proposition 3.4. *Suppose $P : A \to n\text{−}\mathsf{Type}$ is a family of n-types. Then $\prod_{(a:A)} P(a)$ is an n-type.*

Proof. We proof this by induction on n.

If $n = -2$, then $P(a)$ is contractible for all $a : A$. Let $c : \prod_{(a:A)} P(a)$ be the function such that $c(a)$ is the center of contraction of $P(a)$ for all $a : A$. Let $f : \prod_{(a:A)} P(a)$. Then for all $a : A$, we have that $f(a) = c(a)$, since $f(a) : P(a)$ and $P(a)$ is contractible with center of contraction $c(a)$. So by function extensionality, $f = c$, which implies that $\prod_{(a:A)} P(a)$ is contractible with center of contraction c.

Now suppose that the statement holds for all families of n-types and P is a family of $(n+1)$-types. Let $f,\ g : \prod_{(a:A)} P(a)$. By function extensionality, the type $f = g$ is equivalent to $\prod_{(a:A)} f(a) = g(a)$. Since $P(a)$ is an $(n+1)$-type for all $a : A$, the family $a \mapsto f(a) = g(a)$ is a family of n-types. It now follows from the induction hypothesis that $\prod_{(a:A)} f(a) = g(a)$ is an n-type, which proves that $\prod_{(a:A)} P(a)$ is an $(n+1)$-type. □

3.2. Groups

The definition of a group in homotopy type theory contains no surprises.

Definition 3.5. A **group** is a set G together with the following data:

- an element $e\colon G$,

- a map $m \colon G \to G \to G$,
- a map $\iota \colon G \to G$,

satisfying

- for all $g : G$ we have $m(g, e) = g = m(e, g)$,
- for all $g : G$ we have $m(g, \iota(g)) = e = m(\iota(g), g)$,
- for all $g, h, k : G$ we have $m(g, m(h, k)) = m(m(g, h), k)$.

If, moreover

- for all $g, h : G$ we have $m(g, h) = m(h, g)$,

then we call G **abelian**.

Remark. Note that we require a group G to be a set. If we would allow G to be any type, we would end up with the notion of ∞-group, where all the group axioms only need to hold up to homotopy.

Example 3.6. The integers $\mathbb{Z}$ can be constructed as a type with an element $0 : \mathbb{Z}$ and maps $\mathsf{succ} \colon \mathbb{Z} \to \mathbb{Z}$, $\mathsf{pred} \colon \mathbb{Z} \to \mathbb{Z}$, and $\mathsf{min} \colon \mathbb{Z} \to \mathbb{Z}$, where we write the last one as $n \mapsto -n$. These all behave as one would expect. In particular, succ is an equivalence from $\mathbb{Z}$ to itself, since it has pred as a two-sided inverse. It can be proved that $\mathbb{Z}$ is a set. There exists an obvious map $\mathbb{N} \to \mathbb{Z}$, which we use to identify the natural numbers with the non-negative integers.

We can prove things about the integers by means of double induction. More precisely, given $P \colon \mathbb{Z} \to \mathcal{U}$ and

- an element $c_0 : P(0)$,
- a function $c_s : \prod_{(n : \mathbb{N})} P(n) \to P(\mathsf{succ}(n))$,
- a function $c_p : \prod_{(n : \mathbb{N})} P(-n) \to P(-\mathsf{succ}(n))$,

we obtain a function $f : \prod_{(n : \mathbb{Z})} P(n)$ such that $f(0) \equiv c_0$, and for all $n : \mathbb{N}$, $f(\mathsf{succ}(n)) \equiv c_s(n, f(n))$ and $f(-\mathsf{succ}(n)) \equiv c_p(n, f(-n))$.

One easily defines addition $+ \colon \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ using double induction, and this turns $\mathbb{Z}$ into a group.

A homomorphism between groups is a function that commutes with the group operation. An isomorphism of groups is a homomorphism that is also an equivalence.

3.3. Homotopy groups

Algebraic topology is concerned with constructing algebraic invariants of spaces. The simplest such invariant is the fundamental group. Recall that for a topological space X with basepoint $x \in X$, the fundamental group $\pi_1(X, x)$ consists of path-homotopy classes of maps $[0, 1] \to X$ which send 0 and 1 to x. The group operation is given by concatenation of paths. One can think of $\pi_1(X, x)$ as detecting holes in X, since a loop in X that goes around a hole cannot be shrunk continuously to the constant path at x.

It is possible to generalize this construction to higher dimensions. For each $n \geq 1$, we obtain a group $\pi_n(X, x)$, called the **n-th homotopy group** of X, which consists of homotopy classes of maps $[0, 1]^n \to X$ which send the boundary of the n-cube to the base point x. These higher homotopy groups detect higher dimensional holes in the space X. For example, $\mathbb{S}^2$ has trivial fundamental group, but $\pi_2(\mathbb{S}^2)$ is isomorphic to $\mathbb{Z}$, reflecting the intuition that $\mathbb{S}^2$ encloses a 2-dimensional hole.

In light of the interpretation of types as spaces, it is to be expected that we can define homotopy groups in homotopy type theory. We will carry this out in the current section. Because we need to keep track of base points in the construction of the homotopy groups, we start with the following definition.

Definition 3.7. A **pointed type** is a type A together with a specified point $a : A$. Equivalently, it is an element of the type $\sum_{(A:\mathcal{U})} A$.

As we saw, the fundamental group in classical homotopy theory is defined in terms of loops at the base point of a space. Since we have interpreted elements of identity types as paths and loops, it would seem to make sense to define the fundamental group of a pointed type (X, x_0) as the type $x_0 = x_0$. Moreover, if we interpret paths between paths as homotopies or higher-dimensional paths, it seems reasonable to define the second homotopy group to be $\mathsf{refl}_{x_0} = \mathsf{refl}_{x_0}$, and make similar definitions for the higher homotopy groups. However, these types are in general not sets, which is required in order for them to be groups. Therefore, we give them a different name.

Definition 3.8. Let (X, x_0) be a pointed type. The **loop space** of (X, x_0) is the following pointed type

$$\Omega(X, x_0) :\equiv (x_0 = x_0,\ \mathsf{refl}_{x_0}).$$

For $n : \mathbb{N}$, we define the **n-fold iterated loop space** of (X, x_0) recursively as

$$\begin{aligned} \Omega^0(X, x_0) &:\equiv (X, x_0), \\ \Omega^{n+1}(X, x_0) &:\equiv \Omega(\Omega^n(X, x_0)). \end{aligned}$$

Note that for $n \geq 1$, concatenation of paths gives an associative operation with identity element and inverses on $\Omega^n(X, x_0)$. By truncating it to a set, we can turn it into a group.

Definition 3.9. Let (X, x_0) be a pointed type. For $n \geq 1$, the **n-th homotopy group** of (X, x_0) is

$$\pi_n(X, x_0) :\equiv \|\Omega^n(X, x_0)\|_0 .$$

By means of the universal property of truncation, concatenation on $\Omega^n(X, x_0)$ induces an associative operation with unit and inverses on $\pi_n(X, x_0)$, turning this set into a group.

3.4. The universal covering space of $\mathbb{S}^1$

We will now work towards a proof that $\pi_1(\mathbb{S}^1)$ is isomorphic to $\mathbb{Z}$. In fact, we will prove the stronger statement that $\Omega^1(\mathbb{S}^1, \mathsf{base})$ is equivalent to $\mathbb{Z}$, which implies that all higher homotopy groups of $\mathbb{S}^1$ are trivial, since $\mathbb{Z}$ is a set.

Recall that in the classical proof of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$, the key idea is the introduction of the universal covering space of the circle. This is the map

$$\mathbb{R} \to \mathbb{S}^1, \quad t \mapsto (\cos(2\pi t), \sin(2\pi t)),$$

which is often depicted as an infinite helix projecting onto the circle. If we pick $(0, 1)$ as a base point in the circle, the fiber above the base point consists of the integers. Because this map is a covering space, it is possible to uniquely lift paths in the circle to paths in $\mathbb{R}$. Sending a loop in the circle to the end point of its lift in $\mathbb{R}$ provides a bijection between $\pi_1(\mathbb{S}^1)$ and $\mathbb{Z}$, which turns out to be a group isomorphism. (See [7, pp. 29–31] for full details.)

We will try to emulate this proof in homotopy type theory. To do so, we first need to construct the analog of the universal covering space of $\mathbb{S}^1$. As we explained in the first chapter, a fibration of $\mathbb{S}^1$ in homotopy type theory is a map $\mathbb{S}^1 \to \mathcal{U}$. A covering space is a fibration with discrete fiber. Translating to homotopy type theory, all fibers should be sets, and in particular, base should be sent to $\mathbb{Z}$. Because of the path-lifting interpretation of the transport function we gave in the first chapter, transporting along loop should move a point in the fiber up by one level in the spiral, which corresponds to the function $\mathsf{succ} : \mathbb{Z} \to \mathbb{Z}$. These considerations lead us to the following definition.

Definition 3.10 (Universal covering space of $\mathbb{S}^1$)**.** Let $\mathsf{code} : \mathbb{S}^1 \to \mathcal{U}$ be the map defined by circle recursion as

$$\mathsf{code}(\mathsf{base}) :\equiv \mathbb{Z}, \qquad \mathsf{code}(\mathsf{loop}) := \mathsf{ua}(\mathsf{succ}),$$

where $\mathsf{ua}(\mathsf{succ}) : \mathbb{Z} = \mathbb{Z}$ is the path obtained from the equivalence $\mathsf{succ} : \mathbb{Z} \to \mathbb{Z}$ by the univalence axiom.

We now check that code satisfies the desired transport properties.

Lemma 3.11. *Given $n : \mathbb{Z}$, we have that*

(a) $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, n) = \mathsf{succ}(n)$,

(b) $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}, n) = \mathsf{pred}(n)$.

Proof. The two parts are similar, so we only do the first one. By Proposition 2.14, we have that

$$\begin{aligned}\mathsf{transport}^{\mathsf{code}}(\mathsf{loop},\ n) &= \mathsf{transport}^{X\mapsto X}(\mathsf{ap}_{\mathsf{code}}(\mathsf{loop}),\ n)\\ &= \mathsf{transport}^{X\mapsto X}(\mathsf{ua}(\mathsf{succ}),\ n)\\ &= \mathsf{succ}(n),\end{aligned}$$

where the last equality follows from univalence. □

3.5. The proof of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$

The proof now proceeds by showing that for all $x : \mathbb{S}^1$, $\mathsf{code}(x)$ is equivalent to $\mathsf{base} = x$. By instantiating this equivalence at base, we obtain an equivalence $\mathbb{Z} \simeq \Omega(\mathbb{S}^1, \mathsf{base})$, which turns out to take addition to concatenation of paths.

We start by defining the analog of lifting loops from the circle to the covering space.

Definition 3.12. The **encode function** of code is given by

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

The function in the other direction is more complicated to construct. As a preliminary, we define a function $\mathsf{loop}^{-} : \mathbb{Z} \to \Omega(\mathbb{S}^1, \mathsf{base})$ by induction on $\mathbb{Z}$ as

- $\mathsf{loop}^0 :\equiv \mathsf{refl}_{\mathsf{base}}$,
- $\mathsf{loop}^{n+1} :\equiv \mathsf{loop}^n \cdot \mathsf{loop}$ for all $n : \mathbb{N}$,
- $\mathsf{loop}^{-(n+1)} :\equiv \mathsf{loop}^{-n} \cdot \mathsf{loop}^{-1}$ for all $n : \mathbb{N}$.

The map in the other direction will essentially be loop^{-}. To proof this is well defined, we first need the following lemma.

Lemma 3.13. *Given a type X and type families $A, B : X \to \mathcal{U}$, we can form the type family $x \mapsto (A(x) \to B(x))$. Transport in this family is characterized as follows: given a path $p : x =_X y$ and a function $f : A(x) \to B(x)$, we have*

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

Proof. By path induction on p, the left hand side becomes equal to f, while the right hand side reduces to

$$x \mapsto \mathsf{transport}^{B}(\mathsf{refl}_x,\ f(\mathsf{transport}^{A}(\mathsf{refl}_x^{-1},\ x))),$$

which equals $x \mapsto f(x)$. □

Theorem 3.14. *The **decode function** of* code*, defined by circle induction as*

$$\begin{aligned}\mathsf{decode} : \prod_{(x:\mathbb{S}^1)} \mathsf{code}(x) \to (\mathsf{base} = x)\\ \mathsf{decode}(\mathsf{base}) :\equiv \mathsf{loop}^{-}\end{aligned}$$

is well defined.

Proof. To complete the circle induction, we must provide a path

$$\mathsf{transport}^{x \mapsto \mathsf{code}(x) \to (\mathsf{base}=x)}(\mathsf{loop},\ \mathsf{loop}^{-}) = \mathsf{loop}^{-}.$$

By the previous lemma, the left hand side is equal to

$$\mathsf{transport}^{x \mapsto (\mathsf{base}=x)}(\mathsf{loop},\ -) \circ \mathsf{loop}^{-} \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1},\ -)$$

as a function in $\mathbb{Z} \to \mathbb{Z}$. By applying Proposition 2.12, we see this is equal to

$$(- \cdot \mathsf{loop}) \circ (\mathsf{loop}^{-}) \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1},\ -).$$

It follows from Lemma 3.11 that this is just

$$(- \cdot \mathsf{loop}) \circ (\mathsf{loop}^{-}) \circ \mathsf{pred} = (n \mapsto \mathsf{loop}^{n-1} \cdot \mathsf{loop}).$$

We now prove this is equal to loop^{-} by induction on n. For $n \equiv 0$, we have

$$\mathsf{loop}^{0-1} \cdot \mathsf{loop} = \mathsf{loop}^{-1} \cdot \mathsf{loop} = \mathsf{refl}_x = \mathsf{loop}^0.$$

If the statement holds for certain $n : \mathbb{N}$, we have

$$\mathsf{loop}^{(n+1)-1} \cdot \mathsf{loop} = \mathsf{loop}^{n} \cdot \mathsf{loop} = \mathsf{loop}^{n+1},$$

where the last equality holds by definition of loop^{-}. Suppose now the statement holds for some $-n$ where $n : \mathbb{N}$. Then

$$\begin{aligned}\mathsf{loop}^{-(n+1)-1} \cdot \mathsf{loop} &= \mathsf{loop}^{-(n+1)} \cdot \mathsf{loop}^{-1} \cdot \mathsf{loop}\\ &= \mathsf{loop}^{-(n+1)} \cdot \mathsf{refl}_{\mathsf{base}} = \mathsf{loop}^{-(n+1)}.\end{aligned}$$

We conclude that $(n \mapsto \mathsf{loop}^{n-1} \cdot \mathsf{loop}) = \mathsf{loop}^{-}$ by function extensionality. This completes the proof of the theorem. ☐

The next step is to show that for each $x : \mathbb{S}^1$, the functions given by encode and decode are inverse to each other.

Proposition 3.15. *For all* $x : \mathbb{S}^1$ *and* $p : \mathsf{base} = x$*, we have*

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

Proof. By applying based path induction to p, the left hand side becomes

$$\begin{aligned}\mathsf{decode}(\mathsf{base}, \mathsf{encode}(\mathsf{base}, \mathsf{refl}_{\mathsf{base}})) &= \mathsf{decode}(\mathsf{base}, 0)\\ &= \mathsf{loop}^0 = \mathsf{refl}_{\mathsf{base}},\end{aligned}$$

which is equal to p. □

Going the other way is a little more elaborate.

Proposition 3.16. *For all $x : \mathbb{S}^1$ and $c : \mathsf{code}(x)$, we have that*

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

Proof. We proof this by circle induction. This means we first have to prove the equality for $x \equiv \mathsf{base}$. In that case, we have to prove for all $n : \mathbb{Z}$ that

$$\mathsf{encode}(\mathsf{base}, \mathsf{decode}(\mathsf{base}, n)) = n.$$

We proceed by induction on n.

First let $n \equiv 0$. Then $\mathsf{decode}(\mathsf{base}, n) = \mathsf{refl}_{base}$. Since $\mathsf{encode}(\mathsf{base}, \mathsf{refl}_{base}) = 0$, we are done.

Now we suppose the equality holds for some $n : \mathbb{N}$. Since $\mathsf{decode}(\mathsf{base}, n+1) = \mathsf{loop}^{n+1} = \mathsf{loop}^n \cdot \mathsf{loop}$, we have

$$\begin{aligned}\mathsf{encode}(\mathsf{base}, \mathsf{decode}(\mathsf{base}, n+1)) &= \mathsf{encode}(\mathsf{base},\ \mathsf{loop}^n \cdot \mathsf{loop})\\ &= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^n \cdot \mathsf{loop},\ 0)\end{aligned}$$

By Proposition 2.13, this last expression is equal to

$$\begin{aligned}\mathsf{transport}^{\mathsf{code}}(\mathsf{loop},\ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^n,\ 0)) &= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop},\ n)\\ &= n+1,\end{aligned}$$

where the first equality follows from the induction hypothesis, and the second from Lemma 3.11.

For the final case, we assume the equality holds for $-n$ where $n : \mathbb{N}$. Like before, we have $\mathsf{decode}(\mathsf{base}, -(n+1)) = \mathsf{loop}^{-n} \cdot \mathsf{loop}^{-1}$. It follows that

$$\begin{aligned}\mathsf{encode}(\mathsf{base}, \mathsf{decode}(\mathsf{base}, -(n+1))) &= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-n} \cdot \mathsf{loop}^{-1},\ 0)\\ &= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1},\ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-n},\ 0))\\ &= \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1},\ -n)\\ &= -(n+1),\end{aligned}$$

as desired.

To complete the circle induction, we still need to show that the path we have provided in

$$\mathsf{encode}(\mathsf{base}, \mathsf{decode}(\mathsf{base}, n)) =_{\mathbb{Z}} n$$

gets mapped to itself when transported along loop. But since $\mathbb{Z}$ is a set, all its identity types are mere propositions, so this must be the case. □

Corollary 3.17. *For all $x : \mathbb{S}^1$ we have* $(\mathsf{base} = x) \simeq \mathsf{code}(x)$. *In particular,*

$$\begin{array}{rcl} \Omega(\mathbb{S}^1, \mathsf{refl}_{\mathsf{base}}) & \simeq & \mathbb{Z}, \\ p & \mapsto & \mathsf{transport}^{\mathsf{code}}(p,\ 0), \\ \mathsf{loop}^n & \leftarrow\!\shortmid & n. \end{array}$$

This brings us to our goal.

Corollary 3.18. *We have the following isomorphisms of groups:*

$$\pi_n(\mathbb{S}^1) \simeq \begin{cases} \mathbb{Z} & \textit{if } n = 1, \\ 0 & \textit{else.} \end{cases}$$

Proof. By applying $\|-\|_0$ to both sides of the equivalence $\Omega(\mathbb{S}^1) \simeq \mathbb{Z}$, we get $\pi_1(\mathbb{S}^1) \simeq \mathbb{Z}$. Here we use that $\|\mathbb{Z}\|_0 \simeq \mathbb{Z}$, which follows from Proposition 3.3 and the fact that $\mathbb{Z}$ is a set. It can be shown with an easy induction argument that for all $m, n : \mathbb{Z}$, we have

$$\mathsf{loop}^{m+n} = \mathsf{loop}^m \cdot \mathsf{loop}^n.$$

Therefore, the truncation of loop is not only an equivalence, but also a group isomorphism.

Since $\mathbb{Z}$ is a set, all its higher loop spaces are trivial. It then follows from the equivalence $\Omega(\mathbb{S}^1) \simeq \mathbb{Z}$ that $\pi_n(\mathbb{S}^1) \simeq 0$ for $n > 1$. $\square$

4. The Freudenthal Suspension Theorem

4.1. n-connectedness

In the previous chapter we introduced the notion of a type being n-truncated, which means that it has no non-trivial paths above dimension n. The dual notion is n-connectedness. A type is n-connected if it has no non-trivial paths *below* dimension n. It turns out to be useful to define this notion not only for types but also for maps.

Definition 4.1. Let $n \geq -2$. A map $f : A \to B$ is **n-connected** if for all $y : B$ the n-truncated fiber $\left\|\mathsf{fib}_f(y)\right\|_n$ is contractible. A type A is called n-connected if the unique map $A \to \mathbf{1}$ is n-connected.

Similarly, we will call a map from A to B n-truncated if all its fibers are n-truncated.

The most useful property of n-connected maps is stated in the following theorem, which is a kind of induction principle with respect to families of n-types.

Theorem 4.2. *Suppose $f : A \to B$ is n-connected and $P : B \to n{-}\mathsf{Type}$. Then the pullback map*

$$f^* : \left(\prod_{(b:B)} P(b)\right) \to \left(\prod_{(a:A)} P(f(a))\right), \qquad s \mapsto s \circ f$$

is an equivalence.

Proof. See [11, p. 308]. □

We mention the important special case where P is a constant type family. We then have that the pullback

$$f^* : (B \to C) \to (A \to C)$$

is an equivalence if and only if C is an n-type.

Theorem 4.2 is often useful when A is in some sense simpler than B and one wants to prove something for all elements of B. If the conditions of the theorem are met, it is possible to pullback along f, after which one only has to prove the statement for all elements of A. In this way the theorem resembles an induction principle.

This principle sometimes makes it possible to prove a property for all elements of a type by only proving it for a basepoint, as the following proposition shows.

Proposition 4.3. *Let A be a type and $a : \mathbf{1} \to A$ the inclusion of a basepoint. For $n \geq -1$, the type A is n-connected if and only if the map a is $(n-1)$-connected.*

Proof. See [11, p. 310]. $\square$

We will now generalize Theorem 4.2 to a stronger induction principle. To do so, we will need the following characterization of equality in Σ-types, which will also be useful later on.

Proposition 4.4. *Let A be a type and $B : A \to \mathcal{U}$ a family. Let $x, y : \sum_{(a:A)} B(a)$. The following equivalence then holds:*

$$(x = y) \simeq \sum_{(p:\,\mathsf{pr}_1(x) = \mathsf{pr}_1(y))} \mathsf{transport}^B(p,\ \mathsf{pr}_2(x)) = \mathsf{pr}_2(y).$$

Proof. See [11, p. 110]. $\square$

Remark. The function

$$\left(\sum_{(p:\,\mathsf{pr}_1(x) = \mathsf{pr}_1(y))} \mathsf{transport}^B(p,\ \mathsf{pr}_2(x)) = \mathsf{pr}_2(y) \right) \to (x = y)$$

of this equivalence will be denoted by $\mathsf{pair}^=$.

We can now prove the generalized induction principle.

Theorem 4.5. *Suppose we have an n-connected map $f : A \to B$ and a family $P : A \to k{-}\mathsf{Type}$, where $k \geq n$. Then the pullback map*

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

is $(k - n - 2)$-truncated.

Proof. We prove this by induction on $k - n$. If $k = n$, this is just Theorem 4.2. So suppose the statement holds for some $k \geq n$ and that P is a family of $(k+1)$-types. Let $\ell : \prod_{(a:A)} P(f(a))$ and consider

$$\mathsf{fib}_{f^*}(\ell) \equiv \sum_{(k:\,\prod_{(b:B)} P(b))} k \circ f = \ell.$$

We want to show this fiber is $(k - n - 1)$-truncated. So let $(g,\ p)$ and $(h,\ q)$ be elements of this fiber. By the characterization of equality in Σ-types, we have that

$$((g,\ p) = (h,\ q)) \simeq \sum_{(r:\,g=h)} \left(\mathsf{transport}^{k \mapsto k \circ f = \ell}(r,\ p) = q \right). \tag{4.1}$$

We need to prove that this identity type is $(k - n - 2)$-truncated. I claim that for all $r : g = h$, we have

$$(\mathsf{transport}^{k \mapsto k \circ f = \ell}(r,\ p) = q) \simeq (\mathsf{happly}(r) \circ f = \mathsf{happly}(p \cdot q^{-1})).$$

This can be proved by applying path induction on r, which reduces this statement to

$$(p = q) \simeq (\mathsf{happly}(\mathsf{refl}_g) \circ f = \mathsf{happly}(p \cdot q^{-1})).$$

But this equivalence certainly holds, since $\mathsf{happly}(\mathsf{refl}_g) \circ f$ is the function which sends a to $\mathsf{refl}_{g(f(a))}$, and this function is homotopic to $\mathsf{happly}(p \cdot q^{-1})$ if and only if $p = q$.

