Ludwig-Maximilians-Universität München
Mathematisches Institut

Bachelor's Thesis

Higher Inductive Types in Homotopy Type Theory

Kajetan Söhnen

July 17, 2018

Supervision: Dr. I. Petrakis

Contents

1 Introduction 3

2 Homotopy Type Theory 4
2.1 Martin-Löf's Intensional Type Theory . . . 4
2.2 How to define types . . . 5
2.3 Universes . . . 5
2.4 Some basic types . . . 6
2.4.1 Function types . . . 6
2.4.2 Dependent function types . . . 7
2.4.3 Product types . . . 7
2.4.4 Dependent pair types . . . 9
2.4.5 The natural numbers . . . 9
2.5 Identity types . . . 11
2.5.1 Equalities . . . 13
2.5.2 Applications of path induction . . . 13
2.6 Equivalences and identities between functions and types . . . 23
2.6.1 Equivalences . . . 23
2.6.2 Identites between functions . . . 23
2.6.3 Identities between types . . . 24
2.6.4 Applications to the transport function . . . 24
2.7 Some more types . . . 26
2.7.1 The empty type . . . 26
2.7.2 The unit type . . . 26
2.7.3 The type of booleans . . . 27
2.7.4 The integers . . . 29
2.8 Univalence axiom . . . 35

3 Higher inductive Types 36
3.1 The higher interval . . . 36
3.2 The higher circle . . . 41
3.3 Suspensions . . . 45
3.4 The fundamental group of the circle . . . 51

1 Introduction

Homotopy type theory (HoTT) is, as the name suggests, the branch of mathematics which studies the connection between homotopy theory and type theory. While homotopy theory is rooted in algebraic topology and homological algebra, type theory is an alternative to set theory which is used in mathematical logic or theoretical computer science.

The type theory used in HoTT is Martins-Löf's intensional type theory (ITT) (see e.g. [3], [4] and [5]). The idea to connect type theory and homotopy theory was strongly inspired by Hofmann and Streicher's groupoid interpreation of ITT in [2], and is based on the work of Voevodsky in [8] as well as Awodey and Warren in [1].

At first both topics may look unrelated but HoTT yields many interesting results. For example, one can use type theory to directly describe topological structures like the circle or the interval with the tools of mathematical logic by using the so called Higher inductive Types (HiTs).

The goal of this thesis is to give an introduction to HoTT in general and more specifically to present some fundamental results within the theory of HiTs. the end we will be able to prove that the fundamental group of the higher circle (the circle as a HiT) is equivalent to $\mathbb{Z}$. Although this result is expected, it was only recently proven in HoTT by Licata and Shulman in [9]. The notation and base structure of this thesis is inspired by the HoTT Book [7].

2 Homotopy Type Theory

2.1 Martin-Löf's Intensional Type Theory

There are several different type theories in mathematics and theoretical computer science. In this paper we will use Martins-Löf type theory as a framework. In type theory one uses types to classify objects. If, for example, x is of the type $A \times B$, one already knows that x has the form (a, b) where a is of type A and b is of type B . In other words one knows how such an object can be constructed or decomposed. One could say that types are 'constructive sets' in the sense that a type is a set that gives you a construction rule of objects that represent that type. The key idea in HoTT, however, is to view types as spaces one can study with homotopy theory so one has to be careful when drawing parallels between type and set theory.

For example, to express that a term a is of type A we write:

$$a : A$$

and one may be tempted to read it like:

$$a \in A$$

The more appropriate interpretation in HoTT would be:

$$a \text{ is a point in the space } A$$

The most interesting application of this viewpoint is that we can interpret the logical identity $a = b$ of two objects of the same type A as the existence of some path $p : a \rightsquigarrow b$ connecting the points a and b in the space A. We therefore can consider a function $f : A \to B$ between two types as a map from the space A into the space B. These maps turn out to be (path-)continuous, since as functions they must respect identities and therefore in our interpretation respect paths. At this point one should stress that, while we have some notion of continuity of functions, there is no notion of open subsets or convergence. We treat the space only homotopically and not topologically.