Combining this with function extensionality, we get that the right hand side of 4.1 is equivalent to $\sum_{(r:g\sim h)} r \circ f = \mathsf{happly}(p \cdot q^{-1})$. But this type is the fiber over $\mathsf{happly}(p \cdot q^{-1})$ of the pullback function

$$f^* : \left(\prod_{(b:B)} Q(b)\right) \to \left(\prod_{(a:A)} Q(f(a))\right),$$

where $Q(b) :\equiv (g(b) = h(b))$. Since P is a family of $(k+1)$-types, $g(b) = h(b)$ is a k-type for all $b : B$. So Q is a family of k-types, and it follows from the induction hypothesis that this fiber over $\mathsf{happly}(p \cdot q^{-1})$, and hence 4.1, is $(k - n - 2)$-truncated.

□

4.2. Suspensions

In this section, we introduce a construction on types that plays an important role in homotopy theory.

Definition 4.6. Given a type A, we construct a type ΣA, called the **suspension** of A, as the higher inductive type with the following constructors:

- two elements $\mathsf{N}, \mathsf{S} : \Sigma A$,
- a function $\mathsf{merid} : A \to \mathsf{N} =_{\Sigma A} \mathsf{S}$.

The induction principle states that given a type family $P : \Sigma A \to \mathcal{U}$ and the following data

- elements $b_n : P(\mathsf{N})$ and $b_s : P(\mathsf{S})$,
- for each $a : A$ a path $m(a) : b_n =^P_{\mathsf{merid}(a)} b_s$,

we get a function $f : \prod_{(x:\Sigma A)} P(x)$ satisfying the obvious computation rules.

The suspension ΣA can be thought of as being obtained by turning the points of A into paths. A basic example of a series of spaces related to each other by suspensions is given by the spheres. The 0-sphere $\mathbb{S}^0$ is just the type **2**. It is possible to prove that $\Sigma\mathbb{S}^0 \simeq \mathbb{S}^1$ (see [11, p. 246]). We have illustrated this in figure 4.1. Suspensions can also be defined in classical homotopy theory, where this relation turns out to hold in general, i.e. for $n \geq 0$, the space $\Sigma\mathbb{S}^n$ is homeomorphic to $\mathbb{S}^{n+1}$. Therefore, we have the formula $\Sigma^n\mathbb{S}^0 = \mathbb{S}^n$, where Σ^n denotes n iterated applications of the suspension. In homotopy type theory, we use this formula to define $\mathbb{S}^n$ for $n \geq 2$.

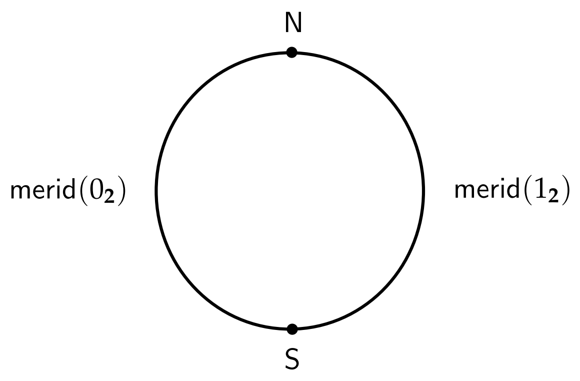


Figure 4.1.: The suspension of $\mathbb{S}^0$ is $\mathbb{S}^1$.

Definition 4.7. For $n \geq 2$, the ***n*-sphere** $\mathbb{S}^n$ is the type $\Sigma^n \mathbb{S}^0$.

Since Σ turns points into paths, and Ω turns paths into points, these operations are in some sense inverse to each other. We can make this more precise. Suppose (X, x_0) is a pointed type. Consider ΣX as a pointed type by taking N as its basepoint. We then obtain the following map

$$\sigma\colon X \to \Omega\Sigma X, \qquad x \mapsto \mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1}.$$

The Freudenthal suspension theorem, which we will prove in this chapter, states that for an n-connected pointed type X, with $n \geq 0$, this map σ is $2n$-connected.This theorem is one of the basic results in homotopy theory, and makes it possible to compute some homotopy groups of spheres. In particular, it allows us to prove that $\pi_n(\mathbb{S}^n) = \mathbb{Z}$ for all $n > 0$. We will come back to this in the final section of this chapter.

4.3. The wedge connectivity lemma

We start out by proving the wedge connectivity lemma, which plays a key role in the proof of the Freudenthal suspension theorem.

Lemma 4.8 (Wedge connectivity lemma)**.** *Let $(A,\ a_0)$, $(B,\ b_0)$ be pointed types, such that A is an $m-\mathsf{Type}$ and B is an $n-\mathsf{Type}$, where $m, n \geq 0$. Suppose we have a family*

$$P : A \to B \to (m+n)-\mathsf{Type},$$

functions $f : \prod_{(a:A)} P(a,\ b_0)$ and $g : \prod_{(b:B)} P(a_0,\ b)$, and a path $p : f(a_0) = g(b_0)$. Then there exists a function $h : \prod_{(a:A)} \prod_{(b:B)} P(a,\ b)$ such that

$$h(-,\ b_0) = f \qquad \text{and} \qquad h(a_0,\ -) = g.$$

Proof. Define $Q : A \to \mathcal{U}$ by setting

$$Q(a) :\equiv \sum_{(k:\, \prod_{(b:B)} P(a,\, b))} f(a) = k(b_0).$$

We first show Q is a family of $(m-1)$-types. Fix $a : A$. Since $b_0 : \mathbf{1} \to B$ is $(n-1)$-connected by Proposition 4.3, and $P(a, b)$ is $(n+m)$-connected for all $b : B$, it follows from Theorem 4.5 that the pullback map

$$\prod_{(b:B)} P(a,\, b) \to P(a,\, b_0)$$

is $(m-1)$-truncated. But $Q(a)$ is the fiber over $f(a)$ of this map. Hence, Q is a family of $(m-1)$-types.

We now prove the theorem. Note that $(g,\, p) : Q(a_0)$. We have that $a_0 : \mathbf{1} \to A$ is $(m-1)$-connected, so since Q is a family of $(m-1)$-types, we can apply Theorem 4.2 to obtain a section $s : \prod_{(a:A)} Q(a)$ satisfying $s(a_0) = (g,\, p)$. Now define h by $h(a,\, b) :\equiv \mathsf{pr}_1(s(a))(b)$. By construction, we have that $h(a_0,\, b) = g(b)$ for $b : B$. When b is b_0, it follows from the definition of Q that $\mathsf{pr}_2(s(a)) : h(a,\, b_0) = f(a)$.

□

4.4. Proof of the suspension theorem

We now proceed to prove the Freudenthal suspension theorem. Throughout this section, we fix $n \geq 0$ and an n-connected and pointed type $(X,\, x_0)$.

We need to prove that the map $\sigma : X \to \Omega\Sigma X$ is $2n$-connected, which means that the $2n$-truncations of all fibers of this map are contractible. To do so, we again introduce a code-function.

Theorem 4.9. *There exists a function* code : $\prod_{(y:\Sigma X)} (\mathsf{N} = y) \to \mathcal{U}$ *satisfying*

$$\mathsf{code}(\mathsf{N},\, p) :\equiv \|\mathsf{fib}_\sigma(p)\|_{2n} \equiv \left\| \sum_{(x:X)} \mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1} = p \right\|_{2n}$$

and

$$\mathsf{code}(\mathsf{S},\, q) :\equiv \|\mathsf{fib}_{\mathsf{merid}}(q)\|_{2n} \equiv \left\| \sum_{(x:X)} \mathsf{merid}(x) = q \right\|_{2n}$$

Our strategy will be to prove that $\mathsf{code}(y, p)$ is contractible for all $y : \Sigma X$ and $p : \mathsf{N} = y$. The theorem then follows by setting $y \equiv \mathsf{N}$. The advantage of proving this more general statement is that we can make use of path induction.

To prove Theorem 4.9, we need the following lemma.

Lemma 4.10. *Given $y : \Sigma X$, a path $m : \mathsf{N} = y$ and function families $C_{\mathsf{N}} : (\mathsf{N} = \mathsf{N}) \to \mathcal{U}$ and $C_y : (\mathsf{N} = y) \to \mathcal{U}$, we obtain an equivalence*

$$\Phi_m : \left(C_{\mathsf{N}} =_m^{\lambda x.(\mathsf{N}=x)\to\mathcal{U}} C_y\right) \simeq \left(\prod_{(q:\mathsf{N}=y)} C_{\mathsf{N}}(q \cdot m^{-1}) \simeq C_y(q)\right),$$

such that $\Phi_{\mathsf{refl}_{\mathsf{N}}} \equiv \mathrm{id}_{C_{\mathsf{N}}}$.

Proof. This follows immediately by path induction on m and function extensionality. □

Proof of Theorem 4.9. We define code by using the induction principle for suspensions. We have already specified $\mathsf{code}(\mathsf{N})$ and $\mathsf{code}(\mathsf{S})$, so it remains to supply for all $x_1 : X$ a path $\mathsf{code}(\mathsf{N}) =_{\mathsf{merid}(x_1)}^{\lambda x.(\mathsf{N}=x)\to\mathcal{U}} \mathsf{code}(\mathsf{S})$. By applying Lemma 4.10 with $y \equiv \mathsf{S}$, $m \equiv \mathsf{merid}(x_1)$, $C_{\mathsf{N}} \equiv \mathsf{code}(\mathsf{N})$, and $C_y \equiv \mathsf{code}(\mathsf{S})$, we find that it is equivalent to prove

$$\prod_{(q:\mathsf{N}=\mathsf{S})} \mathsf{code}(\mathsf{N},\ q \cdot \mathsf{merid}(x_1)^{-1}) \simeq \mathsf{code}(\mathsf{S},\ q).$$

Let $q : \mathsf{N} = \mathsf{S}$. We will construct a map

$$\mathsf{code}(\mathsf{N},\ q \cdot \mathsf{merid}(x_1)^{-1}) \to \mathsf{code}(\mathsf{S},\ q) \tag{4.2}$$

and show it is an equivalence. Since $\mathsf{code}(\mathsf{S},\ q)$ is a $2n$-type and $\mathsf{code}(\mathsf{N},\ q \cdot \mathsf{merid}(x_1)^{-1})$ is the $2n$-truncation of a type, it suffices by the universal property of truncations and the induction principle of Σ-types to provide for each $x_2 : X$ a map

$$(\mathsf{merid}(x_2) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_1)^{-1}) \to \mathsf{code}(\mathsf{S},\ q)$$

Since $\mathsf{code}(\mathsf{S}, q)$ is a $2n$-type, it follows from Proposition 3.4 that for all x_1 and x_2, this function type is a $2n$-type as well. Since X is n-connected, we can apply the wedge connectivity lemma, so that we only have to specify the map in the cases that $x_1 \equiv x_0$ and $x_2 \equiv x_0$ and check that they agree when both x_1 and x_2 are equal to x_0.

Suppose first that $x_1 \equiv x_0$. Then we send $r : \mathsf{merid}(x_2) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_0)^{-1}$ to $|(x_2, r')|_{2n}$, where $r' : \mathsf{merid}(x_2) = q$ is the path obtained from r by cancelling $\mathsf{merid}(x_0)^{-1}$ on both sides. Now suppose $x_2 \equiv x_0$. Then we send $s : \mathsf{merid}(x_0) \cdot \mathsf{merid}(x_0)^{-1} = q \cdot \mathsf{merid}(x_1)^{-1}$ to $|(x_1, s')|_{2n}$, where $s' : \mathsf{merid}(x_1) = q$ is obtained from s by cancelling $\mathsf{merid}(x_0)$ with its inverse and taking $\mathsf{merid}(x_1)$ to the other side. When both x_1 and x_2 are x_0, it can be proved by a simple path induction that the paths r' and s' are the same.

We have now constructed the function 4.2. It remains to show that this is an equivalence. Being an equivalence is a mere proposition. Since X is n-connected for some $n \geq 0$, it follows from Proposition 4.3 that $x_0 : \mathbf{1} \to X$ is at least (-1)-connected. By Theorem 4.2, we can pull back along this inclusion, and it suffices to prove the map 4.2 is an equivalence when x_1 is x_0. But in this case, the function is just the truncated version of cancelling $\mathsf{merid}(x_0)^{-1}$ on the right, which is certainly an equivalence.

□

We are now in a position to prove the suspension theorem.

Theorem 4.11 (Freudenthal suspension theorem)**.** *The map $\sigma : X \to \Omega\Sigma X$ is $2n$-connected.*

Proof. Like we said before, we will prove this by showing that $\mathsf{code}(y, q)$ is contractible for all $y : \Sigma X$ and $q : \mathsf{N} = y$. We start by giving for each y and q a center of contraction $c(y,\ q) : \mathsf{code}(y,\ q)$. We first set $c(\mathsf{N},\ \mathsf{refl_N}) = |(x_0, \mathsf{rinv}_{\mathsf{merid}(x_0)})|_{2n}$, where $\mathsf{rinv}_{\mathsf{merid}(x_0)}$ is the is the standard path in $\mathsf{merid}(x_0) \cdot \mathsf{merid}(x_0)^{-1} = \mathsf{refl_N}$.

Write $\widehat{\mathsf{code}} : \sum_{(y:\Sigma X)}(\mathsf{N} = y) \to \mathcal{U}$ for the uncurried version of code. We define $c(y,\ q)$ as

$$c(y,\ q) :\equiv \mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(q,\ \mathsf{tid}_q),\ c(\mathsf{N},\ \mathsf{refl_N})),$$

where tid_q is the path in $\mathsf{transport}^{\lambda z.(\mathsf{N}=z)}(q,\ \mathsf{refl_N}) = q$ which can be produced by path induction on q.

We must now prove that for all $y : \Sigma X$, $q : \mathsf{N} = y$ and $d : \mathsf{code}(y,\ q)$, we have that $d = c(y,\ q)$. By path induction on q, it suffices to prove it when $y \equiv \mathsf{N}$ and $q \equiv \mathsf{refl_N}$. It turns out to be useful later on to prove it for $y \equiv \mathsf{N}$ and arbitrary $q : \mathsf{N} = \mathsf{N}$. So let $d : \mathsf{code}(\mathsf{N},\ q)$. Since $\mathsf{code}(\mathsf{N},\ q)$ is a $2n$-type, the type $d = c(\mathsf{N},\ q)$ is a $(2n-1)$-type, which is *a forteriori* a $2n$-type. So by the universal property of truncations, it suffices to prove the statement when $d = |(x_1,\ r)|_{2n}$, where $x_1 : X$ and $r : \mathsf{merid}(x_1) \cdot \mathsf{merid}(x_0)^{-1} = q$. We can now apply path induction on r, which means that r becomes $\mathsf{refl}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}}$ and the statement we have to prove shrinks to

$$|(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}})|_{2n} = c(\mathsf{N},\ \mathsf{merid}(x_1) \cdot \mathsf{merid}(x_0)^{-1}).$$

The right hand side of this equation is equal to

$$\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_1) \cdot \mathsf{merid}(x_0)^{-1},\ \mathsf{tid}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}}),\ |(x_0, \mathsf{rinv}_{\mathsf{merid}(x_0)})|_{2n}). \tag{4.3}$$

To compute the result of this application, we first note that we can factor $\mathsf{pair}^{=}(\mathsf{merid}(x_1) \cdot \mathsf{merid}(x_0)^{-1},\ \mathsf{tid}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}})$ as

$$\mathsf{pair}^{=}(\mathsf{merid}(x_1),\ \mathsf{tid}_{\mathsf{merid}(x_1)}) \cdot \mathsf{pair}^{=}(\mathsf{merid}(x_0), \mathsf{t})^{-1},$$

where $\mathsf{t} : \mathsf{transport}^{\lambda z.(\mathsf{N}=z)}(\mathsf{merid}(x_0),\ \mathsf{merid}(x_1) \cdot \mathsf{merid}(x_0)^{-1}) = \mathsf{merid}(x_1)$ follows from Proposition 2.12. This factorization can be proved to hold by considering it for general $y : \Sigma X$ and $p : \mathsf{N} = y$ instead of $\mathsf{merid}(x_1)$ and then applying path induction to p, which results in both factors shrinking to the trivial path. By combining Lemma 2.13 with this factorization, we find that 4.3 is equal to

$$\begin{aligned}\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_0),\ \mathsf{t})^{-1},\ \mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_1),\ \mathsf{tid}_{\mathsf{merid}(x_1)}),\\ |(x_0, \mathsf{rinv}_{\mathsf{merid}(x_0)})|_{2n})).\end{aligned} \tag{4.4}$$

To compute this, we first construct for arbitrary $y : \Sigma X$ and $m : \mathsf{N} = y$ a map $\Psi_m : \prod_{(q:\mathsf{N}=y)} \mathsf{code}(\mathsf{N},\ q \cdot m^{-1}) \to \mathsf{code}(y,\ q)$, by setting $\Psi_m(q) \equiv \Phi_m(\mathsf{apd_{code}}(m))(q)$, where Φ_m is the function from Lemma 4.10, instantiated with $C_{\mathsf{N}} \equiv \mathsf{code}(\mathsf{N})$ and $C_y \equiv \mathsf{code}(y)$. Note that we tacitly identify the equivalence $\Phi_m(\mathsf{apd}_{\mathsf{code}}(m))(q)$ with its underlying function.

Now, as an intermediary step, we will show that

$$\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_1),\ \mathsf{tid}_{\mathsf{merid}(x_1)}),\ |(x_0, \mathsf{rinv}_{\mathsf{merid}(x_0)})|_{2n}) = |(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)})|_{2n}. \tag{4.5}$$

Consider, for arbitrary $y : \Sigma X$ and $m : \mathsf{N} = y$, the function $f_m : \mathsf{code}(\mathsf{N},\ \mathsf{refl}_{\mathsf{N}}) \to \mathsf{code}(\mathsf{N},\ m \cdot m^{-1})$, which is the $2n$-truncation of the function which sends $(x,\ r)$ to $(x,\ r')$, where $r' : \mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1} = m \cdot m^{-1}$ is the obvious path obtained from $r : \mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1} = \mathsf{refl}_{\mathsf{N}}$. It follows immediately from path induction on m that

$$\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(m,\ \mathsf{tid}_m)) = \Psi_m(m) \circ f_m.$$

We now have that 4.5 is equal to

$$\Phi_{\mathsf{merid}(x_1)}(\mathsf{apd}_{\mathsf{code}}(\mathsf{merid}(x_1))(\mathsf{merid}(x_1)) \circ f_{\mathsf{merid}(x_1)}(|(x_0,\ \mathsf{rinv}_{\mathsf{merid}(x_0)})|_{2n}).$$

By the computation rule for the induction principle of suspensions, we know that $\mathsf{apd}_{\mathsf{code}}(\mathsf{merid}(x_1)) = \Phi^{-1}_{\mathsf{merid}(x_1)}(g)$, where $g : \prod_{(q:\,\mathsf{N}=\mathsf{S})} \mathsf{code}(\mathsf{N},\ q \cdot \mathsf{merid}(x_1)^{-1}) \simeq \mathsf{code}(\mathsf{S},\ q)$ is the function we constructed in the definition of code using the wedge connectivity lemma. So $\Phi_{\mathsf{merid}(x_1)}(\mathsf{apd}_{\mathsf{code}}(\mathsf{merid}(x_1))(\mathsf{merid}(x_1)) = g(\mathsf{merid}(x_1))$. Note that we are in the case of the wedge connectivity lemma where x_2 is equal to x_0, so that we can explicitly compute that

$$g(\mathsf{merid}(x_1)) \circ f_{\mathsf{merid}(x_1)}(|(x_0,\ \mathsf{rinv}_{\mathsf{merid}(x_0)})|_{2n}) = |(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)})|_{2n}.$$

We have now established that 4.4 is equal to

$$\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_0),\ \mathsf{t})^{-1},\ |(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)})|_{2n}).$$

Since $\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_0),\ \mathsf{t})^{-1}) = \mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_0),\ \mathsf{t}))^{-1}$, it is enough to show that

$$\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_0),\ \mathsf{t}),\ |(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}})|_{2n}) = |(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)})|_{2n}.$$

This computation is similar to the previous one. Again, it is possible to prove that

$$\mathsf{transport}^{\widehat{\mathsf{code}}}(\mathsf{pair}^{=}(\mathsf{merid}(x_0),\ \mathsf{t})) = \Psi_{\mathsf{merid}(x_0)}(\mathsf{merid}(x_1)),$$

which is equal to $\Phi_{\mathsf{merid}(x_0)}(\mathsf{apd}_{\mathsf{code}}(\mathsf{merid}(x_0))(\mathsf{merid}(x_1))$. By expanding $\mathsf{apd}_{\mathsf{code}}(\mathsf{merid}(x_0))$, and cancelling $\Phi_{\mathsf{merid}(x_0)}$ with its inverse, we end up with $h(\mathsf{merid}(x_1))$, where $h : \prod_{(q:\,\mathsf{N}=\mathsf{S})} \mathsf{code}(\mathsf{N},\ q \cdot \mathsf{merid}(x_0)^{-1}) \simeq \mathsf{code}(\mathsf{S},\ q)$ is the function constructed in the definition of code. We are now in the case of the wedge connectivity lemma where x_1 is equal to x_0, and it follows easily that

$$h(\mathsf{merid}(x_1))(|(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)\cdot\mathsf{merid}(x_0)^{-1}})|_{2n}) = |(x_1,\ \mathsf{refl}_{\mathsf{merid}(x_1)})|_{2n}.$$

This completes the proof. □

4.5. Stability of homotopy groups of spheres

We will now mention some applications of the Freudenthal suspension theorem to the computation of homotopy groups of spheres.

It is possible to prove that if $f : X \to Y$ is n-connected, the truncation $\|f\|_n : \|X\|_n \to \|Y\|_n$ is an equivalence (see [11, pp. 311–312]). It follows from the from the Freudenthal suspension theorem that if X is an n-type for $n \geq 0$, we have an equivalence $\|X\|_{2n} \simeq \|\Omega\Sigma X\|_{2n}$. Using this fact, we can prove the following theorem.

Theorem 4.12. *Suppose $n \geq 0$ and $k \leq 2n - 2$. Then we have that*

$$\pi_k(\mathbb{S}^n) = \pi_{k+1}(\mathbb{S}^{n+1}).$$

Proof sketch. It can be shown that $\mathbb{S}^n$ is $(n-1)$-connected (see [11, p. 351]). Therefore, it follows from the Freudenthal suspension theorem that $\|\Omega\Sigma\mathbb{S}^n\|_{2n-2} = \|\mathbb{S}^n\|_{2n-2}$. Truncating down to $k \leq 2n-2$, we get that

$$\|\Omega\Sigma\mathbb{S}^n\|_k = \|\mathbb{S}^n\|_k. \tag{4.6}$$

Now $\pi_{k+1}(\mathbb{S}^{n+1})$ is equal to $\left\|\Omega^{k+1}\Sigma\mathbb{S}^n\right\|_0$, where we have used that $\mathbb{S}^{n+1} \equiv \Sigma\mathbb{S}^n$. It can be shown that

$$\left\|\Omega^{k+1}\Sigma\mathbb{S}^n\right\|_0 = \Omega^k(\|\Omega\Sigma\mathbb{S}^n\|_k).$$

But this is equal to $\Omega^k(\|\mathbb{S}^n\|_k)$ by 4.6. Repeating the previous step in the opposite direction, we get that this is equal to $\left\|\Omega^k\mathbb{S}^n\right\|_0$, which is just $\pi_k(\mathbb{S}^n)$. □

Corollary 4.13 (Stability of homotopy groups of spheres)**.** *Let $k \geq 0$. For $n \geq k+2$, the groups $\pi_{n+k}(\mathbb{S}^n)$ are all the same.*

The group $\pi_{n+k}(\mathbb{S}^n)$ for $n \gg 0$ is called the ***k*-th stable homotopy group** of the spheres. Computing these groups is an important problem in homotopy theory. See [7, pp. 384-388] for an introduction to this topic.

Finally, we consider the case $k = 0$ from the previous corollary.

Corollary 4.14. *For $n \geq 1$, we have that*

$$\pi_n(\mathbb{S}^n) = \mathbb{Z}.$$

Proof. For $n = 1$, this is just Theorem 3.18. It can also be proved that $\pi_2(\mathbb{S}^2) = \mathbb{Z}$ (see [11, p. 358]). By applying 4.13 with $k = 0$, the result follows. □

5. Homotopy Type Theory in Agda

Agda is a programming language which can be used as a proof assistant for homotopy type theory. In this chapter we give a quick introduction to Agda, explaining all that is necessary for understanding formalizations of proofs from homotopy type theory. We will also discuss the HoTT-Agda library, which contains many basic results from homotopy type theory, as well as various useful idioms. In the next two chapters we will present Agda formalizations of the theorems from synthetic homotopy theory that were discussed in the previous chapters.

The HoTT-Agda library does not seem to work with the latest version of Agda. All Agda code considered in this thesis was tested with Agda version 2.5.4.2.

5.1. Basic syntax of Agda

Looking back at chapter 2, we see that mathematics in homotopy type theory consists of two main components. On the one hand, we have inductive definitions of types. On the other hand, there are constructions of functions, which, by the Curry-Howard correspondence, play the role of proofs. In this section, we will explain how these notions are translated to Agda. Higher inductive types, which are not natively supported in Agda, will be treated in section 5.4.

To define a type, one has to specify the name of the type and give a list of constructors. For example, the definition of the type of natural numbers is as follows.

```
data ℕ : Type where
  0 : ℕ
  S : ℕ → ℕ
```

As in informal type theory, the colon is used to declare the type of an element. The type universe $\mathcal{U}$ is called `Type` in Agda.

It is also possible to define types as *records*, which provide an easy way to construct types whose elements consist of elements from various other types. The Cartesian product is an example of a type that can be defined as a record:

```
record Pair (X Y : Type) : Type where
  constructor pair
  field
    first : X
    second : Y
```

This definition should be read in the following way. Given types `X` and `Y`, we get a type `Pair X Y`. Each element `a` of this type has two components, such that `first a` is an element of `X` and `second a` is an element of `Y`. Given elements `x:X` and `y:Y`, the constructor allows us to construct the element `pair x y` with components `x` and `y`.

A function can be defined by giving its name, domain and codomain and specifying the output for a given input. The following function from the natural numbers to itself is an example of such a definition.

```
cst : ℕ → ℕ
cst n = 0
```

Of course, one of the main characteristics of type theory is that functions are generally constructed inductively. In Agda, we can give inductive definitions of functions by pattern matching. This means that Agda can deduce an induction principle from the constructors of a type, and allows us to define a function by specifying the values it should take on the constructors. For example, the function which doubles a natural number can be defined as

```
double : ℕ → ℕ
double 0 = 0
double S n = S S (double n)
```

Functions of multiple arguments can be handled by means of currying. For example, addition of natural numbers is defined as

```
add : ℕ → ℕ → ℕ
add n 0 = n
add n (S m) = S (add n m)
```

Dependent functions are defined similarly. A type family `P` over a type `X` is just a function of signature

```
P :  X → Type
```

We can define a dependent function `f` into this type family by writing

```
f : (x : X) → P x
f x = e
```

where `e` can depend on `x` and should be of type `P x`. One can make use of pattern matching when constructing dependent functions. Multivariate dependent functions are defined by currying.

A function can also have implicit arguments. These are indicated by curly brackets. Implicit arguments do not have to be supplied when calling a function, since they can be inferred from the context by Agda. For example, when defining a function that has a type `A` as well as a type family `P : A → Type` as arguments, we can given `A` as an implicit argument, since it can be deduced from `P`. The function will then have the following signature.

```
f : {A : Type} (P : A → Type) → C
```

We also mention the possibility of defining functions by means of λ-abstraction. For example, the function which adds 2 to a natural number is defined with λ-abstraction as follows

```
addtwo : ℕ → ℕ
addtwo = λ n → S S n
```

Sometimes when constructing a function, it is useful to define auxiliary functions. These can be introduced in a where-clause. This is illustrated by the following example:

```
f : ℕ → ℕ → ℕ
f n m = add n (times m n) where
  times : ℕ → ℕ → ℕ
  times k 0 = 0
  times k (S r) = add (times k r) k
```

Functions defined in a where-clause are only visible to the function the clause belongs to.

When a number of different functions share arguments, it is possible to specify these arguments only once by organizing the functions in a module. For example, by writing

```
module _ (a : A) (b : B) (c : C) where
```

we create a module with hypotheses `a`, `b`, and `c`. All functions defined in this module will then be able to make use of these arguments. When such a function is called from outside of the module, these arguments need to be supplied as the first three arguments of the function.

5.2. Idioms for encoding homotopy type theory

Universe levels. As was stated in a footnote in section 2.2, instead of having just one type universe, we should have a sequence of nested universes (so-called universes à la Russell) to avoid certain inconsistencies. This is the way type universes are handled in Agda. There is a nested, ℕ-indexed sequence of type universes. If `i` is an index, we write `Type i` for the `i`-th universe. If `A : Type i` and `B : A → Type j`, then the function type `(a : A) → B a` is an element of `Type (max i j)`. There is a useful shorthand for specifying universe level parameters for a function. For instance, we could write

```
f : ∀ {i j} {A : Type i} (P : A → Type j) → C
```

to define a function which has universe levels `i` and `j` as implicit arguments. One can safely ignore all universe levels when reading a piece of Agda code. Only when writing the code, one has to be careful to supply the right universe levels in the right places.

Identity types. The identity types, which were introduced in section 2.5, play a central role in homotopy type theory. They are defined in Agda as

```
data _==_ {i} {A : Type i} (a : A) : A → Type i where
  idp : a == a
```

The underscores in _==_ tell Agda this is an infix operator with two arguments. Since = is a keyword reserved by Agda for function definitions, we use == for propositional equality. The identity type is a dependent type, since we must supply two elements from a type A to obtain it. Note that if A has universe level i, then a == b also has universe level i for any a b : A. From the single constructor idp, Agda infers by pattern matching the principles of path induction and based path induction as defined in section 2.5.

Axiom K. The standard pattern matching rules of Agda are too strong for homotopy type theory. Most problematically, one can use these rules to prove axiom K, which states that for every pair of paths p q : a == b, we have p == q. It is possible to restrict the pattern matching rules and make it impossible to prove axiom K by enabling the option without-K. To do so, each Agda file has to be headed by the line

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

Judgmental equality. Unlike in informal type theory, there is no special symbol for judgmental equality in Agda. Agda simply does not distinguish between elements that are judgmentally equal. For instance, if double : ℕ → ℕ is the function defined above which doubles a natural number, then Agda regards the terms double (S 0) and S S 0 as the same.

5.3. The HoTT-Agda library

The HoTT-Agda library [3] was written by Guillaume Brunerie, Favonia, Evan Cavallo, and many others for formalizing homotopy type theory in Agda. It contains formalizations of all fundamental results from homotopy type theory, including everything that we covered in chapter 2, as well as definitions of n-types, truncations, connectedness and much more. Furthermore, a large section of the library is devoted to proofs of theorems from synthetic homotopy theory (including the proof that $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ and the proof of the Freudenthal suspension theorem).

The formalizations that will be presented in the next two chapters rely heavily on the HoTT-Agda library. In this section, we will discuss some of the important idioms from the library. Throughout, we will refer to the relevant files.

Transport and dependent path types. In the library, the transport function is defined slightly differently from how we defined it in section 2.6, but it behaves essentially in the same way. However, there is a special data type for dependent path types. Recall that if we have a type A, a family $B : A \to \mathcal{U}$, elements $x, y : A$, a path $q : x = y$, and elements $u : B(x)$, $v : B(y)$, we can form the dependent path type $x =^B_q y$, which is nothing else than the type $\mathsf{transport}^B(q,\ u) = v$. In file lib/Base.agda, the dependent path type is defined with path induction as

```
PathOver : ∀ {i j} {A : Type i} (B : A → Type j)
  {x y : A} (p : x == y) (u : B x) (v : B y) → Type j
PathOver B idp u v = (u == v)
```

The library also contains the following syntax for dependent path types, which is less confusing to work with.

```
syntax PathOver B p u v =
  u == v [ B ↓ p ]
```

The file `lib/PathOver.agda` contains functions for translating back and forth between this definition of dependent path types to the one using transport:

```
from-transp : (B : A → Type j) {a a' : A} (p : a == a')
  {u : B a} {v : B a'}
  → (transport B p u == v)
  → (u == v [ B ↓ p ])
from-transp B idp idp = idp

to-transp : {B : A → Type j} {a a' : A} {p : a == a'}
  {u : B a} {v : B a'}
  → (u == v [ B ↓ p ])
  → (transport B p u == v)
to-transp {p = idp} idp = idp
```

In the library, the definition `PathOver` is used almost exclusively. For example, the definition of `apd`, which we defined in Proposition 2.11, is given in the following way:

```
apd : ∀ {i j} {A : Type i} {B : A → Type j} (f : (a : A) → B a) {x y : A}
  → (p : x == y) → f x == f y [ B ↓ p ]
apd f idp = idp
```

This definition is contained in the file `lib/Base.agda`.

Path concatenation. In the library, there are two definitions of path concatenation:

```
_·_ : ∀ {i} {A : Type i} {x y z : A}
  → (x == y → y == z → x == z)
idp · q = q

_·'_ : {x y z : A}
  → (x == y → y == z → x == z)
q ·' idp = q
```

The first one is given in `lib/Base.agda`, the second one in `lib/PathGroupoid.agda`. In both definitions, there is a certain asymmetry in which equalities are judgmental, and which are propositional. For example, it follows from the definition of `_·_` that `idp · q` is judgmentally equal to `q`. However, `q · idp` is only propositionally equal to `q`. For `_·'_` it is the other way around. It is rather surprising that making use of this seemingly innocuous difference can sometimes drastically simplify proofs. A case in point is the formalization of the Freudenthal suspension theorem, which will be discussed in chapter 7. By replacing all occurrences of `_·_` by `_·'_`, we were able to reduce the length of the code by about 100 lines.

Chains of equalities and equivalences. Many propositional equalities are proved by stringing together multiple paths using path concatenation. This quickly leads to illegible code. The library provides a way to write down such a sequence of concatenations in a more comprehensible way. Given a path `p : x == y`, the expression

```
x =⟨ p ⟩
y =■
```

is defined to be a proof of `x == y`. If `q : y == z` is a path as well, and we want to prove `x == z`, we can do so by writing

```
x =⟨ p ⟩
y =⟨ q ⟩
z =■
```

This can be done with arbitrarily many paths, and gives a way of writing down chains of concatenations which is easy to read and adjust.

Chains of equivalences can be written down in a similar manner. The definition of equivalence given in the library is the same as the one we gave in chapter 2. If `A` and `B` are types, the type of equivalences from `A` to `B` is written as `A ≃ B`. If `e : A ≃ B` and `f : B ≃ C`, we can write

```
A ≃⟨ e ⟩
B ≃⟨ f ⟩
C ≃■
```

to prove `A ≃ C`. Just like in the case of equalities, we can do this with arbitrarily many equivalences. Chains of equalities and equivalences are defined in the files `lib/Base.agda` and `lib/Equivalence.agda` respectively.

Π- and Σ-types. We already wrote that given a type `A` and type family `P : A → Type`, the type of dependent functions into this type family is denoted by `(x : A) → P a`. The library offers syntax which is more in line with what we have used in previous chapters:

```
∏ : ∀ {i j} (A : Type i) (P : A → Type j) → Type (lmax i j)
∏ A P = (x : A) → P x
```

The library also contains a definition of Σ-types, which we will often use:

```
record Σ {i j} (A : Type i) (B : A → Type j) : Type (lmax i j) where
  constructor _,_
  field
    fst : A
    snd : B fst
```

Pointed types. In the library, pointed types are defined as a record type:

```
record Ptd (i : ULevel) : Type (lsucc i) where
  constructor ⊙[_,_]
  field
    de⊙ : Type i
    pt : de⊙
open Ptd public
```

So if `X` is a pointed type, then `de⊙ X` is the underlying type and `pt X` is the base point. This definition is given in the file `lib/Base.agda`.

Truncated and connected types. We defined truncated and connected types in sections 3.1 and 4.1. Since they feature prominently in the proof of the Freudenthal suspension theorem, we quickly explain how they are dealt with in the library.

Since connected and truncated types are indexed by integers greater than or equal to -2, a special type is defined for bookkeeping purposes:

```
data TLevel : Type₀ where
  ⟨-2⟩ : TLevel
  S : (n : TLevel) → TLevel

ℕ₋₂ = TLevel
```

The following map is often used. It embeds the natural numbers in this type, sending 0 to -2.

```
⟨_⟩₋₂ : ℕ → ℕ₋₂
⟨ O ⟩₋₂ = ⟨-2⟩
⟨ S n ⟩₋₂ = S ⟨ n ⟩₋₂
```

There are also variants `⟨_⟩₋₁` and `⟨_⟩` which send 0 to -1 and 0 respectively.

Now that we have introduced this indexing type, we can explain the way truncated and connected types are handled in the library. We start with truncated types. Given `n : ℕ₋₂` and a type `A`, the proposition that `A` is an n-type is

```
has-level n A
```

To prove `A` is an n-type, we can use the constructor `has-level-in`. We then have to supply a proof of the following proposition:

```
has-level-aux : ℕ₋₂ → (Type i → Type i)
has-level-aux ⟨-2⟩ A = Σ A (λ x → ((y : A) → x == y))
has-level-aux (S n) A = (x y : A) → has-level n (x == y)
```

These definitions are all contained in the file `lib/NType.agda`.

Truncation of types is defined as a higher inductive type. The most basic functions necessary to work with truncations are

```
Trunc : (n : ℕ₋₂) (A : Type i) → Type i

[_] : {n : ℕ₋₂} {A : Type i} → A → Trunc n A

Trunc-level : {n : ℕ₋₂} {A : Type i} → has-level n (Trunc n A)
```

The first sends a type to its truncation. The second is the introduction rule for truncations. The final function asserts that the n-truncation of a type is n-truncated. We will also need the universal property of truncations:

```
Trunc-elim : {n : ℕ₋₂} {A : Type i} {j} {P : Trunc n A → Type j}
  {{p : (x : Trunc n A) → has-level n (P x)}} (d : (a : A) → P [ a ])
  → ∏ (Trunc n A) P
```

These definitions are contained in the file `lib/types/Truncation.agda`.

Next come connected types. The definition of n-connectedness is as follows:

```
is-connected : ∀ {i} → ℕ₋₂ → Type i → Type i
is-connected n A = is-contr (Trunc n A)
```

The function `is-contr` is nothing else than `has-level -2`. Recall that a map is n-connected if all its fibers have this property. The definition of a fiber in the library is

```
module _ {i j} {A : Type i} {B : Type j} (f : A → B) where
  hfiber : (y : B) → Type (lmax i j)
  hfiber y = Σ A (λ x → f x == y)
```

The definition of n-connectedness is given as

```
has-conn-fibers : ∀ {i j} {A : Type i} {B : Type j}
                  → ℕ₋₂ → (A → B) → Type (lmax i j)
has-conn-fibers {A = A} {B = B} n f =
  ∏ B (λ b → is-connected n (hfiber f b))
```

There are two propositions about n-connected maps we often use. The first is the induction principle for n-connected maps (Theorem 4.2), which is called `conn-extend` in the library. The other is about the inclusion of a base point in an n-connected type (Proposition 4.3), and is called `pointed-conn-out`. See the file `lib/NConnected.agda` for more details regarding the implementation of these propositions.

5.4. Higher inductive types

Higher inductive types are an essential ingredient for doing synthetic homotopy theory in homotopy type theory. However, the theory of higher inductive types is still an active area of research and as of yet there is no known way to rigorously define them. For this reason, higher inductive types are not natively supported in Agda (for a discussion of higher inductive types and proof assistants, see [6]).

There is a trick to get higher inductive types to work in Agda using postulates and rewriting. The keyword `postulate` can be used to postulate the existence of certain types and functions as an axiom. We use this to postulate the existence of the higher inductive type we want to define. Rewriting rules are used to redefine what a certain function evaluation should evaluate to. We use these to enforce the computation rules of higher inductive types.

To give an example, we show how to define the circle in Agda using this trick. First, we postulate the existence of the type $\mathbb{S}^1$ together with its point constructor `base` and path constructor `loop : base == base`.

```
postulate
  𝕊¹ : Type₀
  base : 𝕊¹
  loop : base == base
```

Next we postulate circle induction:

```
postulate
  circle-ind : ∀ {i} (Y : 𝕊¹ → Type i) (b : Y base)
               (deppath : b == b [ Y ↓ loop ]) →
               (x : 𝕊¹) → Y x
```

Now we need to specify the computation rules of circle induction using rewriting rules. We have two computation rules for circle induction. The first one says that `circle-ind Y b deppath` applied to `base` should yield `b`. We can enforce this by writing

```
postulate
  circle-comp-base : ∀ {i} (Y : 𝕊¹ → Type i) (b : Y base)
                     (deppath : b == b [ Y ↓ loop ]) →
                     (circle-ind Y b deppath base) ↦ b
{-# REWRITE circle-comp-base #-}
```

The other computation rule says that `apd (circle-ind Y b deppath)` applied to `loop` should give back `deppath`. To enforce this, we write

```
postulate
  circle-comp-loop : ∀ {i} (Y : 𝕊¹ → Type i) (b : Y base)
                     (deppath : b == b [ Y ↓ loop ]) →
                     (apd (circle-ind Y b deppath) loop) ↦ deppath
{-# REWRITE circle-comp-loop #-}
```

This completes the definition of the circle. Circle recursion can now be defined by applying circle induction to a constant type family.

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

In this chapter, we will discuss our formalization of the proof that $\pi_1(\mathbb{S}^1) = \mathbb{Z}$. The entire code of the formalization is included in appendix A. The code is contained in three files. The first one, `Circle.agda`, is made up of 35 lines of code and contains a definition of the circle as a higher inductive type. The second, `Integers.agda`, is made up of 216 lines of code and contains a definition of the type of integers, as well as proofs of some of its basic properties. The final one, `FundamentalGroupCircle.agda`, is made up of 244 lines of code and contains the proof of the theorem. Together, these three files span 12 pages. The paper proof contained in chapter 3 is about 4 pages long when we omit the more expository paragraphs, as well as the section on truncations. However, we left out the construction of the integers in the paper proof, while this accounts for almost half of the lines of code. Without the file `Integers.agda`, the formalization is a little over 7 pages in length.

Note that a formalization of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ is included in the HoTT-Agda library. The formalization under consideration in this chapter was written without consulting the one in the library.

6.1. Definition of the integers

In set theory, the integers can be constructed by partitioning $\mathbb{N} \times \mathbb{N}$ into the equivalence classes of the relation $\sim$, where $(a,\ b) \sim (c,\ d)$ if and only if $a + d = c + b$. The pair $(m,\ n) \in \mathbb{N} \times \mathbb{N}$ then becomes a representative of the integer $m - n$. A similar approach is used in the homotopy type theory book [11, pp. 261–266]. This is somewhat more complicated than in set theory, since constructing the quotient of a type by an equivalence relation in general requires the use of higher inductive types. However, in this case, it is possible to describe the type of equivalence classes in a simpler way since the integers have canonical representatives: pairs of the form $(n,\ 0)$ and $(0,\ n)$. This would lead to the following definition in Agda:

```
subtract : ℕ × ℕ → ℕ × ℕ
subtract (m , 0) = (m , 0)
subtract (0 , S n) = (0 , S n)
subtract (S m , S n) = subtract (m , n)

ℤ = Σ (ℕ × ℕ) (λ y → subtract y == y)
```

Essentially, this defines the integers as just the pairs of the form $(n,\ 0)$ or $(0,\ n)$. It is then possible to define a function which sends an arbitrary pair in $\mathbb{N} \times \mathbb{N}$ to the integer it represents as follows:

```
subtract-idemp : (x : ℕ × ℕ) → subtract (subtract x) == subtract x
subtract-idemp (m , 0) = idp
subtract-idemp (0 , S n) = idp
subtract-idemp (S m , S n) = subtract-idemp (m , n)

ℤ-quot : (ℕ × ℕ) → ℤ
ℤ-quot x = (subtract x , subtract-idemp x)
```

It turns out that this definition of the integers is very cumbersome to work with. The main reason is that with this definition, all equalities of integers become propositional equalities in a Σ-type, which are quite complicated pieces of data. Simple statements, such as that the successor function is an equivalence, become difficult to prove.

For this reason, we instead use the following definition of the integers, which is standard in Agda:

```
data ℤ : Type₀ where
  pos : ℕ → ℤ
  negsucc : ℕ → ℤ
```

The constructor `pos` embeds the natural numbers in $\mathbb{Z}$ as the non-negative integers and the constructor `negsucc` embeds the natural numbers in $\mathbb{Z}$ as the negative integers, with `negsucc n` representing the integer $-(n+1)$. The advantage of this definition is that it immediately enables us to prove things about the integers using double induction. Furthermore, when proving basic properties of the integers, many equalities become judgmental. For example, the definition of the successor and predecessor functions and the proof that the successor function is an equivalence, become trivial:

```
succ : ℤ → ℤ
succ (pos x) = pos (S x)
succ (negsucc O) = pos O
succ (negsucc (S x)) = negsucc x

pred : ℤ → ℤ
pred (pos O) = negsucc O
pred (pos (S x)) = pos x
pred (negsucc x) = negsucc (S x)

succ-is-equiv : is-equiv succ
succ-is-equiv = is-eq succ pred succ-pred pred-succ where
  succ-pred : (x : ℤ) → succ (pred x) == x
  succ-pred (pos O) = idp
  succ-pred (pos (S x)) = idp
  succ-pred (negsucc x) = idp

  pred-succ : (x : ℤ) → pred (succ x) == x
  pred-succ (pos x) = idp
  pred-succ (negsucc 0) = idp
  pred-succ (negsucc (S x)) = idp
```

After this, the file `Integers.agda` contains a number of propositions about the integers that are necessary for the proof of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$. First comes the function `p-conc`, which allows for taking integer powers of a loop in an arbitrary type:

```
power-concat : ∀ {i} {X : Type i} (x : X) (p : x == x) → ℤ → x == x
power-concat x p (pos 0) = idp
power-concat x p (pos (S n)) = (power-concat x p (pos n)) · p
power-concat x p (negsucc 0) = ! p
power-concat x p (negsucc (S n)) = (power-concat x p (negsucc n)) · (! p)

p-conc : ∀ {i} {X : Type i} {x : X} (p : x == x) → ℤ → x == x
p-conc {i} {X} {x} p n = power-concat x p n
```

The function `power-concat` is an auxiliary function. The function `p-conc` is more convenient to use, since we do not have to supply the base point `x` as an argument. We also need the following propositions about this function:

```
p-conc-succ : ∀ {i} {X : Type i} {x : X} (p : x == x) (n : ℤ) →
              (p-conc p (succ n)) == (p-conc p n) · p

p-conc-pred : ∀ {i} {X : Type i} {x : X} (p : x == x) (n : ℤ) →
              (p-conc p (pred n)) == (p-conc p n) · (! p)
```

We omit the proofs, which are easy induction arguments combined with some rearrangements of paths.

Next, we prove that $\mathbb{Z}$ is a set (section 2.7). To do so, we make use of Hedberg's theorem, which says that a type with decidable equality is a set. (For a proof of Hedberg's theorem, see [11, pp. 290–294].) A proposition A is decidable if $A + \neg A$ holds. A type X has decidable equality if for any pair of elements $x, y : X$, we have that $x = y$ is decidable. To prove $\mathbb{Z}$ has decidable equality, we first define the following relation on $\mathbb{Z}$:

```
_∼∼_ : (n m : ℤ) → Type₀
pos 0 ∼∼ pos 0 = ⊤
pos 0 ∼∼ pos (S x) = ⊥
pos 0 ∼∼ negsucc x = ⊥
pos (S x) ∼∼ pos 0 = ⊥
pos (S x) ∼∼ pos (S y) = pos x ∼∼ pos y
pos (S x) ∼∼ negsucc y = ⊥
negsucc 0 ∼∼ pos x = ⊥
negsucc 0 ∼∼ negsucc 0 = ⊤
negsucc 0 ∼∼ negsucc (S x) = ⊥
negsucc (S x) ∼∼ pos y = ⊥
negsucc (S x) ∼∼ negsucc O = ⊥
negsucc (S x) ∼∼ negsucc (S y) = negsucc x ∼∼ negsucc y
```

Here, ⊤ is the unit type and ⊥ is the empty type. The idea is that `_∼∼_` gives an inductive characterization of equality in ℤ. Next we prove that `_∼∼_` is decidable:

```
∼∼-dec : (n m : ℤ) → Dec (n ∼∼ m)
```

We omit the proof, which is a straightforward induction argument. The next step is to show that for all `n m : ℤ`, we have that `n ∼∼ m` if and only if `n == m`:

```
ℤ-encode : (n m : ℤ) → (n == m) → (n ∼∼ m)

ℤ-decode : (n m : ℤ) → (n ∼∼ m) → (n == m)
```

The first one is proved using path induction, the second one using double induction. We omit both. Using these propositions, we show ℤ has decidable equality:

```
ℤ-=-dec : (n m : ℤ) → Dec (n == m)
ℤ-=-dec n m = ⊔-rec (inl ∘ (ℤ-decode n m))
                    (inr ∘ (λ g → (g ∘ ℤ-encode n m)))
                    (∼∼-dec n m)
```

Let us analyse this proof. The function `⊔-rec` is the recursion principle for coproduct types, and can be used to give a case by case definition of a function. We use it to define a function of type

```
(n ∼∼ m) ⊔ ¬ (n ∼∼ m) → (n == m) ⊔ ¬ (n == m)
```

Note that the symbol ⊔ is used in the HoTT-Agda library to denote the coproduct. In case `(n ∼∼ m)` holds, we define the function to be `inl ∘ (ℤ-decode n m)`. In case `¬ (n ∼∼ m)` holds, we want to supply a function of type

```
¬ (n ∼∼ m) → ¬ (n == m)
```

and then compose with `inr`. Note that `¬ (n ∼∼ m)` is by definition equal to the type `(n ∼∼ m) → ⊥`. Given an element g of this type, we want to obtain an element of `(n == m) → ⊥`. But we have the function `ℤ-encode n m : (n == m) → (n ∼∼ m)`, so we can compose g with this function to obtain

```
g ∘ ℤ-encode n m : ¬ (n == m)
```

The function obtained by means of `⊔-rec` is then applied to `(∼∼-dec n m)` to obtain an element of `Dec (n == m)`.

To conclude the proof, we apply Hedberg's theorem, which is called `dec-eq-is-set` in the library:

```
ℤ-is-set : is-set ℤ
ℤ-is-set = dec-eq-is-set ℤ-=-dec
```

The final part of the file `Integers.agda` is concerned with integer arithmetic. We start with definitions of integer addition and multiplication by -1:

```
plus : ℤ → ℤ → ℤ
plus (pos O) n = n
plus (pos (S x)) n = succ (plus (pos x) n)
plus (negsucc O) n = pred n
plus (negsucc (S x)) n = pred (plus (negsucc x) n)

_+'_ : ℤ → ℤ → ℤ
m +' n = plus m n

min : ℤ → ℤ
min (pos O) = pos 0
min (pos (S x)) = negsucc x
min (negsucc x) = succ (pos x)
```

We then prove the following properties of integer addition:

```
+'-unit-r : (m : ℤ) → (m +' (pos 0)) == m

+'-succ-r : (m n : ℤ) → (m +' (succ n)) == succ (m +' n)

+'-pred-r : (m n : ℤ) → (m +' (pred n)) == pred (m +' n)
```

We omit the proofs, which are straightforward induction arguments.

6.2. Definition of the circle

We already showed the definition of the circle in section 5.4 of the previous chapter. The only other thing contained in the file `Circle.agda` is the derivation of circle recursion from circle induction. Recall that circle induction was defined as

```
postulate
  circle-ind : ∀ {i} (Y : S¹ → Type i) (b : Y base)
                   (deppath : b == b [ Y ↓ loop ]) →
                   (x : S¹) → Y x
```

Circle recursion says that that given a type `B`, a point `b : B` and a loop `l : b == b`, we get a function `f : S¹ → B` such that `f base == b` and `ap f loop == l`. We prove this as

```
circle-rec : ∀ {i} {B : Type i} (b : B) (l : b == b) → S¹ → B
circle-rec {i} {B} b l = circle-ind (λ _ → B) b (↓-cst-in l)
```

Note that we can write `(λ _ → B)` for the constant type family which sends everything to `B`. The function `↓-cst-in` comes from the library, and turns a regular path into a dependent path in a constant type family.

6.3. Proof of the theorem

In our formalization of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$, our goal is to prove that `(base == base)` $\simeq \mathbb{Z}$ and that the underlying function of this equivalence is a homomorphism, meaning that it sends concatenation of paths to addition of integers. We closely follow the proof we gave in sections 3.4 and 3.5. Recall that we employ the following strategy: we construct a function $\mathsf{code} : \mathbb{S}^1 \to \mathcal{U}$ such that $\mathsf{code}(\mathsf{base}) = \mathbb{Z}$. We then define functions $\mathsf{encode} : (\mathsf{base} = x) \to \mathsf{code}(x)$ and $\mathsf{decode} : \mathsf{code}(x) \to (\mathsf{base} = x)$ which are mutual inverses. This gives an equivalence $(\mathsf{base} = x) \simeq \mathsf{code}(x)$. By instantiating at $x \equiv \mathsf{base}$ we get the desired equivalence.

The first step is the construction of the code function. For this, we use `circle-rec`:

```
code : 𝕊¹ → Type₀
code = circle-rec ℤ (ua succ-eq)
```

Here, `ua` denotes the univalence function, which sends the equivalence `succ-eq` to a loop in ℤ `==` ℤ. Next come the encode and decode functions. The encode function is given by transporting in `code`:

```
encode : (x : 𝕊¹) → (base == x) → (code x)
encode x p = transport code p (pos 0)
```

The definition of the decode function is the core of the proof. We define it using circle induction:

```
decode : (x : 𝕊¹) → (code x) → (base == x)
decode x = circle-ind P (p-conc loop) loop-coherence x
```

Here, `P` is the following type family:

```
P : 𝕊¹ → Type₀
P y = (code y) → (base == y)
```

We already discussed the function `p-conc loop`, which is defined in the file `Integers.agda`. It has type ℤ `→ base == base`, which is equal to `P base` since `code base` is equal to ℤ by definition. The function `p-conc loop` sends an integer `n` to the `n`-th power of `loop`.

The lemma `loop-coherence`, which is necessary to complete the circle induction, has type

```
(p-conc loop) == (p-conc loop) [ P ↓ loop ]
```

To prove this lemma, we first prove the following:

```
transp-path : transport P loop (p-conc loop) == (p-conc loop)
```

And then use the function `from-transp` from the library to turn this into a dependent path:

```
loop-coherence : (p-conc loop) == (p-conc loop) [ P ↓ loop ]
loop-coherence = from-transp P loop transp-path
```

The advantage of working with `transport` instead of dependent paths is that we can use a chain of equalities to prove the statement. We do this as follows:

```
transp-path : transport P loop (p-conc loop) == (p-conc loop)
transp-path =
  (transport P loop (p-conc loop))
    =⟨ (lemma1 code (λ u → base == u) loop (p-conc loop)) ⟩

  ((transport (λ u → base == u) loop ∘ ((p-conc loop)
                                      ∘ transport code (! loop)))
    =⟨ (∘-left-trans ((p-conc loop) ∘ transport code (! loop))
                        (transport (λ u → base == u) loop)
                        (λ p → p · loop) (transport-path-fib loop)) ⟩

  ((λ p → p · loop) ∘ ((p-conc loop) ∘ transport code (! loop)))
    =⟨ ∘-right-trans (transport code (! loop)) pred
                        ((λ p → p · loop) ∘ (p-conc loop))
                        transp-loop-to-pred ⟩

  ((λ p → p · loop) ∘ (p-conc loop) ∘ pred)
    =⟨ idp ⟩

  ((λ n → (p-conc loop (pred n) · loop))
    =⟨ (λ= lemma2) ⟩

  (p-conc loop =■)))
```

We will now go over the individual equalities in this chain. For the first equality, we use following lemma, which characterizes transport in a type family of function types:

```
lemma1 : ∀ {i j k} {X : Type i} (A : X → Type j) (B : X → Type k)
         {x y : X} (p : x == y) (f : A x → B x) →
         transport (λ u → (A u → B u)) p f ==
         (λ y → transport B p (f (transport A (! p) y)))
lemma1 A B idp f = idp
```

We then apply this to the type family `P`, which is indeed a family of function types. For the second and third equalities, we need the following lemmas, which allow us to apply equalities to individual factors of a composition of functions:

```
∘-right-trans : ∀ {i j k} {A : Type i} {B : Type j} {C : Type k}
                (f : A → B) (g : A → B) (h : B → C) (p : f == g)
                → (h ∘ f == h ∘ g)
∘-right-trans f g h idp = idp
```

```
∘-left-trans : ∀ {i j k} {A : Type i} {B : Type j} {C : Type k}
               (f : A → B) (g : B → C) (h : B → C) (p : g == h)
               → (g ∘ f == h ∘ f)
∘-left-trans f g h idp = idp
```

For the second equality, we combine `∘-left-trans` with the following lemma, which characterizes transport in a type family of identity types with one endpoint fixed:

```
transport-path-fib : ∀ {i} {A : Type i} {a x₁ x₂ : A} (p : x₁ == x₂) →
                     transport (λ u → a == u) p == (λ q → q · p)
transport-path-fib idp = λ= (λ q → ! (·-unit-r q))
```

Here, λ= is function extensionality. We need another proposition for the next step. The proof of this proposition is essentially an application of the univalence axiom:

```
loop-to-pred : (x : ℤ) → x == pred x [ code ↓ (! loop) ]
loop-to-pred x = ↓-ap-out (λ y → y) code (! loop)
                (↓-idf-in (ap code (! loop)) t) where
  t : coe (ap code (! loop)) x == pred x
  t = (coe (ap code (! loop)) x)      =⟨ ap2 coe (ap-! code loop) idp ⟩
      (coe (! (ap code loop)) x)      =⟨ coe-! (ap code loop) x ⟩
      (coe! (ap code loop) x)         =⟨ ap2 coe! circle-rec-comp idp ⟩
      (coe! (ua succ-eq) x)           =⟨ coe!-β succ-eq x ⟩
      pred x                          =∎

transp-loop-to-pred : transport code (! loop) == pred
transp-loop-to-pred = λ= (λ x → to-transp (loop-to-pred x))
```

We apply `∘-right-trans` together with `transp-loop-to-pred` to prove the third equality. For clarity, we rewrite the expression in the fourth equality. In the final equality, we combine function extensionality with the following lemma:

```
lemma2 : (x : ℤ) → p-conc loop (pred x) · loop == p-conc loop x
```

We omit the proof, which proceeds by double induction on `x`. This completes the definition of the decode function.

The next step in the proof is to show the encode and decode functions are inverses. One direction is immediate:

```
decode-encode : (x : S¹) (p : base == x) → decode x (encode x p) == p
decode-encode .base idp = idp
```

For the other direction, we make use of circle induction:

```
encode-decode : (x : S¹) (c : code x) → encode x (decode x c) == c
encode-decode =
  circle-ind (λ x → (c : code x) → encode x (decode x c) == c)
             base-case path-coherence2
```

The lemma base-case has type

```
base-case : (m : code base) → encode base (decode base m) == m
```

Note that code base is equal to ℤ. We proved this statement by induction on m. We omit the proof, which is straightforward. The lemma path-coherence2 has type

```
path-coherence2 : base-case == base-case [ Q ↓ loop ]
```

Where Q is the type family

```
Q : 𝕊¹ → Type₀
Q x = (c : code x) → encode x (decode x c) == c
```

Again, we prove the equivalent statement in terms of transport:

```
transport Q loop base-case == base-case
```

For every x : $\mathbb{S}^1$, applying both sides of this equation to x gives an equality of paths in $\mathbb{Z}$. Since $\mathbb{Z}$ is a set, such an equality always holds. This yields the following proof:

```
path-coherence2 : base-case == base-case [ Q ↓ loop ]
path-coherence2 = from-transp P loop (λ= (λ x → set-path ℤ-is-set
                        (transport P loop base-case x) (base-case x)))
```

It now follows immediately that we have an equivalence between code x and base == x:

```
𝕊¹-fiber-equiv : (x : 𝕊¹) → (base == x) ≃ code x
𝕊¹-fiber-equiv x = equiv (encode x) (decode x) (encode-decode x)
                       (λ a → decode-encode x a)
```

By applying this function to base, we then prove the desired equivalence:

```
loopsp-eq : ℤ ≃ (base == base)
loopsp-eq = (<- (𝕊¹-fiber-equiv base)) ,
            (is-equiv-inverse (snd (𝕊¹-fiber-equiv base)))
```

The final step of the proof is to show that this equivalence is a group homomorphism:

```
loopsp-eq-mult : (m n : ℤ) →
   (-> loopsp-eq) (m +' n) == (-> loopsp-eq m) · (-> loopsp-eq n)
```

We omit the proof, which proceeds by induction on n.

7. Formalization of the Freudenthal Suspension Theorem

In this chapter, we will discuss our formalization of the proof of the Freudenthal suspension theorem. The full code of the formalization is included in appendix B. The proof is contained in two files. The file `WedgeConnectivity.agda` contains the proofs of Theorem 4.5 from the text, which I will refer to as the pullback theorem, and the wedge connectivity lemma (Lemma 4.8). The file consists of 236 lines of code. The file `FreudenthalSuspension.agda` contains the proof of the theorem and consists of 534 lines of code. Together, these files span 18 pages, whereas the paper proof is about 6 pages long.

This formalization is based on the proof from the homotopy type theory book [11, pp. 364–371]. While working on the formalization, we found an error in the final part of this proof. Eventually, we were able to repair this. The result is a slightly different proof, which was written directly in Agda. Only later, while writing chapter 4, we informalized the proof. This reversal of the standard process from paper proof to formalization is an indication of the usefulness of proof assistants for homotopy type theory.

Note that a formalization of this theorem is included in the HoTT-Agda library. The formalization under consideration in this chapter was written without consulting the one in the library.

7.1. Proof of the pullback theorem

We recall the statement of the pullback theorem: if we have integers $k \geq n \geq -2$, an n-connected map $f : A \to B$ and a family of k-types $P : B \to k-\mathsf{Type}$, then the pullback map

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

is $(k-n-2)$-truncated.

For the proof, we work in a module, fixing some of the hypotheses of the theorem:

```
module _ {i j} {A : Type i} {B : Type j} (n : ℕ₋₂)
                    (f : A → B) (c : has-conn-fibers n f)
```

It will become clear later why we do not include the integer k as an additional hypothesis to this module. The pullback map is now given as

```
pullback : ∀ {ℓ} (k : ℕ₋₂) (P : B → k -Type ℓ)
           → ∏ B (fst ∘ P) → ∏ A (fst ∘ P ∘ f)
pullback k P s = s ∘ f
```

To be able to state the pullback theorem, we must first define what it means for a map to be n-truncated. We do this in a separate module:

```
module _ {i j} {A : Type i} {B : Type j} (f : A → B) where
  has-trunc-fibers : ℕ₋₂ → Type (lmax i j)
  has-trunc-fibers n = ∏ B λ b → has-level n (hfiber f b)
```

Recall that we proved the pullback theorem in section 4.1 by induction on the natural number $k - n$. To be able to do this in Agda, we introduce a natural number d as a hypothesis, which we think of as $d \equiv k - n$. We can then recover k by setting $k :\equiv d + n$. The statement of the theorem now becomes

```
pullback-theorem : ∀ {ℓ} (d : ℕ) → (P : B → (⟨ d ⟩₋₂ +2+ n)  -Type ℓ)
                  → has-trunc-fibers (pullback (⟨ d ⟩₋₂ +2+ n) P) ⟨ d ⟩₋₂
```

We can now induct on d to prove the theorem. The base case is nothing else than the induction principle for pullbacks along n-connected maps (see Theorem 4.2). This is called `ConnExtend.restr-is-equiv` in the library. This proposition only gives us that the pullback map is an equivalence. This is equivalent to the map being (-2)-truncated, but we need a small proposition to translate between these concepts:

```
module _ {i j} {A : Type i} {B : Type j} (f : A → B) where
  eq-to-contr-fib : is-equiv f → has-trunc-fibers ⟨ 0 ⟩₋₂
  eq-to-contr-fib e = equiv-is-contr-map e
```

We can now prove the base case:

```
pullback-theorem 0 P = eq-to-contr-fib (pullback n P)
                        (ConnExtend.restr-is-equiv c P)
```

For the induction step we have to prove that given

```
x y : hfiber (pullback (S (⟨ d ⟩₋₂ +2+ n)) P) s
```

the type `x == y` is `(d-2)`-truncated. Note that `x` and `y` are pairs `(g, p)`, `(h, q)` with `g, h : ∏ B (fst ∘ P)` and `p : g ∘ f == s`, `q : h ∘ f == s`. To start out, we introduce the following construction which turns a family of `k+1`-types into a family of `k`-types:

```
lower-lvl : ∀ {ℓ} {k : ℕ₋₂} (P : B → (S k) -Type ℓ)
            (g h : ∏ B (fst ∘ P)) → (B → k -Type ℓ)
lower-lvl P g h = λ b → (g b == h b) ,
    (has-level.has-level-apply (snd (P b))) (g b) (h b)
```

We use this construction to define the following type family:

```
lower-lvl-fib : (x y : hfiber (pullback (S (⟨ d ⟩₋₂ +2+ n)) P) s)
   → Type (lmax i (lmax j ℓ))
lower-lvl-fib (g , p) (h , q) = (hfiber (pullback (⟨ d ⟩₋₂ +2+ n)
      (lower-lvl P g h)) (app= (p · (! q))))
```

We can now use the induction hypothesis to show that this is a family of (d-2)-types:

```
eq-fiber-lvl : (x y : hfiber (pullback (S (⟨ d ⟩₋₂ +2+ n)) P) s)
                  → (has-level ⟨ d ⟩₋₂ (lower-lvl-fib x y))
eq-fiber-lvl x y = pullback-theorem d ((lower-lvl P (fst x) (fst y)))
                  (app= (((snd x) · (! (snd y)))))
```

Our strategy now is to show that `x == y` is equivalent to `lower-lvl-fib x y`, which is enough to finish the proof. We will need one lemma:

```
pathover-pb-equiv : ∀ {ℓ} {k : ℕ₋₂} (P : B → k -Type ℓ)
                  → {s : ∏ A (fst ∘ P ∘ f)}
                  → (x y : hfiber (pullback k P) s)
                  → (r : (fst x) == (fst y))
                  → ((snd x) == (snd y) [ (λ φ → (φ ∘ f == s)) ↓ r ]) ≃
                     (((app= r) ∘ f) == (app= ((snd x) · (! (snd y)))))
pathover-pb-equiv P (g , p) (g , q) idp =
  (p == q)
    ≃⟨ (symm-path-is-equiv p q) ⟩
  (q == p)
    ≃⟨ (ap-equiv (·-r-equiv (! q)) q p) ⟩
  (q · (! q)) == p · (! q)
    ≃⟨ ·-l-equiv (! (!-inv-r q)) ⟩
  (idp == p · (! q))
    ≃⟨ ap-equiv (app= , StrongFunextDep.app=-is-equiv) idp (p · (! q)) ⟩
  ((app= idp) == (app= (p · (! q)))) ≃■
```

The proof of this lemma is by path induction on `r`, which reduces the statement to

```
(p == q) ≃ ((app= idp) ∘ f == (app= (p · (! q))))
```

Note that `(app= idp) ∘ f` is the same as `app= idp`, since both are equal to the dependent function that sends `a : A` to `idp : (f a) == (f a)`.

We now prove the equivalence `(x == y) ≃ lower-lvl-fib x y` by exhibiting a chain of equivalences:

```
fib-eq : (x y : hfiber (pullback (S (⟨ d ⟩₋₂ +2+ n)) P) s)
              → (x == y) ≃ lower-lvl-fib x y
fib-eq (g , p) (h , q) =
  ((g , p) == (h , q))

     ≃⟨ ((=Σ-econv (g , p) (h , q)) ⁻¹) ⟩
```

```
((Σ (g == h) (λ r → p == q [ (λ φ → (φ ∘ f == s)) ↓ r ]))

    ≃⟨ (Σ-fmap-r (λ r → -> (pathover-pb-equiv P (g , p) (h , q) r))) ,
          fiber-equiv-is-total-equiv (λ r →
          -> (pathover-pb-equiv P (g , p) (h , q) r))
          (λ r → snd (pathover-pb-equiv P (g , p) (h , q) r)) ⟩
(Σ (g == h) (λ r → ((app= r) ∘ f) == (app= (p · (! q)))))

    ≃⟨ Σ-emap-l (λ r → (r ∘ f) == (app= (p · (! q)))) app=-equiv ⟩
(lower-lvl-fib (g , p) (h , q)) ≃∎)
```

The first equivalence is given by the characterization of equality in Σ-types. The second equivalence is an application of the lemma `pathover-pb-equiv`. The final equivalence is `app=`, applied to `g == h`.

7.2. Proof of the wedge connectivity lemma

We start by recalling the statement of the wedge connectivity lemma (Lemma 4.8). Let $(A,\, a_0)$, $(B,\, b_0)$ be pointed types, such that A is an $m-$Type and B is an $n-$Type, where $m, n \geq 0$. Suppose we have a family

$$P : A \to B \to (m+n)\text{−Type},$$

functions $f : \prod_{(a:A)} P(a,\, b_0)$ and $g : \prod_{(b:B)} P(a_0,\, b)$, and a path $p : f(a_0) = g(b_0)$. Then there exists a function $h : \prod_{(a:A)} \prod_{(b:B)} P(a,\, b)$ such that

$$h(-,\, b_0) = f \qquad \text{and} \qquad h(a_0,\, -) = g.$$

To be able to state the lemma in a more comprehensible form, we introduce the double Π-type, which is just a type of dependent functions with two variables:

```
module _ {i j k} (A : Type i) (B : Type j) (P : A → B → Type k) where
  ∏∏ : Type (lmax i (lmax j k))
  ∏∏ = ∏ A (λ a → ∏ B (λ b → P a b))
```

We give the proof of the wedge connectivity lemma in a module to fix some hypotheses:

```
module _ {i j ℓ} (A : Ptd i) (B : Ptd j) (n m : ℕ)
           (v : is-connected ⟨ n ⟩  (de⊙ A))
           (w : is-connected ⟨ m ⟩ (de⊙ B))
           (P : (de⊙ A) → (de⊙ B) → ⟨ n + m ⟩ -Type ℓ) where
```

We then state the lemma as follows:

```
wedge-conn : (f : ∏ (de⊙ A) λ a → (fst (P a (pt B))))
           → (g : ∏ (de⊙ B) λ b → (fst (P (pt A) b)))
           → (p : g (pt B) == f (pt A))
           → Σ (∏∏ (de⊙ A) (de⊙ B) (λ a b → fst (P a b)))
                 (λ h → (((λ a → h a (pt B)) ∼ f)) ×
                        ((λ b → h (pt A) b) ∼ g))
```

Note that we have packaged the entire conclusion of the lemma in one Σ-type. The first part of the proof is the construction of a function of type

```
∏∏ (de⊙ A) (de⊙ B) (λ a b → fst (P a b))
```

To do so, we first define the following type family:

```
Q : de⊙ A → Type (lmax j ℓ)
Q a = Σ (∏ (de⊙ B) (λ b → fst (P a b))) λ k → k (pt B) == f a
```

The strategy will be to construct a dependent function `s` into this type family. The function `λ a b → fst (s a) b` will then be a dependent function of the required type.

Our first step is to show that for all `a : A`, the type `Q a` is equivalent to the fiber over `f a` of the map

```
pullback ⟨ m ⟩₋₁ (cst (pt B))
    (pointed-conn-out (de⊙ B) (pt B) {{ w }}) ⟨ n + m ⟩ (P a)
```

which is the pullback map along the constant map `cst (pt B)` from the unit type `⊤` to `B`. This pullback map takes functions of type

```
∏ (de⊙ B) (λ b → fst (P a b))
```

to functions of type

```
∏ ⊤ (P a (pt B))
```

So we have to show that:

```
Q-is-fib : (a : de⊙ A) → hfiber (pullback ⟨ m ⟩₋₁ (cst (pt B))
    (pointed-conn-out (de⊙ B) (pt B) {{ w }}) ⟨ n + m ⟩ (P a))
    (cst (f a)) ≃ (Q a)
```

In an informal proof, this step would not be mentioned explicitly, since it is common to tacitly identify functions of type `∏ ⊤ (P a (pt B))` with elements of type `P a (pt B)`, because such a function is determined by the image of the single element of `⊤`. In a formalization, this of course has to be made explicit. We do this by means of the following lemmas:

```
module _ {i} {P : ⊤ → Type i} where
  ⊤-rep : ∏ ⊤ P → P unit
  ⊤-rep s = s unit

  ⊤-rep-is-equiv : is-equiv ⊤-rep
  ⊤-rep-is-equiv = is-eq ⊤-rep (λ x → cst x) (λ _ → idp)
                   λ s → λ= λ x → idp
```

The proof of `Q-is-fib` is essentially an application of these lemmas:

```
Q-is-fib a = (Σ-fmap-r (λ r → ap ⊤-rep)) ,
                 fiber-equiv-is-total-equiv (λ r → ap ⊤-rep)
                 λ r → ap-is-equiv ⊤-rep-is-equiv
                 (cst (r (pt B))) (cst (f a))
```

Having identified `Q a` with the fiber of a certain pullback map, we can now use the pullback theorem, which we proved in the previous section, to show that for all `a : A`, `Q a` is an $(n-1)$-type.

```
Q-has-lvl : (a : (de⊙ A)) → has-level ⟨ n ⟩₋₁ (Q a)
Q-has-lvl a = equiv-preserves-level (Q-is-fib a)
              {{ pullback-theorem ⟨ m ⟩₋₁ ((cst (pt B)))
              ((pointed-conn-out (de⊙ B) (pt B) {{ w }})) (S n)
              (λ b → level-eq (P a b) lemma) (cst (f a)) }}
```

We package this information in the following function:

```
Q-lvl : (de⊙ A) → ⟨ n ⟩₋₁ -Type (lmax j ℓ)
Q-lvl a = (Q a) , (Q-has-lvl a)
```

We are now in a position to construct the desired function. Since `A` is n-connected, the inclusion of the base point `cst (pt A) : ⊤ → A` is $(n-1)$-connected. This is expressed by the library function `pointed-conn-out`. Since `Q` is a family of $(n-1)$-types, we can use the induction principle for connected maps (which is called `conn-extend` in the library). This means we only have to provide an element of `Q (pt A)`. The pair `g, p` is of this type. The proof then becomes:

```
s : (a : de⊙ A) → Q a
s = conn-extend (pointed-conn-out (de⊙ A) (pt A) {{ v }})
     Q-lvl (λ x → g , p)
```

This completes the first part of the proof of the wedge connectivity lemma. For the second part, we have to show that

```
(λ a → fst (s a) (pt B)) ∼ f
```

and that

```
(λ b → fst (s (pt A)) b) ∼ g
```

Note that the function `λ a → snd (s a)` is a proof of the first statement. The second statement follows since we have defined s to satisfy `s (pt A) == g , p`.

```
s-comp' : s (pt A) == g , p
s-comp' = conn-extend-β (pointed-conn-out (de⊙ A) (pt A) {{ v }})
                            Q-lvl (λ x → g , p) unit

s-comp : (λ b → fst (s (pt A)) b) ∼ g
s-comp = λ b → app= (ap fst s-comp') b
```

By combining everything, we get a proof of the wedge connectivity lemma:

```
wedge-conn f g p = (λ a b → fst (s a) b) ,
                   ((λ a → snd (s a)) , s-comp)
```

7.3. Proof of the theorem

We now come to the proof of the Freudenthal supension theorem. First we recall the statement of this theorem. Let $n \geq 0$ and let $(X,\ x_0)$ be an n-connected pointed type. Then the Freudenthal suspension theorem says that the map

$$\sigma\colon X \to \Omega\Sigma X, \qquad x \mapsto \mathsf{merid}(x) \cdot \mathsf{merid}(x_0)^{-1}$$

is $2n$-connected.

We work in a module to fix hypotheses of the theorem:

```
module _ {i} (X : Ptd i) (n : ℕ) (c : is-connected ⟨ n ⟩ (de⊙ X))
```

The map σ is called `σloop` in the library and is defined as follows:

```
σloop : ∀ {i} (X : Ptd i) → de⊙ X → north' (de⊙ X) == north' (de⊙ X)
σloop X x = merid x · ! (merid (pt X))
```

For our proof, we will modify this definition slightly. We already mentioned in section 5.3 that using the alternative definition of path concatenation from the library simplifies some of our proofs. Therefore we will use `_·'_` instead of `_·_` in all the proofs of this section. The definition of `σloop` we use is

```
σloop' : de⊙ X → north' (de⊙ X) == north' (de⊙ X)
σloop' x = merid x ·' ! (merid (pt X))
```

We also prove the following lemma, which allows us to prove the Freudenthal suspension theorem for the library version of `σloop` after we have proved it for the alternative version:

```
σloop-eq : (σloop X) == σloop'
σloop-eq = λ= (λ x → ·=·' (merid x) (! (merid (pt X))))
```

We will first give a quick overview of the proof of the suspension theorem. We start by constructing a type family

```
code : (y : Susp (de⊙ X)) → (north == y) → Type i
```

such that `code north` is equal to

```
λ p → Trunc ⟨ n *2 ⟩ (hfiber σloop' p)
```

We then proof that for all `y : Susp (de⊙ X)` and `q : north == y`, the type `code y q` is contractible. By setting y equal to `north`, we have then proved the theorem.

The construction of the code function proceeds by suspension induction. This means that we first have to specify the values this function takes on `north' (de⊙ X)` and `south' (de⊙ X)`:

```
code-N : (p : north == north) → Type i
code-N p = Trunc ⟨ n *2 ⟩ (hfiber σloop' p)

code-S : (q : (north' (de⊙ X)) == south) → Type i
code-S q = Trunc ⟨ n *2 ⟩ (hfiber merid q)
```

To finish the construction of code, we have to give for each `x : de⊙ X` a dependent path from `code-N` to `code-S` over `merid x`:

```
code-coherence : (x : de⊙ X) → code-N == code-S
                           [ (λ z → north == z → Type i) ↓ merid x ]
```

Providing these dependent paths is the most elaborate part of the proof. We first show that for `x : X`, this type of dependent paths is equivalent to the type

```
∏ (north == south) (λ p → ((code-N (p ·' (! (merid x))) ≃ (code-S p)))
```

Instead of proving this directly, we prove a more general statement, which we will also be needing later on:

```
paths-↓-reduce : {y : Susp (de⊙ X)} (q : north == y)
  (C-N : (north == north) → Type i) (C-y : (north == y) → Type i) →
  (C-N == C-y [ (λ z → north == z → Type i) ↓ q ]) ≃
  (∏ (north == y) (λ p → ((C-N (p ·' (! q))) ≃ (C-y p))))
```

By setting y equal to `south`, q equal to `merid x`, `C-N` equal to `code-N`, and `C-y` equal to `code-S`, we have deduced the desired equivalence.

The next step is to provide a family of functions of the following form

```
∏ (north == south) (λ p → ((code-N (p ·' (! (merid x))) → (code-S p)))
```

and prove these functions are equivalences for all `p : north == south`. Since `code-N` and `code-S` are families of $2n$-types, we can appeal to the universal property of such types. It is then enough to provide for all `q : north == south` and all `x₁ x₂ : X` a function

```
((merid x₂) ·' (! (merid (pt X))) == q ·' (! (merid x₁))) → (code-S q)
```

We use the wedge connectivity lemma to provide these functions. First we define the following family of $(n+n)$-types:

```
fun-fam-aux : (q : north == south) → (de⊙ X) → (de⊙ X)
               → (⟨ n *2 ⟩ -Type i)
fun-fam-aux q x₁ x₂ =
  (((merid x₂) ·' (! (merid (pt X))) == q ·' (! (merid x₁)))
          → (code-S q)) , (∏-level (λ _ → Trunc-level))

fun-fam : (q : north == south) → (de⊙ X) → (de⊙ X)
               → (⟨ n + n ⟩ -Type i)
fun-fam q x₁ x₂ = level-eq (fun-fam-aux q x₁ x₂) (ap ⟨_⟩ (+-lemma n))
```

We split this definition in two functions, since `n *2` is only propositionally equal to `n + n`, so we explicitly need to translate from a family of `(n *2)`-types to a family of `(n+n)`-types. Now that we have a family of `(n+n)`-types, we can apply the wedge connectivity lemma, since `X` is `n`-connected. To apply the lemma, we need to provide functions when x_2 is equal to `pt X` and when x_1 is equal to `pt X`:

```
wedge-1 : (q : north == south) (x₂ : de⊙ X)
          → ((merid x₂) ·' (! (merid (pt X))) == q ·' (! (merid (pt X))))
          → (code-S q)
wedge-1 q x₂ r = [ (x₂ , cancel-·'-r (merid x₂) q (! (merid (pt X))) r) ]

wedge-2 : (q : north == south) → (x₁ : de⊙ X)
          → ((merid (pt X)) ·' (! (merid (pt X))) == q ·' (! (merid x₁)))
          → (code-S q)
wedge-2 q x₁ r = [ (x₁ , shuffle-·'-inv (merid (pt X)) q (merid x₁) r) ]
```

Here, we have employed the following lemmas:

```
module _ {i} {A : Type i} where

  cancel-·'-r : {a₁ a₂ a₃ : A} (p q : a₁ == a₂) (w : a₂ == a₃)
                (r : p ·' w == q ·' w) → (p == q)
  cancel-·'-r p q idp r = r

  shuffle-·'-r : {a₁ a₂ : A} (p q : a₁ == a₂)
                 (r : idp == q ·' (! p)) → (p == q)
  shuffle-·'-r idp q r = r

  shuffle-·'-inv : {a₁ a₂ a₃ : A} (p : a₁ == a₂) (q w : a₁ == a₃)
                   (r : p ·' (! p) == q ·' (! w)) → (w == q)
  shuffle-·'-inv idp q w r = shuffle-·'-r w q r
```

This is one of the places where using `_·'_` instead of `_·_` greatly simplified the proofs. To complete the application of the wedge connectivity lemma, we need to show that `wedge-1` and `wedge-2` are equal when both x_1 and x_2 are equal to `pt X`:

```
wedge-12 : (q : north == south) →
           (wedge-1 q (pt X)) == (wedge-2 q (pt X))
wedge-12 q =
  λ= (λ r → ap [_] (pair= idp (app= (shuffle=cancel (merid (pt X)) q) r)))
```

The proof is an application of the following lemma

```
module _ {i} {A : Type i} where
  shuffle=cancel : {a₁ a₂ : A} (p q : a₁ == a₂) →
                   (cancel-·'-r p q (! p)) == (shuffle-·'-inv p q p)
  shuffle=cancel idp q = λ= (λ r → idp)
```

We now can produce the desired family of functions:

```
wedged-aux : (q : north == south) →
  Σ (∏∏ (de⊙ X) (de⊙ X) (λ x₁ x₂ → fst (fun-fam q x₁ x₂)))
    (λ h → (((λ x₁ → h x₁ (pt X)) ∼ wedge-2 q)) ×
           ((λ x₂ → h (pt X) x₂) ∼ wedge-1 q))
wedged-aux q = wedge-conn X X n n c c (fun-fam q)
                          (wedge-2 q) (wedge-1 q) (wedge-12 q)

wedged : (q : north == south) (x₁ x₂ : (de⊙ X))
    → ((merid x₂) ·' (! (merid (pt X))) == q ·' (! (merid x₁)))
    → (code-S q)
wedged q = fst (wedged-aux q)

wedged-tr : (q : north == south) (x₁ : (de⊙ X))
            → (code-N (q ·' (! (merid x₁)))) → (code-S q)
wedged-tr q x₁ = Trunc-rec λ t → wedged q x₁ (fst t) (snd t)
```

The next step is to show that for all `q : north == south` and `x₁ : (de⊙ X)`, the function `wedged-tr q x₁` is an equivalence. Being an equivalence is a mere proposition (which is a (-1)-type) and X is n-connected for $n \geq 0$, so the inclusion of the base point in `X` is at least (-1)-connected. We can therefore use the induction principle for connected maps, and only have to prove the map is an equivalence when `x₁` is equal to `pt X`. But since this map is constructed by means of the wedge connectivity lemma, we know that in this case, the map is equal to the truncation of `wedge-1`. This final step is established in the following lemmas.

```
wedge-1-tr : (q : north == south)
    → (code-N (q ·' (! (merid (pt X))))) → (code-S q)
wedge-1-tr q = Trunc-rec (λ t → wedge-1 q (fst t) (snd t))

wedged-pt : (q : north == south) (x₂ : de⊙ X)
        → ((merid x₂) ·' (! (merid (pt X))) == q ·' (! (merid (pt X))))
        → (code-S q)
```

```
wedged-pt q x2 = wedged q (pt X) x2

wedged-pt-to-wedge-1-aux : (q : north == south)
                         → (wedged-pt q) ∼ (wedge-1 q)
wedged-pt-to-wedge-1-aux q = snd (snd (wedged-aux q))

wedged-pt-to-wedge-1 : (q : north == south) →
                     (λ t → wedged-pt q (fst t) (snd t)) ∼
                     (λ t → wedge-1 q (fst t) (snd t))
wedged-pt-to-wedge-1 q =
      λ t → app= (wedged-pt-to-wedge-1-aux q (fst t)) (snd t)

wedged-tr-pt-to-wedge-1-tr : (q : north == south)
                           → (wedged-tr-pt q) ∼ (wedge-1-tr q)
wedged-tr-pt-to-wedge-1-tr q =
      trunc-pres-∼ Trunc-level (wedged-pt-to-wedge-1 q)
```

What remains is to prove that `wedge-1-tr` is an equivalence. We first prove that `wedge-1` is an equivalence and then show that it stays an equivalence after truncation. The function `wedge-1` essentially consists of cancelling a path on the right on both sides of a propositional equality. The inverse is given by

```
wedge-1-inv : (q : north == south) (x2 : de⊙ X) →
  (merid x2) == q → (code-N (q ·' (! (merid (pt X)))))
wedge-1-inv q x2 r =
    [ (x2 , uncancel-·'-r (merid x2) q (! (merid (pt X))) r) ]
```

Here, `uncancel-·'-r` is the following function

```
module _ {i} {A : Type i} where
  uncancel-·'-r : {a1 a2 a3 : A} (p q : a1 == a2) (w : a2 == a3) →
                  (p == q) → p ·' w == q ·' w
  uncancel-·'-r p q idp r = r
```

The proof that `wedge-1-inv` is a two-sided inverse to `wedge-1` follows from the fact that `uncancel-·'-r` is a two-sided inverse to `cancel-·'-r`, which can be immediately established by path induction:

```
module _ {i} {A : Type i} where
  unc-c : {a1 a2 a3 : A} (p q : a1 == a2) (w : a2 == a3)
    (r : p ·' w == q ·' w) →
    (uncancel-·'-r p q w (cancel-·'-r p q w r)) == r
  unc-c p q idp r = idp

  c-unc : {a1 a2 a3 : A} (p q : a1 == a2) (w : a2 == a3)
    (r : p == q) → (cancel-·'-r p q w (uncancel-·'-r p q w r)) == r
  c-unc p q idp r = idp
```

Again, these proofs would be more complicated if we had used `_·_` instead of `_·'_`. The proof that `wedge-1` and `wedge-1-inv` are inverses now follows immediately:

```
wedge-1-do-undo : (q : north == south)
  (t : hfiber merid q) →
  (wedge-1-tr q (wedge-1-tr-inv q [ t ])) == [ t ]
wedge-1-do-undo q (x₂ , r) =
  ap [_] (pair= idp (c-unc (merid x₂) q (! (merid (pt X))) r))

wedge-1-undo-do : (q : north == south)
  (t : hfiber σloop' (q ·' (! (merid (pt X))))) →
  (wedge-1-tr-inv q (wedge-1-tr q [ t ])) == [ t ]
wedge-1-undo-do q (x₂ , r) =
  ap [_] (pair= idp (unc-c (merid x₂) q (! (merid (pt X))) r))
```

It follows from these propositions that the truncated version of `wedge-1-inv` is a two-sided inverse of `wedge-1-tr`, and that therefore `wedge-1-tr` is an equivalence:

```
wedge-1-tr-inv : (q : north == south)
  → (code-S q) → (code-N (q ·' (! (merid (pt X)))))
wedge-1-tr-inv q = Trunc-rec λ u → wedge-1-inv q (fst u) (snd u)

wedge-1-tr-do-undo : (q : north == south)
  (t : code-S q) → (wedge-1-tr q (wedge-1-tr-inv q t)) == t
wedge-1-tr-do-undo q =
  Trunc-elim-aux (λ t → (wedge-1-tr q (wedge-1-tr-inv q t)) == t)
          (λ t → Trunc-=-level (wedge-1-tr q (wedge-1-tr-inv q t)) t)
          (wedge-1-do-undo q)

wedge-1-tr-undo-do : (q : north == south)
  (t : code-N (q ·' (! (merid (pt X)))))
  → (wedge-1-tr-inv q (wedge-1-tr q t)) == t
wedge-1-tr-undo-do q =
  Trunc-elim-aux (λ t → (wedge-1-tr-inv q (wedge-1-tr q t)) == t)
          (λ t → Trunc-=-level (wedge-1-tr-inv q (wedge-1-tr q t)) t)
          (wedge-1-undo-do q)

wedge-1-tr-is-equiv : (q : north == south) → is-equiv (wedge-1-tr q)
wedge-1-tr-is-equiv q = is-eq (wedge-1-tr q) (wedge-1-tr-inv q)
                              (wedge-1-tr-do-undo q)
                              (wedge-1-tr-undo-do q)
```

Like we argued before, this implies that `wedged-tr` is an equivalence:

```
wedged-tr-pt-is-equiv : (q : north == south)
                      → is-equiv (wedged-tr-pt q)
```

```
wedged-tr-pt-is-equiv q =
  ∼-preserves-equiv (∼-! (wedged-tr-pt-to-wedge-1-tr q))
                   (wedge-1-tr-is-equiv q)

equiv-fam : (x : de⊙ X) → (⟨ 0 ⟩₋₁ -Type i)
equiv-fam x = (∏ (north == south) (λ q → is-equiv (wedged-tr q x))) ,
              ∏-level (λ q → is-equiv-is-prop)

wedged-tr-is-equiv :
   (x : de⊙ X) (q : north == south) → is-equiv (wedged-tr q x)
wedged-tr-is-equiv =
  conn-extend (pointed-conn-out (de⊙ X) (pt X) {{n-to-0-conn c}})
                        equiv-fam λ _ q → wedged-tr-pt-is-equiv q

wedged-tr-equiv : (x : de⊙ X) (q : north == south) →
  code-N (q ∙' (! (merid x))) ≃ code-S q
wedged-tr-equiv x q = (wedged-tr q x) , (wedged-tr-is-equiv x q)
```

This completes the definition of code:

```
code-coherence : (x : de⊙ X) → code-N == code-S
                         [ (λ z → north == z → Type i) ↓ merid x ]
code-coherence x = <- (paths-↓-reduce (merid x) (code-N) (code-S))
                      λ q → wedged-tr-equiv x q

code : (y : Susp (de⊙ X)) → (north == y) → Type i
code = Susp-elim code-N code-S code-coherence
```

The next part of the proof consists of showing that for all y : Susp (de⊙ X) and q : north == y, the type code y q is contractible. The first step is to define a center of contraction. We first define this for code north idp as

```
[ pt X , !-inv'-r (merid (pt X)) ]
```

We then transport this to code y q:

```
code-transp : (y : Susp (de⊙ X)) (q : north == y)
              → (code north idp) → (code y q)
code-transp y q = transport (uncurry code) (pair= q (tid q))

code-center : (y : Susp (de⊙ X)) (q : north == y) → code y q
code-center y q = code-transp y q [ pt X , !-inv'-r (merid (pt X)) ]
```

Here, tid q is the following dependent path

```
tid : {x y : A} (p : x == y) → idp == p [ (λ v → x == v) ↓ p ]
tid idp = idp
```

To finish the proof that `code y q` is contractible, we have to show the following:

```
code-contraction : (y : Susp (de⊙ X)) (q : north == y) (v : code y q)
                    → (code-center y q) == v
```

Since the proof of this theorem we gave in chapter 4 (where it is Theorem 4.11) is based on and closely follows the formalization, we omit the proof here.

8. Conclusion

In the first part of this thesis, we have studied homotopy type theory as a foundational system for synthetic homotopy theory. We have shown that by interpreting inhabitants of identity types as paths, we can interpret types as spaces, about which one can prove things using simple principles like path induction and the univalence axiom. We have also shown how familiar spaces and operations on spaces can be defined by means of higher inductive types. We then proceeded to give proofs of $\pi_1(\mathbb{S}^1) = \mathbb{Z}$ and the Freudenthal suspension theorem in homotopy type theory.

We want to mention that there are many more results from homotopy theory which have been proved in homotopy type theory. Some highlights are the construction of singular cohomology (see [4]), the proof that $\pi_4(\mathbb{S}^3) = \mathbb{Z}/2\mathbb{Z}$ (see [2]), and the construction of the Serre spectral sequence (see [5]). However, the classical counterparts of these results are over 60 years old. It would be interesting to see synthetic proofs of more modern results from homotopy theory.

In the second part of this thesis, we have introduced the proof assistant Agda and have shown how it can be used to write formalizations of proofs from homotopy theory. We then discussed formalizations we made of the two main theorems from the first part of the thesis. We expected beforehand it would be straightforward to formalize these theorems, because it is often claimed that informal proofs in homotopy type theory are a lot closer to formal proofs than in set-theoretic mathematics. Indeed, it has been our experience that many informal proofs from homotopy type theory are readily translated into computer code. However, it was surprising to see how much is still left implicit in some informal proofs. For this reason, writing a formalization in homotopy type theory remains a difficult and time-consuming enterprise. Another surprise was that proof assistants are useful not only for verifying proofs, but also as an aide for finding new proofs, as we learned when trying to fix an error in the proof of the Freudenthal suspension theorem.

It remains to be seen whether researchers in homotopy theory will start using homotopy type theory to formalize their proofs. We do not think this will happen very soon, since the body of results that have been formalized is still too small, and the time it takes to formalize a proof is still too long. However, it is clear that homotopy type theory is an example of how alternative foundations can simplify the process of computer formalization. For this reason, we believe that further research into homotopy type theory could lead to a foundational system that will be adopted by research mathematicians working in homotopy theory for formalizing their proofs.

Bibliography

[1] John Baez, Michael Shulman. *Lectures on n-Categories and Cohomology.* `https://arxiv.org/abs/math/0608420`.

[2] Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory* `https://arxiv.org/abs/1606.05916`. 2016.

[3] Guillaume Brunerie, Favonia, Evan Cavallo et al. *The HoTT-Agda library.* `https://github.com/HoTT/HoTT-Agda`

[4] Evan Cavallo. *Synthetic Cohomology in Homotopy Type Theory.* `http://www.cs.cmu.edu/ rwh/theses/cavallo-msc.pdf`. 2015.

[5] Floris van Doorn. *On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory.* `https://florisvandoorn.com/papers/dissertation.pdf`. 2018.

[6] Kuen-Bang Hou (Favonia). *Higher-Dimensional Types in the Mechanization of Homotopy Theory.* `https://favonia.org/files/thesis.pdf`. School of Computer Science, Carnegie Mellon University, 2017.

[7] Allen Hatcher. *Algebraic Topology.* Cambridge University Press, 2018.

[8] Egbert Rijke, Bas Spitters. *Homotopy type theory and the formalization of mathematics.* Nieuw Archief voor Wiskunde 5/17 (2016), 159–164.

[9] Michael Shulman. *Homotopy Type Theory: A synthetic approach to higher equalities.* `https://arxiv.org/abs/1601.05035`.

[10] Michael Shulman. *Homotopy type theory: the logic of space.* `https://arxiv.org/abs/1703.03007`.

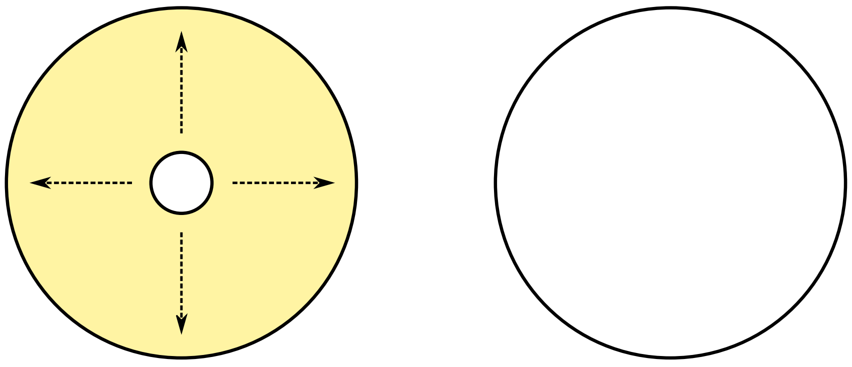
[11] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`. Institute for Advanced Study, 2013.

Populaire samenvatting

Iedereen leert op de middelbare school twee soorten vlakke meetkunde: synthetische meetkunde en analytische meetkunde. Synthetische meetkunde is meetkunde zoals hij door de oude Grieken beoefend werd. De elementaire objecten zijn punten, lijnen en cirkels en er zijn axioma's over hoe deze objecten zich gedragen. De vraag: "wat is een cirkel?" kan je niet beantwoorden in de synthetische meetkunde. Er zijn in deze theorie geen simpelere concepten aan de hand waarvan je een cirkel kan definiëren. In de analytische meetkunde is dit anders. Hierbij worden coördinaten geïntroduceerd waarmee punten in het vlak beschreven kunnen worden. Alle meetkundige figuren zijn nu te definiëren aan de hand van coördinaten en rekenkundige operaties. De cirkel om de oorsprong met straal 1 is nu bijvoorbeeld de verzameling van alle punten uit het vlak met coördinaten (x, y) die voldoen aan de vergelijking $x^2 + y^2 = 1$.

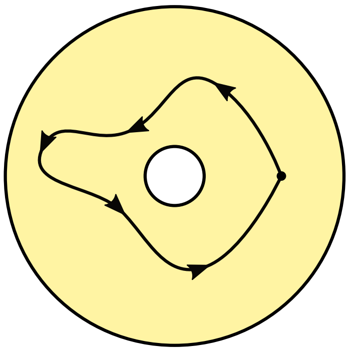
Tegenwoordig zijn er veel meer meetkundige theorieën dan alleen de synthetische en analytische vlakke meetkunde. Al deze theorieën zijn analytisch. Dat wil zeggen dat alle meetkundige concepten in deze theorieën terug te voeren zijn op simpelere concepten, die geen meetkundige betekenis hebben. Je zou kunnen proberen om een van deze meetkundige theorieën te nemen en er een synthetische beschrijving van te geven. In mijn scriptie heb ik dit gedaan voor een bepaalde meetkundige theorie: de homotopietheorie.

Homotopietheorie is de studie van eigenschappen van meetkundige figuren die behouden blijven als de figuur een beetje vervormd wordt, zonder de figuur te scheuren. Een schijfje met een gat in het midden is bijvoorbeeld hetzelfde als een cirkel, omdat we het gat steeds groter kunnen maken totdat we een cirkel hebben. Maar een cirkel kan niet in een lijnstuk veranderd worden, omdat je de cirkel daarvoor moet openknippen.



Figuur 8.1.: Een schijfje met een gat kan vervormd worden tot een cirkel.

Een effectieve manier om figuren in de homotopietheorie te onderzoeken, is door lussen te tekenen. Als we in het schijfje een lus tekenen die niet om het gat heen gaat, kunnen we deze altijd tot een punt samentrekken. Maar als we dit proberen met een lus die wel om het gat heen gaat, blijft deze als een lasso aan het gat haken. Het blijkt dat we hieruit kunnen concluderen dat het schijfje altijd een gat blijft houden, hoe we het ook vervormen. Dit lijkt misschien een hele ingewikkelde methode om zoiets simpels vast te stellen, maar deze techniek werkt ook voor vreemde, hoger-dimensionale figuren die veel moeilijker voor te stellen zijn.



Figuur 8.2.: Een lus om het gat van het schijfje.

We kunnen een synthetische beschrijving van de homotopietheorie geven door een theorie op te zetten waarin lussen en paden tussen punten de meest elementaire objecten zijn, zoals lijnen en cirkels dit waren in de synthetische vlakke meetkunde. Dit werd zo'n tien jaar geleden voor het eerst door wiskundigen gedaan, en leverde een theorie op die *homotopietypentheorie* heet. In de eerste helft van mijn scriptie geef ik een inleiding in de homotopietypentheorie, en bewijs ik twee klassieke stellingen uit de homotopietheorie met synthetische methodes.

Een groot voordeel van deze synthetische benadering is dat hij veel beter geschikt is om computerprogramma's te schrijven die bewijzen uit de homotopietheorie op correctheid toetsen. Dit was een van de belangrijke redenen voor Vladimir Voevodski – een beroemd wiskundige die aan de wieg stond van de homotopietypentheorie – om de synthetische homotopietheorie te onderzoeken. Hij werd beroemd door zijn bewijs van het Bloch-Kato vermoeden, waar hij in 2002 de Fields-medaille voor kreeg. Aan dit honderd pagina's lange bewijs werkte hij tien jaar. Toen er in twee andere artikelen van hem fouten werden ontdekt, werd hij bang dat zijn bewijs van het Bloch-Kato vermoeden ook onjuistheden bevatte. Hoewel deze nooit gevonden zijn, kwam hij tot de conclusie dat zijn vakgebied te ingewikkeld was geworden om zeker te zijn dat je geen fouten over het hoofd ziet. Hij zag het als een noodzaak om een gemakkelijk bruikbaar systeem te ontwikkelen waarmee bewijzen uit de homotopietheorie door de computer geverifieerd kunnen worden. Dit werd homotopietypentheorie.

Voor het tweede deel van mijn scriptie heb ik computerprogramma's geschreven die de twee bewijzen uit het eerste deel van mijn scriptie verifiëren.

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

This appendix contains the Agda code of my formalization of the proof that $\pi_1(\mathbb{S}^1) = \mathbb{Z}$.
The first file, `Circle.agda`, contains a definition of the circle as a higher inductive type.

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

open import lib.Basics

{- This module contains a definition of the higher inductive type of the
    Circle including the propositional induction and recursion rules. -}

module Circle where

postulate  --HIT
  𝕊¹ : Type₀
  base : 𝕊¹
  loop : base == base
  circle-ind : ∀ {i} (Y : 𝕊¹ → Type i) (b : Y base)
               (deppath : b == b [ Y ↓ loop ]) →
               (x : 𝕊¹) → Y x

  circle-comp-base : ∀ {i} (Y : 𝕊¹ → Type i) (b : Y base)
                     (deppath : b == b [ Y ↓ loop ]) →
                     (circle-ind Y b deppath base) ↦ b
{-# REWRITE circle-comp-base #-}

postulate  --HIT
  circle-comp-loop : ∀ {i} (Y : 𝕊¹ → Type i) (b : Y base)
                     (deppath : b == b [ Y ↓ loop ]) →
                     (apd (circle-ind Y b deppath) loop) ↦ deppath
{-# REWRITE circle-comp-loop #-}

{- circle recursion -}
circle-rec : ∀ {i} {B : Type i} (b : B) (l : b == b) → 𝕊¹ → B
circle-rec {i} {B} b l = circle-ind (λ _ → B) b (↓-cst-in l)

circle-rec-comp : ∀ {i} {B : Type i} {b : B} {l : b == b} →
```

```
                      (ap (circle-rec {i} {B} b l) loop == l)
circle-rec-comp {i} {B} {b} {l} = apd=cst-in idp
```

The file `Integers.agda` contains a definitions of the integers together with proof of some basic properties of this type.

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

open import lib.Basics
open import lib.types.Sigma
open import lib.types.Nat
open import lib.types.Coproduct

module Integers where

{- Definitions of integers. negsucc sends 0 to -1 -}
data ℤ : Type₀ where
  pos : ℕ → ℤ
  negsucc : ℕ → ℤ

{-# BUILTIN INTEGER       ℤ    #-}
{-# BUILTIN INTEGERPOS    pos    #-}
{-# BUILTIN INTEGERNEGSUC negsucc #-}

succ : ℤ → ℤ
succ (pos x) = pos (S x)
succ (negsucc O) = pos O
succ (negsucc (S x)) = negsucc x

pred : ℤ → ℤ
pred (pos O) = negsucc O
pred (pos (S x)) = pos x
pred (negsucc x) = negsucc (S x)

succ-is-equiv : is-equiv succ
succ-is-equiv = is-eq succ pred succ-pred pred-succ where
  succ-pred : (x : ℤ) → succ (pred x) == x
  succ-pred (pos O) = idp
  succ-pred (pos (S x)) = idp
  succ-pred (negsucc x) = idp

  pred-succ : (x : ℤ) → pred (succ x) == x
  pred-succ (pos x) = idp
  pred-succ (negsucc O) = idp
  pred-succ (negsucc (S x)) = idp

succ-eq : ℤ ≃ ℤ
succ-eq = succ , succ-is-equiv

pred-eq : ℤ ≃ ℤ
pred-eq = (succ-eq)⁻¹

{- Power concat: take the nth power of a loop p, where n is in ℤ -}
power-concat : ∀ {i} {X : Type i} (x : X) (p : x == x) → ℤ → x == x
power-concat x p (pos O) = idp
power-concat x p (pos (S n)) = (power-concat x p (pos n)) · p
power-concat x p (negsucc O) = ! p
power-concat x p (negsucc (S n)) = (power-concat x p (negsucc n)) · (! p)

p-conc : ∀ {i} {X : Type i} {x : X} (p : x == x) → ℤ → x == x
p-conc {i} {X} {x} p n = power-concat x p n

p-conc-succ : ∀ {i} {X : Type i} {x : X} (p : x == x) (n : ℤ) →
                (p-conc p (succ n)) == (p-conc p n) · p
p-conc-succ p (pos m) = idp
p-conc-succ p (negsucc O) = ! (!-inv-l p)
p-conc-succ p (negsucc (S m)) =
  (p-conc p (negsucc m))
    =⟨ (! (·-unit-r (p-conc p (negsucc m)))) ⟩

  (((p-conc p (negsucc m)) · idp)
    =⟨ ((p-conc p (negsucc m)) ·□ (! (!-inv-l p))) ⟩

  (((p-conc p (negsucc m)) · ((! p) · p))
    =⟨ ! (·-assoc (p-conc p (negsucc m)) (! p) p) ⟩

  (((p-conc p (negsucc m)) · (! p)) · p) =■))

p-conc-pred : ∀ {i} {X : Type i} {x : X} (p : x == x) (n : ℤ) →
                (p-conc p (pred n)) == (p-conc p n) · (! p)
p-conc-pred p (pos O) = idp
p-conc-pred p (pos (S n)) =
  (p-conc p (pos n))
    =⟨ (! (·-unit-r (p-conc p (pos n)))) ⟩

  (((p-conc p (pos n)) · idp)
    =⟨ ((p-conc p (pos n)) ·□ (! (!-inv-r p))) ⟩

  ((p-conc p (pos n)) · (p · (! p))
    =⟨ ! (·-assoc (p-conc p (pos n)) p (! p)) ⟩

  (((p-conc p (pos n)) · p) · (! p)) =■))
p-conc-pred p (negsucc n) = idp

-- Encode-decode for = of integers. We do this to show the integers
-- have decidable equality. From this it follows ℤ is a set.
-- We need this in the proof of π₁(𝕊¹) = ℤ

_∼∼_ : (n m : ℤ) → Type₀
pos 0 ∼∼ pos 0 = ⊤
pos 0 ∼∼ pos (S x) = ⊥
pos 0 ∼∼ negsucc x = ⊥
pos (S x) ∼∼ pos 0 = ⊥
pos (S x) ∼∼ pos (S y) = pos x ∼∼ pos y
pos (S x) ∼∼ negsucc y = ⊥
negsucc 0 ∼∼ pos x = ⊥
negsucc 0 ∼∼ negsucc 0 = ⊤
negsucc 0 ∼∼ negsucc (S x) = ⊥
negsucc (S x) ∼∼ pos y = ⊥
negsucc (S x) ∼∼ negsucc 0 = ⊥
negsucc (S x) ∼∼ negsucc (S y) = negsucc x ∼∼ negsucc y

∼∼-refl : (n : ℤ) → (n ∼∼ n)
∼∼-refl (pos 0) = unit
∼∼-refl (pos (S x)) = ∼∼-refl (pos x)
∼∼-refl (negsucc 0) = unit
∼∼-refl (negsucc (S x)) = ∼∼-refl (negsucc x)

ℤ-encode : (n m : ℤ) → (n == m) → (n ∼∼ m)
ℤ-encode n .n idp = ∼∼-refl n

ℤ-decode : (n m : ℤ) → (n ∼∼ m) → (n == m)
ℤ-decode (pos 0) (pos 0) y = idp
ℤ-decode (pos 0) (pos (S x)) ()
ℤ-decode (pos 0) (negsucc x) ()
ℤ-decode (pos (S x)) (pos 0) ()
ℤ-decode (pos (S x)) (pos (S z)) y =
  ap succ (ℤ-decode (pos x) (pos z) y)
ℤ-decode (pos (S x)) (negsucc 0) ()
ℤ-decode (pos (S x)) (negsucc (S z)) ()
ℤ-decode (negsucc 0) (pos z) ()
ℤ-decode (negsucc (S x)) (pos z) ()
```

```
ℤ-decode (negsucc O) (negsucc O) y = idp
ℤ-decode (negsucc O) (negsucc (S z)) ()
ℤ-decode (negsucc (S x)) (negsucc O) ()
ℤ-decode (negsucc (S x)) (negsucc (S z)) y =
  ap pred (ℤ-decode (negsucc x) (negsucc z) y)

∼∼-dec : (n m : ℤ) → Dec (n ∼∼ m)
∼∼-dec (pos O) (pos O) = inl unit
∼∼-dec (pos O) (pos (S x)) = inr ⊥-elim
∼∼-dec (pos O) (negsucc x) = inr ⊥-elim
∼∼-dec (pos (S x)) (pos O) = inr ⊥-elim
∼∼-dec (pos (S x)) (pos (S y)) = ∼∼-dec (pos x) (pos y)
∼∼-dec (pos (S x)) (negsucc O) = inr ⊥-elim
∼∼-dec (pos (S x)) (negsucc (S y)) = inr ⊥-elim
∼∼-dec (negsucc O) (pos x) = inr ⊥-elim
∼∼-dec (negsucc O) (negsucc O) = inl unit
∼∼-dec (negsucc O) (negsucc (S x)) = inr ⊥-elim
∼∼-dec (negsucc (S x)) (pos y) = inr ⊥-elim
∼∼-dec (negsucc (S x)) (negsucc O) = inr ⊥-elim
∼∼-dec (negsucc (S x)) (negsucc (S y)) =
  ∼∼-dec (negsucc x) (negsucc y)

ℤ-=-dec : (n m : ℤ) → Dec (n == m)
ℤ-=-dec n m = ⊔-rec (inl ∘ (ℤ-decode n m))
                    (inr ∘ (λ g → (g ∘ ℤ-encode n m)))
                    (∼∼-dec n m)

ℤ-is-set : is-set ℤ
ℤ-is-set = dec-eq-is-set ℤ-=-dec

-- Integer addition
plus : ℤ → ℤ → ℤ
plus (pos O) n = n
plus (pos (S x)) n = succ (plus (pos x) n)
plus (negsucc O) n = pred n
plus (negsucc (S x)) n = pred (plus (negsucc x) n)

-- I use the symbol +˙ for integer addition, since + has already
-- been taken for natural numbers addition
_+˙_ : ℤ → ℤ → ℤ
m +˙ n = plus m n

-- Sends n to -n
min : ℤ → ℤ
min (pos 0) = pos 0
min (pos (S x)) = negsucc x
min (negsucc x) = succ (pos x)

-- (pos 0) is a right unit for integer addition
+˙-unit-r : (m : ℤ) → (m +˙ (pos 0)) == m
+˙-unit-r (pos 0) = idp
+˙-unit-r (pos (S x)) = ap succ (+˙-unit-r (pos x))
+˙-unit-r (negsucc 0) = idp
+˙-unit-r (negsucc (S x)) = ap pred (+˙-unit-r (negsucc x))

-- Taking successors on the right commutes with addition.
+˙-succ-r : (m n : ℤ) → (m +˙ (succ n)) == succ (m +˙ n)
+˙-succ-r (pos 0) n = idp
+˙-succ-r (pos (S x)) n = ap succ (+˙-succ-r (pos x) n)
+˙-succ-r (negsucc 0) n = (pred (succ n)) =⟨ (<--inv-l succ-eq n) ⟩
                          n               =⟨ (! (<--inv-r succ-eq n)) ⟩
                          (succ (pred n)) =∎
+˙-succ-r (negsucc (S x)) n =
  (pred ((negsucc x) +˙ (succ n)))
    =⟨ (ap pred (+˙-succ-r (negsucc x) n)) ⟩

  ((pred (succ ((negsucc x) +˙ n)))
    =⟨ (<--inv-l succ-eq ((negsucc x) +˙ n)) ⟩

  (((negsucc x) +˙ n)
    =⟨ (! (<--inv-r succ-eq ((negsucc x) +˙ n))) ⟩

  ((succ (pred ((negsucc x) +˙ n))) =∎)))

-- Taking predecessors on the right commutes with addition.
+˙-pred-r : (m n : ℤ) → (m +˙ (pred n)) == pred (m +˙ n)
+˙-pred-r (pos 0) n = idp
+˙-pred-r (pos (S x)) n =
  succ ((pos x) +˙ (pred n))
    =⟨ ap succ (+˙-pred-r (pos x) n) ⟩

  (succ (pred ((pos x) +˙ n)))
    =⟨ <--inv-r succ-eq ((pos x) +˙ n) ⟩

  ((pos x) +˙ n)
     =⟨ (! (<--inv-l succ-eq ((pos x) +˙ n))) ⟩

  ((pred (succ ( (pos x) +˙ n)))) =∎
```

```
+'-pred-r (negsucc O) n = idp
+'-pred-r (negsucc (S x)) n = ap pred (+'-pred-r (negsucc x) n)
```

The final file, `FundamentalGroupCircle.agda`, contains the proof itself

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

open import lib.Basics
open import lib.types.Sigma
open import lib.types.Nat
open import Integers
open import Circle

module FundamentalGroupCircle where

{- Universal cover of 𝕊¹ -}
code : 𝕊¹ → Type₀
code = circle-rec ℤ (ua succ-eq)

loop-to-succ : (x : ℤ) → x == succ x [ code ↓ loop ]
loop-to-succ x = ↓-ap-out (λ y → y) code loop
                  (↓-idf-in (ap code loop) t) where
  t : coe (ap code loop) x == succ x
  t = (coe (ap code loop) x)       =⟨ ap2 coe circle-rec-comp idp ⟩
      (coe (ua succ-eq) x)         =⟨ coe-β succ-eq x ⟩
      succ x                       =∎

loop-to-pred : (x : ℤ) → x == pred x [ code ↓ (! loop) ]
loop-to-pred x = ↓-ap-out (λ y → y) code (! loop)
                 (↓-idf-in (ap code (! loop)) t) where
  t : coe (ap code (! loop)) x == pred x
  t = (coe (ap code (! loop)) x)      =⟨ ap2 coe (ap-! code loop) idp ⟩
      (coe (! (ap code loop)) x)      =⟨ coe-! (ap code loop) x ⟩
      (coe! (ap code loop) x)         =⟨ ap2 coe! circle-rec-comp idp ⟩
      (coe! (ua succ-eq) x)           =⟨ coe!-β succ-eq x ⟩
      pred x                          =∎

transp-loop-to-pred : transport code (! loop) == pred
transp-loop-to-pred = λ= (λ x → to-transp (loop-to-pred x))

transport-path-fib : ∀ {i} {A : Type i} {a x₁ x₂ : A} (p : x₁ == x₂) →
                     transport (λ u → a == u) p == (λ q → q · p)
transport-path-fib idp = λ= (λ q → ! (·-unit-r q))

```

```
encode : (x : 𝕊¹) → (base == x) → (code x)
encode x p = transport code p (pos 0)

decode : (x : 𝕊¹) → (code x) → (base == x)
decode x = circle-ind P (p-conc loop) loop-coherence x where
  P : 𝕊¹ → Type₀
  P y = (code y) → (base == y)

  lemma1 : ∀ {i j k} {X : Type i} (A : X → Type j) (B : X → Type k)
           {x y : X} (p : x == y) (f : A x → B x) →
           transport (λ u → (A u → B u)) p f ==
           (λ y → transport B p (f (transport A (! p) y)))
  lemma1 A B idp f = idp

  lemma2 : (x : ℤ) → p-conc loop (pred x) · loop == p-conc loop x
  lemma2 (pos O) = !-inv-l loop
  lemma2 (pos (S n)) = idp
  lemma2 (negsucc O) =
    ((! loop · ! loop) · loop)
      =⟨ (·-assoc (! loop) (! loop) loop) ⟩

    ((! loop · (! loop · loop))
      =⟨ ((! loop) ·□ (!-inv-l loop)) ⟩

    ((! loop · idp)
      =⟨ (·-unit-r (! loop)) ⟩

    (! loop =■)))
  lemma2 (negsucc (S n)) =
    (((power-concat base loop (negsucc n) · ! loop) · ! loop) · loop)
      =⟨ (·-assoc (power-concat base loop (negsucc n) · ! loop)
          (! loop) loop) ⟩

    (((power-concat base loop (negsucc n) · ! loop) · ! loop · loop)
      =⟨ ((power-concat base loop (negsucc n) · ! loop) ·□ (!-inv-l loop)) ⟩

    (((power-concat base loop (negsucc n) · ! loop) · idp)
      =⟨ (·-unit-r (power-concat base loop (negsucc n) · ! loop)) ⟩

    (power-concat base loop (negsucc n) · ! loop) =■))

  ∘-right-trans : ∀ {i j k} {A : Type i} {B : Type j} {C : Type k}
                  (f : A → B) (g : A → B) (h : B → C) (p : f == g)
                  → (h ∘ f == h ∘ g)
  ∘-right-trans f g h idp = idp

  ∘-left-trans : ∀ {i j k} {A : Type i} {B : Type j} {C : Type k}
                 (f : A → B) (g : B → C) (h : B → C) (p : g == h)
                 → (g ∘ f == h ∘ f)
  ∘-left-trans f g h idp = idp

  transp-path : transport P loop (p-conc loop) == (p-conc loop)
  transp-path =
    (transport P loop (p-conc loop))
      =⟨ (lemma1 code (λ u → base == u) loop (p-conc loop)) ⟩

    ((transport (λ u → base == u) loop ∘ ((p-conc loop) ∘
                  transport code (! loop)))
      =⟨ (∘-left-trans ((p-conc loop) ∘ transport code (! loop))
              (transport (λ u → base == u) loop)
              (λ p → p · loop) (transport-path-fib loop)) ⟩

    ((λ p → p · loop) ∘ ((p-conc loop) ∘ transport code (! loop)))
      =⟨ ∘-right-trans (transport code (! loop)) pred
              ((λ p → p · loop) ∘ (p-conc loop))
              transp-loop-to-pred ⟩

    ((λ p → p · loop) ∘ (p-conc loop) ∘ pred)
      =⟨ idp ⟩

    ((λ n → (p-conc loop (pred n) · loop))
      =⟨ (λ= lemma2) ⟩

    (p-conc loop =■)))

  loop-coherence : (p-conc loop) == (p-conc loop) [ P ↓ loop ]
  loop-coherence = from-transp P loop transp-path

decode-encode : (x : 𝕊¹) (p : base == x)
                      → decode x (encode x p) == p
decode-encode .base idp = idp

encode-decode : (x : 𝕊¹) (c : code x)
                      → encode x (decode x c) == c
encode-decode =
  circle-ind (λ x → (c : code x) → encode x (decode x c) == c)
    base-case path-coherence2 where

```

```
base-case : (m : code base) → encode base (decode base m) == m
base-case (pos O) = idp
base-case (pos (S x)) =
  (encode base (p-conc loop (pos (S x))))
    =⟨ idp ⟩

  ((encode base ((p-conc loop (pos x)) · loop))
    =⟨ idp ⟩

  (transport code ((p-conc loop (pos x)) · loop) (pos 0)
    =⟨ transp-· ((p-conc loop (pos x))) loop (pos 0) ⟩

  (transport code loop (transport code (p-conc loop (pos x))
                          (pos 0)))
    =⟨ to-transp (loop-to-succ
             ((transport code (p-conc loop (pos x)) (pos 0)))) ⟩

  (succ (transport code (p-conc loop (pos x)) (pos 0)))
    =⟨ (ap succ (base-case (pos x))) ⟩

  (succ (pos x) =■)))

base-case (negsucc O) =
  (transport code (! loop) (pos 0))
    =⟨ (to-transp (loop-to-pred (pos 0))) ⟩

  (negsucc 0 =■)

base-case (negsucc (S x)) =
  (transport code ((p-conc loop (negsucc x)) · (! loop))
                                      (pos 0))
    =⟨ transp-· (p-conc loop (negsucc x)) (! loop) (pos 0) ⟩

  (transport code (! loop) (transport code (p-conc loop (negsucc x))
                          (pos 0)))
    =⟨ to-transp (loop-to-pred
              (transport code (p-conc loop (negsucc x)) (pos 0))) ⟩

  (pred (transport code (p-conc loop (negsucc x)) (pos 0)))
     =⟨ ap pred (base-case (negsucc x)) ⟩

  pred (negsucc x) =■

P : 𝕊¹ → Type₀
```

```
  P x = (c : code x) → encode x (decode x c) == c

  path-coherence2 : base-case == base-case [ P ↓ loop ]
  path-coherence2 = from-transp P loop (λ= (λ x → set-path ℤ-is-set
                 (transport P loop base-case x) (base-case x)))

𝕊¹-fiber-equiv : (x : 𝕊¹) → (base == x) ≃ code x
𝕊¹-fiber-equiv x = equiv (encode x) (decode x) (encode-decode x)
                         (λ a → decode-encode x a)

-- Equivalence between ℤ and the loopspace of 𝕊¹ at base.
loopsp-eq : ℤ ≃ (base == base)
loopsp-eq = (<- (𝕊¹-fiber-equiv base)) ,
                (is-equiv-inverse (snd (𝕊¹-fiber-equiv base)))

-- The map from ℤ to (base == base) takes addition to concatenation.
loopsp-eq-mult : (m n : ℤ) → (-> loopsp-eq) (m +˙ n) ==
                              (-> loopsp-eq m) · (-> loopsp-eq n)
loopsp-eq-mult m (pos O) =
  (p-conc loop (m +˙ (pos 0)))
    =⟨ (ap (p-conc loop) (+˙-unit-r m)) ⟩

  ((p-conc loop m)
    =⟨ (! (·-unit-r (p-conc loop m))) ⟩

  (((p-conc loop m) · idp) =∎))

loopsp-eq-mult m (pos (S x)) =
  (p-conc loop (m +˙ (succ (pos x))))
    =⟨ ap (p-conc loop) (+˙-succ-r m (pos x)) ⟩

  (p-conc loop (succ (m +˙ (pos x))))
    =⟨ (p-conc-succ loop (m +˙ (pos x))) ⟩

  (((p-conc loop (m +˙ (pos x))) · loop)
    =⟨ ((loopsp-eq-mult m (pos x)) ·ᵣ loop) ⟩

  ((((p-conc loop m) · (p-conc loop (pos x))) · loop)
   =⟨ (·-assoc (p-conc loop m) (p-conc loop (pos x)) loop) ⟩

  (((p-conc loop m) · ((p-conc loop (pos x)) · loop))
    =⟨ (p-conc loop m) ·□ (! (p-conc-succ loop (pos x))) ⟩

  ((p-conc loop m) · (p-conc loop (succ (pos x)))) =∎)))
```

```

loopsp-eq-mult m (negsucc 0) =
  (p-conc loop (m +˙ (negsucc 0)))
    =⟨ (ap (p-conc loop) (+˙-pred-r m (pos 0))) ⟩

  ((p-conc loop (pred (m +˙ (pos 0))))
    =⟨ (p-conc-pred loop (m +˙ (pos 0))) ⟩

  (((p-conc loop (m +˙ (pos 0))) · (! loop))
    =⟨ ((ap (p-conc loop) (+˙-unit-r m)) ·ᵣ ! loop) ⟩

  ((p-conc loop m) · (! loop)) =∎))

loopsp-eq-mult m (negsucc (S x)) =
  (p-conc loop (m +˙ pred (negsucc x)))
    =⟨ (ap (p-conc loop) (+˙-pred-r m (negsucc x))) ⟩

  ((p-conc loop (pred (m +˙ (negsucc x))))
    =⟨ (p-conc-pred loop (m +˙ (negsucc x))) ⟩

  (((p-conc loop (m +˙ (negsucc x))) · (! loop))
    =⟨ ((loopsp-eq-mult m (negsucc x)) ·ᵣ (! loop)) ⟩

  ((((p-conc loop m) · (p-conc loop (negsucc x))) · (! loop))
    =⟨ (·-assoc (p-conc loop m) (p-conc loop (negsucc x)) (! loop)) ⟩

  (((p-conc loop m) · ((p-conc loop (negsucc x)) · (! loop)))
    =⟨ ((p-conc loop m) ·□ (! (p-conc-pred loop (negsucc x)))) ⟩

  (((p-conc loop m) · (p-conc loop (pred (negsucc x)))) =∎)))))
```

B. Code for the Freudenthal Suspension Theorem

This appendix contains the Agda code of my formalization of the proof of the Freudenthal Suspension theorem. The proof is spread out over two files: `WedgeConnectivity.agda` and `FreudenthalSuspension.agda`.

The first file, `WedgeConnectivity.agda`, contains a proof of theorem 4.5, as well as a proof of the wedge connectivity lemma (lemma 4.8).

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

open import lib.Basics
open import lib.NConnected
open import lib.NType2
open import lib.Equivalence2
open import lib.types.Sigma
open import lib.types.TLevel
open import lib.types.Nat

{- This module contains the proof of the wedge connectivity lemma.-}
module WedgeConnectivity where

-- A map has n-truncated fibers if all its fibers are n-types.
module _ {i j} {A : Type i} {B : Type j} (f : A → B) where
  has-trunc-fibers : ℕ₋₂ → Type (lmax i j)
  has-trunc-fibers n = Π B λ b → has-level n (hfiber f b)

  contr-fib-to-eq : has-trunc-fibers ⟨ 0 ⟩₋₂ → is-equiv f
  contr-fib-to-eq w = contr-map-is-equiv w

  eq-to-contr-fib : is-equiv f → has-trunc-fibers ⟨ 0 ⟩₋₂
  eq-to-contr-fib e = equiv-is-contr-map e

-- Some equivalences of path types
module _ {i} {A : Type i} (x y : A) where
  symm-path-is-equiv : (x == y) ≃ (y == x)
  symm-path-is-equiv =
    (λ p → ! p) , is-eq (λ p → ! p) (λ p → ! p) !-! !-!
```

```

-- Pre- and post-concatenation with paths give an equivalence of
-- path types. Weirdly enough, I couldn't find this in the library.
module _ {i} {A : Type i} {x y z : A} where
  ·-r : (y == z) → (x == y) → (x == z)
  ·-r q = λ p → p · q

  ·-r-is-equiv : (q : y == z) → (is-equiv (·-r q))
  ·-r-is-equiv q = is-eq (·-r q) (λ p → p · (! q))
    (λ b → (·-assoc b (! q) q) · (b ·□ (!-inv-l q)) · (·-unit-r b))
    λ a → (·-assoc a q (! q))  · ((a ·□ (!-inv-r q)) · (·-unit-r a))

  ·-r-equiv : (q : y == z) → (x == y) ≃ (x == z)
  ·-r-equiv q = (·-r q) , (·-r-is-equiv q)

  ·-l : (x == y) → (y == z) → (x == z)
  ·-l q = λ p → q · p

  ·-l-is-equiv : (q : x == y) → (is-equiv (·-l q))
  ·-l-is-equiv q = is-eq (·-l q) (λ p → (! q) · p)
    (λ b → (! (·-assoc q (! q) b)) · ((!-inv-r q) ·ᵣ b))
    λ a → (! (·-assoc (! q) q a)) · (((!-inv-l q) ·ᵣ a))

  ·-l-equiv : (q : x == y) → (y == z) ≃ (x == z)
  ·-l-equiv q = (·-l q) , (·-l-is-equiv q)


-- Adapted from the library (equivalence.agda)
equiv-pres-lvl : ∀ {i j} {A : Type i} {B : Type j} {n : ℕ₋₂} (e : A ≃ B)
                 (w : has-level n A) → has-level n B
equiv-pres-lvl {n = ⟨-2⟩} e w =
  has-level-in (-> e (contr-center w) ,
               (λ y → ap (-> e) (contr-path w _) · <--inv-r e y))
equiv-pres-lvl {n = S n} e w = has-level-in (λ x y →
  equiv-preserves-level (ap-equiv (e ⁻¹) x y ⁻¹)
               {{has-level-apply w (<- e x) (<- e y)}})

-- This module contains a proof of lemma 8.6.1 in the HoTT book.
module _ {i j} {A : Type i} {B : Type j} (n : ℕ₋₂)
                       (f : A → B) (c : has-conn-fibers n f) where

  pullback : ∀ {ℓ} (k : ℕ₋₂) (P : B → k -Type ℓ)
             → ∏ B (fst ∘ P) → ∏ A (fst ∘ P ∘ f)
  pullback k P s = s ∘ f
```

```

pathover-pb-equiv : ∀ {ℓ} {k : ℕ₋₂} (P : B → k -Type ℓ)
                  → {s : ∏ A (fst ∘ P ∘ f)}
                  → (x y : hfiber (pullback k P) s)
                  → (r : (fst x) == (fst y))
                  → ((snd x) == (snd y) [ (λ φ → (φ ∘ f == s)) ↓ r ]) ≃
                      (((app= r) ∘ f) == (app= ((snd x) · (! (snd y)))))
pathover-pb-equiv P (g , p) (g , q) idp =
  (p == q)
    ≃⟨ (symm-path-is-equiv p q) ⟩
  (q == p)
    ≃⟨ (ap-equiv (·-r-equiv (! q)) q p) ⟩
  (q · (! q)) == p · (! q)
    ≃⟨ ·-l-equiv (! (!-inv-r q)) ⟩
  (idp == p · (! q))
    ≃⟨ ap-equiv (app= , StrongFunextDep.app=-is-equiv) idp (p · (! q)) ⟩
  ((app= idp) == (app= (p · (! q)))) ≃■

-- construct a family of k-types Q out of a family of (k + 1)-types P given
-- sections g and h into P by setting Q b = (g b == h b)
lower-lvl : ∀ {ℓ} {k : ℕ₋₂} (P : B → (S k) -Type ℓ) (g h : ∏ B (fst ∘ P))
            → (B → k -Type ℓ)
lower-lvl P g h = λ b → (g b == h b) ,
                         (has-level.has-level-apply (snd (P b))) (g b) (h b)

pullback-theorem : ∀ {ℓ} (d : ℕ) → (P : B → (⟨ d ⟩₋₂ +2+ n)  -Type ℓ)
                  → has-trunc-fibers (pullback (⟨ d ⟩₋₂ +2+ n) P) ⟨ d ⟩₋₂

-- In this case, n = k, so the lemma reduces to the "induction principle"
-- for n-connected maps.
pullback-theorem 0 P =
  eq-to-contr-fib (pullback n P) (ConnExtend.restr-is-equiv c P)

-- For the induction step, n stays fixed, so (add (S d) n) = k + 1.
-- We have to show that the fiber over s has level k - n - 2 (= ⟨ d ⟩₋₂),
-- where s is a section in  ∏ A (fst ∘ P ∘ f).
pullback-theorem {ℓ} (S d) P s =
  has-level-in λ x y → equiv-pres-lvl ((fib-eq x y) ⁻¹) (eq-fiber-lvl x y)
    where
    -- This is a fiber of the lowered version of the pullback map of P.
    lower-lvl-fib : (x y : hfiber (pullback (S (⟨ d ⟩₋₂ +2+ n)) P) s)
              → Type (lmax i (lmax j ℓ))
    lower-lvl-fib (g , p) (h , q) =
      (hfiber (pullback (⟨ d ⟩₋₂ +2+ n) (lower-lvl P g h)) (app= (p · (! q))))
```

```

        -- By induction, we have that the fiber of the lowered family over
        -- x==y has level ⟨ d ⟩₋₂.
        eq-fiber-lvl : (x y : hfiber (pullback (S (⟨ d ⟩₋₂ +2+ n)) P) s)
                     → (has-level ⟨ d ⟩₋₂ (lower-lvl-fib x y))
        eq-fiber-lvl x y = pullback-theorem d ((lower-lvl P (fst x) (fst y)))
                             (app= (((snd x) · (! (snd y)))))

        -- We can then prove that if x and y are in the fiber over s
        -- of the pullback, then x == y is equivalent to a fiber of
        -- the lowered pullback map.
        fib-eq : (x y : hfiber (pullback (S (⟨ d ⟩₋₂ +2+ n)) P) s)
                → (x == y) ≃ lower-lvl-fib x y
        fib-eq (g , p) (h , q) =
            ((g , p) == (h , q))
              -- Characterization of equality in Σ-types
              ≃⟨ ((=Σ-econv (g , p) (h , q)) ⁻¹) ⟩

            ((Σ (g == h) (λ r → p == q [ (λ φ → (φ ∘ f == s)) ↓ r ]))
              -- Applying pathover-pb-equiv to the the
              -- type family of the Σ-type
              ≃⟨ (Σ-fmap-r (λ r →
                 -> (pathover-pb-equiv P (g , p) (h , q) r))) ,
                 fiber-equiv-is-total-equiv
                 (λ r → -> (pathover-pb-equiv P (g , p) (h , q) r))
                 (λ r → snd (pathover-pb-equiv P (g , p) (h , q) r)) ⟩
            (Σ (g == h) (λ r → ((app= r) ∘ f) == (app= (p · (! q)))))
              -- Pulling back the Σ type along the equivalence app=
              ≃⟨ Σ-emap-l (λ r → (r ∘ f) == (app= (p · (! q)))) app=-equiv ⟩
              (lower-lvl-fib (g , p) (h , q)) ≃∎)

-- Double ∏-type
module _ {i j k} (A : Type i) (B : Type j) (P : A → B → Type k) where

  ∏∏ : Type (lmax i (lmax j k))
  ∏∏ = ∏ A (λ a → ∏ B (λ b → P a b))

-- If m == n and A is an m-Type, then A is an n-Type
module _ {i} {m n : ℕ₋₂} (A : m -Type i) where

  level-eq' : {B : Type i} (c : has-level m B)
              (p : m == n) → (has-level n B)
  level-eq' c idp = c

  level-eq : (p : m == n) → n -Type i
  level-eq p = (fst A) , level-eq’ (snd A) p

-- Maps from ⊤ correspond to elements (⊤ represents the forgetful functor).
module _ {i} {P : ⊤ → Type i} where

  ⊤-rep : ∏ ⊤ P → P unit
  ⊤-rep s = s unit

  ⊤-rep-is-equiv : is-equiv ⊤-rep
  ⊤-rep-is-equiv =
    is-eq ⊤-rep (λ x → cst x) (λ _ → idp) λ s → λ= λ x → idp

  ⊤-rep-equiv : ∏ ⊤ P ≃ P unit
  ⊤-rep-equiv = ⊤-rep , ⊤-rep-is-equiv

--This module contains a proof of the
--wedge connectivity lemma (8.6.2 in the book)
module _ {i j ℓ} (A : Ptd i) (B : Ptd j) (n m : ℕ)
               (v : is-connected ⟨ n ⟩  (de⊙ A))
               (w : is-connected ⟨ m ⟩ (de⊙ B))
               (P : (de⊙ A) → (de⊙ B) → ⟨ n + m ⟩ -Type ℓ) where

  wedge-conn : (f : ∏ (de⊙ A) λ a → (fst (P a (pt B))))
               → (g : ∏ (de⊙ B) λ b → (fst (P (pt A) b)))
               → (p : g (pt B) == f (pt A))
               → Σ (∏∏ (de⊙ A) (de⊙ B) (λ a b → fst (P a b)))
                   (λ h → (((λ a → h a (pt B)) ∼ f)) ×
                          ((λ b → h (pt A) b) ∼ g))

  wedge-conn f g p = (λ a b → fst (s a) b) ,
                     ((λ a → snd (s a)) , s-comp) where

    Q : de⊙ A → Type (lmax j ℓ)
    Q a = Σ (∏ (de⊙ B) (λ b → fst (P a b))) λ k → k (pt B) == f a

    -- This comes down to the equivalence ⊤-rep
    Q-is-fib : (a : de⊙ A) → hfiber (pullback ⟨ m ⟩₋₁ (cst (pt B))
      (pointed-conn-out (de⊙ B) (pt B) {{ w }}) ⟨ n + m ⟩ (P a))
      (cst (f a)) ≃ (Q a)
    Q-is-fib a = (Σ-fmap-r (λ r → ap ⊤-rep)) ,
                 fiber-equiv-is-total-equiv (λ r → ap ⊤-rep)
                 λ r → ap-is-equiv ⊤-rep-is-equiv
                 (cst (r (pt B))) (cst (f a))

    lemma : ⟨ n + m ⟩ == ⟨ S n ⟩₋₂ +2+ ⟨ m ⟩₋₁
    lemma =
    (S (S ⟨ n + m ⟩₋₂))
      =⟨ (ap (λ x → S (S x)) (+-+2+ n m)) ⟩

    ((S (S (⟨ n ⟩₋₂ +2+ ⟨ m ⟩₋₂)))
      =⟨ ap S (! (+2+-βr ⟨ n ⟩₋₂ ⟨ m ⟩₋₂)) ⟩

    (⟨ S n ⟩₋₂ +2+ ⟨ m ⟩₋₁) =∎)

    -- Q is a family of (n-1)-types
    Q-has-lvl : (a : (de⊙ A)) → has-level ⟨ n ⟩₋₁ (Q a)
    Q-has-lvl a = equiv-preserves-level (Q-is-fib a)
                  {{ pullback-theorem ⟨ m ⟩₋₁ ((cst (pt B)))
                  ((pointed-conn-out (de⊙ B) (pt B) {{ w }})) (S n)
                  (λ b → level-eq (P a b) lemma) (cst (f a)) }}

    Q-lvl : (de⊙ A) → ⟨ n ⟩₋₁ -Type (lmax j ℓ)
    Q-lvl a = (Q a) , (Q-has-lvl a)

    s : (a : de⊙ A) → Q a
    s = conn-extend (pointed-conn-out (de⊙ A) (pt A) {{ v }})
          Q-lvl λ x → g , p

    s-comp' : s (pt A) == g , p
    s-comp' = conn-extend-β (pointed-conn-out (de⊙ A) (pt A) {{ v }})
                            Q-lvl (λ x → g , p) unit

    s-comp : (λ b → fst (s (pt A)) b) ∼ g
    s-comp = λ b → app= (ap fst s-comp') b
```

The second file, `FreudenthalSuspension.agda` contains the proof of the theorem together with many lemma's that are used in the proof.

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

open import WedgeConnectivity

open import lib.Basics
open import lib.NConnected
open import lib.NType2
open import lib.Equivalence2
open import lib.types.Suspension.Core
open import lib.types.Truncation
open import lib.types.Sigma
open import lib.types.Pi
open import lib.types.TLevel
open import lib.types.Nat
open import lib.types.Paths
open import lib.types.Pushout


module FreudenthalSuspension where

module _ {i} {A : Type i} {n : ℕ} (c : is-connected ⟨ n ⟩ A) where

  -- A is n-connected for n ≥ 0, so A is certainly 0-connected.
  n-to-0-conn : is-connected ⟨ 0 ⟩ A
  n-to-0-conn = connected-≤T (≤T-ap-S (≤T-ap-S (-2≤T ⟨ n ⟩₋₂))) {{ c }}

module _ {i j} {A : Type i} {n : ℕ₋₂} (P : Trunc n A → (Type j))
         (w : (x : Trunc n A) → has-level n (P x))
         (d : (a : A) → P [ a ]) where

  Trunc-elim-aux : ∏ (Trunc n A) P
  Trunc-elim-aux = Trunc-elim {A = A} {P = P} {{w}} d

module _ {i} {A : Type i} {n : ℕ} where

  Trunc-=-level : (x y : Trunc ⟨ n ⟩ A) → has-level ⟨ n ⟩ (x == y)
  Trunc-=-level x y = raise-level ⟨ n ⟩₋₁ (has-level-apply Trunc-level x y)

module _ {i j} {A : Type i} {B : Type j} {n : ℕ} (c : has-level ⟨ n ⟩ B)
         {f g : A → B} where

  trunc-pres-∼ : (f ∼ g) → (Trunc-rec {{c}} f) ∼ (Trunc-rec {{c}} g)
  trunc-pres-∼ w =
    Trunc-elim-aux (λ x → (Trunc-rec {{c}} f x) == (Trunc-rec {{c}} g x))
                   (λ x → raise-level ⟨ n ⟩₋₁ (has-level-apply c
                   (Trunc-rec {{c}} f x) (Trunc-rec {{c}} g x))) w

module _ {i j} {A : Type i} {B : Type j} {f g : A → B} where

  =-pres-conn : {n : ℕ₋₂} (p : f == g)
    (c : has-conn-fibers n f) → (has-conn-fibers n g)
  =-pres-conn idp c = c

```

```
module _ {i j} {A : Type i} {B : Type j} {f g : A → B} where
  ∼-! : (f ∼ g) → (g ∼ f)
  ∼-! h = ! ∘ h

module _ {i j} {A : Type i} {n : A} where

  fun-fib-to-conc-= : {a₁ a₂ : A} (m : a₁ == a₂)
    (C : (n == a₁) → Type j)
    → (transport (λ a → ((n == a) → Type j)) m C)
    == (λ b → C (b ·' (! m)))
  fun-fib-to-conc-= idp C = idp

  fun-fib-to-conc-∼ : {a₁ a₂ : A} (m : a₁ == a₂) (b : n == a₂)
    (C : (n == a₁) → Type j)
    → (transport (λ a → ((n == a) → Type j)) m C) b
    == C (b ·' (! m))
  fun-fib-to-conc-∼ m b C = app= (fun-fib-to-conc-= m C) b

module _ {i j} {A : Type i} {B : A → Type j} {x y : A}
         {v : B x} {w : B y} {p : x == y}
         {q u : v == w [ B ↓ p ]} where

  Σ-path-id : (α : q == u) → (pair= p q) == (pair= p u)
  Σ-path-id idp = idp

module _ {i j} {A : Type i} {B : A → Type j}
               {x y : A} {p q : x == y} where

  id-path-transp : (r : p == q) (c : B x)
                 → (transport B p c) == (transport B q c)
  id-path-transp idp c = idp

-- WedgeConn paths module
module _ {i} {A : Type i} where

  cancel-·'-r : {a₁ a₂ a₃ : A} (p q : a₁ == a₂) (w : a₂ == a₃)
                (r : p ·' w == q ·' w) → (p == q)
  cancel-·'-r p q idp r = r

  cancel-·'-r-idp : {a₁ a₂ a₃ : A} (p : a₁ == a₂) (w : a₂ == a₃)
                    → (cancel-·'-r p p w idp) == idp
  cancel-·'-r-idp p idp = idp

  uncancel-·'-r : {a₁ a₂ a₃ : A} (p q : a₁ == a₂) (w : a₂ == a₃) →
```

```
                 (p == q) → p ·' w == q ·' w
  uncancel-·'-r p q idp r = r

  unc-c : {a₁ a₂ a₃ : A} (p q : a₁ == a₂) (w : a₂ == a₃)
    (r : p ·' w == q ·' w)
    → (uncancel-·'-r p q w (cancel-·'-r p q w r)) == r
  unc-c p q idp r = idp

  c-unc : {a₁ a₂ a₃ : A} (p q : a₁ == a₂) (w : a₂ == a₃)
    (r : p == q) → (cancel-·'-r p q w (uncancel-·'-r p q w r)) == r
  c-unc p q idp r = idp

  shuffle-·'-r : {a₁ a₂ : A} (p q : a₁ == a₂)
               (r : idp == q ·' (! p)) → (p == q)
  shuffle-·'-r idp q r = r

  shuffle-·'-inv : {a₁ a₂ a₃ : A} (p : a₁ == a₂) (q w : a₁ == a₃)
                 (r : p ·' (! p) == q ·' (! w)) → (w == q)
  shuffle-·'-inv idp q w r = shuffle-·'-r w q r

  undo-·'-inv : {a₁ a₂ : A} (p : a₁ == a₁) (q : a₁ == a₂) (r : p == idp)
              → p == q ·' (! q)
  undo-·'-inv p idp r = r

  inv-·'-eq : {a₁ a₂ a₃ : A} (p : a₁ == a₂) (q : a₁ == a₃)
              → (p ·' (! p)) == (q ·' (! q))
  inv-·'-eq idp idp = idp

  undo-·'-eq : {a₁ a₂ a₃ : A} (p : a₁ == a₂) (q : a₁ == a₃) →
              (shuffle-·'-inv p q q (inv-·'-eq p q)) == idp
  undo-·'-eq idp idp = idp

  undo-·'-triv : {a₁ a₂ a₃ : A} (p : a₁ == a₂) (q : a₁ == a₃)
                 → undo-·'-inv (p ·' (! p)) q (!-inv'-r p) == inv-·'-eq p q
  undo-·'-triv idp idp = idp

  shuffle=cancel : {a₁ a₂ : A} (p q : a₁ == a₂) →
                 (cancel-·'-r p q (! p)) == (shuffle-·'-inv p q p)
  shuffle=cancel idp q = λ= (λ r → idp)

module _ {i} {A : Type i} where

  path-fib-↓-conc' : {a b c : A} (p : a == b) (q : b == c)
                   → p == p ·' q [ (λ v → a == v) ↓ q ]
  path-fib-↓-conc' p idp = idp

  path-fib-↓-conc'-inv : {a b c : A} (p : a == b) (q : c == b)
                       → p ·' (! q) == p [ (λ v → a == v) ↓ q ]
  path-fib-↓-conc'-inv p idp = idp

  invert-path-fib : {a b : A} (p q : a == b)
        → (pair= (! q) (path-fib-↓-conc' p (! q))) ==
          (! (pair= q (path-fib-↓-conc'-inv p q)))
  invert-path-fib p idp = idp

module _ {i j} {A : Type i} (B : A → Type j) where

  transp!-shuffle : {a₁ a₂ : A} {b₁ : B a₁} {b₂ : B a₂} (p : a₁ == a₂) →
              ((transport B p b₁) == b₂) → ((transport B (! p) b₂) == b₁)
  transp!-shuffle idp r = ! r

module _ {i} {A : Type i} where

  tid : {x y : A} (p : x == y) → idp == p [ (λ v → x == v) ↓ p ]
  tid idp = idp

  tid-· : {x y z : A} (p : x == y) (q : y == z) → (tid (p ·'  q)) ==
                (tid p) ·'ᵈ (path-fib-↓-conc' p q)
  tid-· idp idp = idp

module _ where
  +-lemma : (n : ℕ) → n *2 == n + n
  +-lemma O = idp
  +-lemma (S n) = ap S ((ap S (+-lemma n)) · (! (+-βr n n)))

module _ {i} (X : Ptd i) (n : ℕ) (c : is-connected ⟨ n ⟩ (de⊙ X)) where

  -- Version of σloop using ·' instead of ·
  σloop' : de⊙ X → north' (de⊙ X) == north' (de⊙ X)
  σloop' x = merid x ·' ! (merid (pt X))

  -- These two versions of σloop are equal, so it suffices to prove the
  -- Freudenthal suspension theorem for σloop'.
  σloop-eq : (σloop X) == σloop' --(σloop X) == σloop'
  σloop-eq = λ= (λ x → ·=·' (merid x) (! (merid (pt X))))

  code-N : (p : north == north) → Type i
  code-N p = Trunc ⟨ n *2 ⟩ (hfiber σloop' p)
```

```
code-S : (q : (north' (de⊙ X)) == south) → Type i
code-S q = Trunc ⟨ n *2 ⟩ (hfiber merid q)

code-transp-reduce :  {y : Susp (de⊙ X)} (q : north == y)
  (C-N : (north == north) → Type i) →
  (transport (λ z → north == z → Type i) q C-N) ==
  (λ p → C-N (p ·' (! q)))
code-transp-reduce q C-N = fun-fib-to-conc-= q C-N

paths-↓-reduce : {y : Susp (de⊙ X)} (q : north == y)
  (C-N : (north == north) → Type i) (C-y : (north == y) → Type i) →
  (C-N == C-y [ (λ z → north == z → Type i) ↓ q ]) ≃
  (∏ (north == y) (λ p → ((C-N (p ·' (! q))) ≃ (C-y p) )))
paths-↓-reduce {y} q C-N C-y =
  C-N == C-y [ (λ z → north == z → Type i) ↓ q ]
    ≃⟨ to-transp-equiv (λ z → north == z → Type i) q ⟩
  transport (λ z → north == z → Type i) q C-N == C-y
    ≃⟨ pre·'-equiv (! (code-transp-reduce q C-N)) ⟩
  (λ p → C-N (p ·' (! q))) == C-y
    ≃⟨ app=-equiv ⟩
  (∏ (north == y) λ p → C-N (p ·' (! q)) == C-y p)
    ≃⟨ ∏-emap-r (λ _ → ua-equiv ⁻¹) ⟩
  ∏ (north == y) (λ p → C-N (p ·' (! q)) ≃ C-y p) ≃■

fun-fam-aux : (q : north == south) → (de⊙ X) → (de⊙ X)
               → ((⟨ n *2 ⟩ -Type i)
fun-fam-aux q x1 x2 =
  (((merid x2) ·' (! (merid (pt X))) == q ·' (! (merid x1))) → (code-S q)) ,
  (∏-level (λ _ → Trunc-level))

fun-fam : (q : north == south) → (de⊙ X) → (de⊙ X) → (⟨ n + n ⟩ -Type i)
fun-fam q x1 x2 = level-eq (fun-fam-aux q x1 x2) (ap ⟨_⟩ (+-lemma n))

wedge-1 : (q : north == south) (x2 : de⊙ X)
          → ((merid x2) ·' (! (merid (pt X))) == q ·' (! (merid (pt X))))
          → (code-S q)
wedge-1 q x2 r =
  [ (x2 , cancel-·'-r (merid x2) q (! (merid (pt X))) r) ]

wedge-1-inv : (q : north == south) (x2 : de⊙ X) →
  (merid x2) == q → (code-N (q ·' (! (merid (pt X)))))
wedge-1-inv q x2 r =
  [ (x2 , uncancel-·'-r (merid x2) q (! (merid (pt X))) r) ]
```

```

wedge-1-tr : (q : north == south)
  → (code-N (q ·' (! (merid (pt X))))) → (code-S q)
wedge-1-tr q = Trunc-rec (λ t → wedge-1 q (fst t) (snd t))

wedge-1-tr-inv : (q : north == south)
  → (code-S q) → (code-N (q ·' (! (merid (pt X)))))
wedge-1-tr-inv q = Trunc-rec λ u → wedge-1-inv q (fst u) (snd u)

wedge-1-do-undo : (q : north == south)
  (t : hfiber merid q) →
  (wedge-1-tr q (wedge-1-tr-inv q [ t ])) == [ t ]
wedge-1-do-undo q (x₂ , r) =
  ap [_] (pair= idp (c-unc (merid x₂) q (! (merid (pt X))) r))

wedge-1-undo-do : (q : north == south)
  (t : hfiber σloop' (q ·' (! (merid (pt X))))) →
  (wedge-1-tr-inv q (wedge-1-tr q [ t ])) == [ t ]
wedge-1-undo-do q (x₂ , r) =
  ap [_] (pair= idp (unc-c (merid x₂) q (! (merid (pt X))) r))

wedge-1-tr-do-undo : (q : north == south)
  (t : code-S q) → (wedge-1-tr q (wedge-1-tr-inv q t)) == t
wedge-1-tr-do-undo q =
  Trunc-elim-aux (λ t → (wedge-1-tr q (wedge-1-tr-inv q t)) == t)
                 (λ t → Trunc-=-level (wedge-1-tr q (wedge-1-tr-inv q t)) t)
                 (wedge-1-do-undo q)

wedge-1-tr-undo-do : (q : north == south)
  (t : code-N (q ·' (! (merid (pt X)))))
  → (wedge-1-tr-inv q (wedge-1-tr q t)) == t
wedge-1-tr-undo-do q =
  Trunc-elim-aux (λ t → (wedge-1-tr-inv q (wedge-1-tr q t)) == t)
                 (λ t → Trunc-=-level (wedge-1-tr-inv q (wedge-1-tr q t)) t)
                 (wedge-1-undo-do q)

wedge-1-tr-is-equiv : (q : north == south) → is-equiv (wedge-1-tr q)
wedge-1-tr-is-equiv q = is-eq (wedge-1-tr q) (wedge-1-tr-inv q)
                              (wedge-1-tr-do-undo q)
                              (wedge-1-tr-undo-do q)

wedge-2 : (q : north == south) → (x₁ : de⊙ X)
          → ((merid (pt X)) ·' (! (merid (pt X))) == q ·' (! (merid x₁)))
          → (code-S q)
```

```
wedge-2 q x₁ r = [ (x₁ , shuffle-·'-inv (merid (pt X)) q (merid x₁) r) ]

wedge-12 : (q : north == south)
                  → (wedge-1 q (pt X)) == (wedge-2 q (pt X))
wedge-12 q =
  λ= (λ r → ap [_]
    (pair= idp (app= (shuffle=cancel (merid (pt X)) q) r)))

wedged-aux : (q : north == south) →
  Σ (∏∏ (de⊙ X) (de⊙ X) (λ x₁ x₂ → fst (fun-fam q x₁ x₂)))
    (λ h → (((λ x₁ → h x₁ (pt X)) ∼ wedge-2 q)) ×
           ((λ x₂ → h (pt X) x₂) ∼ wedge-1 q))
wedged-aux q = wedge-conn X X n n c c (fun-fam q)
                          (wedge-2 q) (wedge-1 q) (wedge-12 q)

wedged : (q : north == south) (x₁ x₂ : (de⊙ X))
    → ((merid x₂) ·' (! (merid (pt X))) == q ·' (! (merid x₁)))
    → (code-S q)
wedged q = fst (wedged-aux q)

wedged-pt : (q : north == south) (x₂ : de⊙ X)
          → ((merid x₂) ·' (! (merid (pt X))) == q ·' (! (merid (pt X))))
          → (code-S q)
wedged-pt q x₂ = wedged q (pt X) x₂

wedged-pt-to-wedge-1-aux : (q : north == south)
                           → (wedged-pt q) ∼ (wedge-1 q)
wedged-pt-to-wedge-1-aux q = snd (snd (wedged-aux q))

wedged-pt-to-wedge-1 : (q : north == south) →
  (λ t → wedged-pt q (fst t) (snd t)) ∼
  (λ t → wedge-1 q (fst t) (snd t))
wedged-pt-to-wedge-1 q =
  λ t → app= (wedged-pt-to-wedge-1-aux q (fst t)) (snd t)

wedged-tr : (q : north == south) (x₁ : (de⊙ X))
            → (code-N (q ·' (! (merid x₁)))) → (code-S q)
wedged-tr q x₁ = Trunc-rec λ t → wedged q x₁ (fst t) (snd t)

wedged-tr-pt : (q : north == south)
               → (code-N (q ·' (! (merid (pt X))))) → (code-S q)
wedged-tr-pt q = wedged-tr q (pt X)

wedged-tr-pt-to-wedge-1-tr : (q : north == south)
```

```
                                → (wedged-tr-pt q) ∼ (wedge-1-tr q)
wedged-tr-pt-to-wedge-1-tr q =
  trunc-pres-∼ Trunc-level (wedged-pt-to-wedge-1 q)

wedged-tr-pt-is-equiv : (q : north == south) → is-equiv (wedged-tr-pt q)
wedged-tr-pt-is-equiv q =
  ∼-preserves-equiv (∼-! (wedged-tr-pt-to-wedge-1-tr q))
                                (wedge-1-tr-is-equiv q)

equiv-fam : (x : de⊙ X) → (⟨ 0 ⟩₋₁ -Type i)
equiv-fam x = (∏ (north == south) (λ q → is-equiv (wedged-tr q x))) ,
              ∏-level (λ q → is-equiv-is-prop)

wedged-tr-is-equiv : (x : de⊙ X) (q : north == south)
                    → is-equiv (wedged-tr q x)
wedged-tr-is-equiv =
  conn-extend (pointed-conn-out (de⊙ X) (pt X) {{n-to-0-conn c}})
                    equiv-fam λ _ q → wedged-tr-pt-is-equiv q

wedged-tr-equiv : (x : de⊙ X) (q : north == south) →
  code-N (q ·' (! (merid x))) ≃ code-S q
wedged-tr-equiv x q = (wedged-tr q x) , (wedged-tr-is-equiv x q)

code-coherence : (x : de⊙ X) → code-N == code-S
                            [ (λ z → north == z → Type i) ↓ merid x ]
code-coherence x = <- (paths-↓-reduce (merid x) (code-N) (code-S))
                      λ q → wedged-tr-equiv x q

code : (y : Susp (de⊙ X)) → (north == y) → Type i
code = Susp-elim code-N code-S code-coherence

code-transp : (y : Susp (de⊙ X)) (q : north == y)
              → (code north idp) → (code y q)
code-transp y q = transport (uncurry code) (pair= q (tid q))

code-center : (y : Susp (de⊙ X)) (q : north == y) → code y q
code-center y q = code-transp y q [ pt X , !-inv'-r (merid (pt X)) ]

decomp-tid-path : {y : Susp (de⊙ X)} (p q : north == y) →
  (pair= (p ·' (! q)) (tid (p ·' (! q)))) ==
  ((pair= p (tid p)) ·' (pair= (! q) (path-fib-↓-conc' p (! q))))
decomp-tid-path p idp = idp

factor1 : (y : Susp (de⊙ X)) (q : north == y)
```

```
          → ((code north idp) → (code north (q ·' (! q))))
factor1 y q = Trunc-fmap f where
  f : hfiber σloop' idp → hfiber σloop' (q ·' (! q))
  f (x₁ , r) = x₁ , undo-·'-inv ((merid x₁) ·' (! (merid (pt X)))) q r

factor1-id : (c : code north idp) → (factor1 north idp c) == c
factor1-id = Trunc-elim-aux (λ c → (factor1 north idp c) == c)
  (λ c → Trunc-=-level (factor1 north idp c) c) λ a → idp

factor2 : {y : Susp (de⊙ X)} (p q : north == y)
          → (code north (p ·' (! q))) → (code y p)
factor2 {y} p q =
  -> ((-> (paths-↓-reduce q (code north) (code y)) (apd code q)) p)

factorization : (y : Susp (de⊙ X)) (q : north == y)
  → ((factor2 q q) ∘ (factor1 y q)) ==
       (transport (uncurry code) (pair= q (tid q)))
factorization .(left unit) idp = λ= (λ c → factor1-id c)

apd-code : (x₁ : de⊙ X)
  → (apd code (merid x₁)) ==
    (<- (paths-↓-reduce (merid x₁) (code north) (code south))
      λ q → wedged-tr-equiv x₁ q)
apd-code x₁ = SuspElim.merid-β (code north) (code south) code-coherence x₁

factor2-β : (x₁ x₂ : de⊙ X)
             → (factor2 (merid x₁) (merid x₂)) == (wedged-tr (merid x₁) x₂)
factor2-β x₁ x₂ =
  (factor2 (merid x₁) (merid x₂))
    =⟨ idp ⟩
  -> ((-> (paths-↓-reduce (merid x₂) (code north) (code south))
      (apd code (merid x₂))) (merid x₁))
    =⟨ ap (λ z → (-> ((-> (paths-↓-reduce (merid x₂) (code north)
        (code south)) z (merid x₁))))) (apd-code x₂) ⟩
  -> ((-> (paths-↓-reduce (merid x₂) (code north) (code south))
      (<- (paths-↓-reduce (merid x₂) (code north) (code south))
      λ q → wedged-tr-equiv x₂ q)) (merid x₁))
    =⟨ ap (λ z → -> (z (merid x₁))) (<--inv-r (paths-↓-reduce (merid x₂)
        (code north) (code south)) (λ q → wedged-tr-equiv x₂ q)) ⟩
  (-> (wedged-tr-equiv x₂ (merid x₁)))
    =⟨ idp ⟩
  wedged-tr (merid x₁) x₂ =∎

transp-to-factor2 : {y : Susp (de⊙ X)} (p q : north == y) →
```

```
          (transport (uncurry code) (pair= q
          (path-fib-↓-conc'-inv p q))) ==
          (factor2 p q)
transp-to-factor2 p idp = idp

transport-tid : (x1 : de⊙ X) →
  (transport (uncurry code) (pair= (merid x1) (tid (merid x1)))
      [ pt X , !-inv'-r (merid (pt X)) ]) == [ x1 , idp ]
transport-tid x1 =
  (transport (uncurry code) (pair= (merid x1)
       (tid (merid x1))) [ pt X , !-inv'-r (merid (pt X)) ])
    =⟨ (app= (! (factorization south (merid x1)))
       [ pt X , !-inv'-r (merid (pt X)) ]) ⟩

  (factor2 (merid x1) (merid x1)
       (factor1 south (merid x1) [ pt X , !-inv'-r (merid (pt X)) ]))
    =⟨ (ap (λ t → factor2 (merid x1) (merid x1) [ ((pt X), t) ])
       (undo-·'-triv (merid (pt X)) (merid x1))) ⟩

  (factor2 (merid x1) (merid x1)
       [ ((pt X) , (inv-·'-eq (merid (pt X)) (merid x1))) ])
    =⟨ app= (factor2-β x1 x1)
       [ ((pt X) , (inv-·'-eq (merid (pt X)) (merid x1))) ] ⟩

  wedged-tr (merid x1) x1 [ ((pt X) , (inv-·'-eq (merid (pt X)) (merid x1))) ]
    =⟨ idp ⟩

  wedged (merid x1) x1 (pt X) (inv-·'-eq (merid (pt X)) (merid x1))
    =⟨ app= ((fst (snd (wedged-aux (merid x1)))) $ x1)
         (inv-·'-eq (merid (pt X)) (merid x1)) ⟩

  wedge-2 (merid x1) x1 (inv-·'-eq (merid (pt X)) (merid x1))
    =⟨ idp ⟩

  [ (x1 , (shuffle-·'-inv (merid (pt X)) (merid x1) (merid x1)
      (inv-·'-eq (merid (pt X)) (merid x1)))) ]
    =⟨ ap (λ z → [ (x1 , z) ]) (undo-·'-eq (merid (pt X)) (merid x1)) ⟩

  [ (x1 , idp) ] =∎

transport-path-fib-↓ : (x1 : de⊙ X) →
  (transport (uncurry code) (! (pair= (merid (pt X))
  (path-fib-↓-conc'-inv (merid x1)
    (merid (pt X))))) [ x1 , idp ]) == [ x1 , idp ]
```

```
transport-path-fib-↓ x1 =
  transp!-shuffle (uncurry code) ((pair= (merid (pt X))
  (path-fib-↓-conc'-inv (merid x1) (merid (pt X)))))
  (transport (uncurry code) ((pair= (merid (pt X))
      (path-fib-↓-conc'-inv (merid x1) (merid (pt X)))))
      [ x1 , idp ]
    =⟨ app= (transp-to-factor2 (merid x1) (merid (pt X))) [ x1 , idp ] ⟩

  (factor2 (merid x1) (merid (pt X)) [ x1 , idp ])
    =⟨ app= (factor2-β x1 (pt X)) [ x1 , idp ] ⟩

  wedged-tr (merid x1) (pt X) [ x1 , idp ]
    =⟨ idp ⟩

  wedged (merid x1) (pt X) x1 idp
    =⟨ app= ((snd (snd (wedged-aux (merid x1)))) $ x1) idp ⟩

  wedge-1 (merid x1) x1 idp
    =⟨ idp ⟩

  [ (x1 , (cancel-·'-r (merid x1) (merid x1) (! (merid (pt X))) idp)) ]
    =⟨ ap (λ z → [ (x1 , z) ])
          (cancel-·'-r-idp (merid x1) (! (merid (pt X)))) ⟩

  [ (x1 , idp) ] =■)

code-contraction-lemma : (q : (north' (de⊙ X)) == (north' (de⊙ X)))
    (a : hfiber σloop' q) → code-center north q == [ a ]
code-contraction-lemma .(glue x1 ·' ! (glue (pt _))) (x1 , idp) =
  code-center north (merid x1 ·' ! (merid (pt X)))
    =⟨ idp ⟩

  code-transp north (merid x1 ·' ! (merid (pt X)))
                [ pt X , !-inv'-r (merid (pt X)) ]
    =⟨ id-path-transp (decomp-tid-path (merid x1) (merid (pt X)))
                [ pt X , !-inv'-r (merid (pt X)) ] ⟩

  transport (uncurry code) ((pair= (merid x1) (tid (merid x1))) ·'
        (pair= (! (merid (pt X))) (path-fib-↓-conc' (merid x1)
        (! (merid (pt X))))))
        [ pt X , !-inv'-r (merid (pt X)) ]
    =⟨ transp-·' (pair= (merid x1) (tid (merid x1)))
        (pair= (! (merid (pt X))) (path-fib-↓-conc' (merid x1)
        (! (merid (pt X)))))
        [ pt X , !-inv’-r (merid (pt X)) ] ⟩

  transport (uncurry code) (pair= (! (merid (pt X)))
        (path-fib-↓-conc’ (merid x₁) (! (merid (pt X)))))
        (transport (uncurry code) (pair= (merid x₁) (tid (merid x₁)))
        [ pt X , !-inv’-r (merid (pt X)) ])
    =⟨ ap (λ z → (transport (uncurry code) (pair= (! (merid (pt X)))
        (path-fib-↓-conc’ (merid x₁) (! (merid (pt X)))))) z)
        (transport-tid x₁) ⟩

  (transport (uncurry code) (pair= (! (merid (pt X)))
        (path-fib-↓-conc’ (merid x₁) (! (merid (pt X))))) [ x₁ , idp ])
    =⟨ id-path-transp (invert-path-fib (merid x₁)
        (merid (pt X))) [ x₁ , idp ] ⟩

  (transport (uncurry code) (! (pair= (merid (pt X))
  (path-fib-↓-conc’-inv (merid x₁) (merid (pt X))))) [ x₁ , idp ])
    =⟨ transport-path-fib-↓ x₁ ⟩

  [ x₁ , idp ] =∎

code-contraction-aux : (q : (north’ (de⊙ X)) == (north’ (de⊙ X)))
                        → (v : code north q) → (code-center north q == v)
code-contraction-aux q =
  Trunc-elim-aux (λ v → code-center north q == v)
                 (λ v → Trunc-=-level (code-center north q) v)
                 (code-contraction-lemma q)

code-contraction : (y : Susp (de⊙ X)) (q : north == y) (v : code y q)
                    → (code-center y q) == v
code-contraction .(left unit) idp v = code-contraction-aux idp v

code-is-contr : (y : Susp (de⊙ X)) (q : north == y) → is-contr (code y q)
code-is-contr y q = has-level-in (code-center y q , code-contraction y q)

freudenthal-suspension-theorem’ : has-conn-fibers ⟨ n *2 ⟩ σloop’
freudenthal-suspension-theorem’ = λ q → code-is-contr north q

freudenthal-suspension-theorem : has-conn-fibers ⟨ n *2 ⟩ (σloop X)
freudenthal-suspension-theorem =
  =-pres-conn (! σloop-eq) freudenthal-suspension-theorem’
```