As mentioned before, type theory is more constructive than set theory. To get a better understanding of that statement, we will first introduce a conception called *propositions as types*. This means that one can think of a term $a : A$ as a point in the space A and at the same time regard a as a proof of the proposition A.

Take for example the type:

$$\prod_{n,m:\mathbb{N}} n + m = m + n$$

We will see in chapter 2.2.4 that this notation describes the type of all functions that, evaluated at some $n, m : \mathbb{N}$, yield a path between $n + m$ and $m + n$ in $\mathbb{N}$. At the same time, if you read the $\prod$ as a ”for all”, you get the proposition that $\mathbb{N}$ is commutative. So in type theory we can identify the proposition with the type and can therefore construct a function from $\mathbb{N} \times \mathbb{N}$ into the type $(n + m = m + n)$, to prove the proposition that $\mathbb{N}$ is commutative.

In this setting, to prove $A \wedge B$, we simply give an element of $A \times B$ and to prove $A \Rightarrow B$ we construct an element of $A \to B$, i.e. a function from A to B. One can think of this function as a mapping of proofs of A to proofs of B. Proving propositions by constructing an element of the corresponding type turns out to be an algorithmic method. If you have, for example, a proof for the existence of an element, you already have an algorithm to find that element. This is why type theory finds application in computer science.

2.2 How to define types

Types are always defined by a certain pattern. When defining types, we need to give:

1. a formation rule, i.e. a rule on how to form new types of a specific kind of type. For example, if we have to types A and B, we may form the type $A \times B$.
2. constructors (or introduction rules) that specify how to construct elements of this type. The introduction rule of $\mathbb{N}$ for example states that Elements of $\mathbb{N}$ are generated by the fixed 0 element and the successor function $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$.
3. eliminators or elimination rules, that explain how to use elements of that type. For example the elemination rule of the function type is the application of functions, i.e. we can use a function by applying it on some element of its domain.
4. a computation rule which specifies, what happens when we apply an eliminator on a constructor of that type.
5. an optional uniqueness principle, which expresses uniqueness of maps into or out of that type. The uniqueness principle for the function type for example states that a function f is equivalent to the function $x \mapsto f(x)$ and therefore is uniquely determined by its values.

2.3 Universes

In order to define types we first need to define the surrounding setting. What do we mean when we say something is a type? The idea is to introduce a universe $\mathcal{U}_\infty$, a type which contains all the types, but that would mean $\mathcal{U}_\infty : \mathcal{U}_\infty$. Similar to set theory such a structure would be problematic. To avoid any unsound definitions we give a hierachy to our universes, i.e. we want:

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

While our universes are now ordered, we also want that all the elements of the i^{th} universe are in the $(i+1)^{th}$ universe as well.

When we talk about some type A, we mean A inhabits some univere $\mathcal{U}_i$. Most of the times we will omit the exact level i for better readability and just write $A : \mathcal{U}$.

2.4 Some basic types

2.4.1 Function types

Given two types A and B, we can form the type of functions from A to B denoted by $A \to B$. As we mentioned above, we define types by giving rules on how to construct elements of said type and how to use said type. This is the formation rule of the function type.

Given a function $f : A \to B$ and element $a : A$, we can use the function by applying it on a and we obtain an element of B denoted $f(a)$ and called the value of f at a. As mentioned before this is the elimination rule.

The more interesting question is how one can construct an element of $A \to B$, i.e. what the introduction rule for the function type looks like. There are two equivalent notations one may use when constructing a function. The first possibility is to directly define the function by giving it a name, e.g. f, and then give a defining equation for $f : A \to B$,

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

where $\Phi(x)$ is a term that may depend on the variable x. Of course one has to check that if $x : A$, $\Phi(x)$ is of the type B. To compute $f(a)$, one simply replaces the variable x in Φ by a. For example, we can define $f : \mathbb{N} \to \mathbb{N}$ by $f(x) :\equiv x + x$. While we still have to define $\mathbb{N}$, we expect that for $x : \mathbb{N}$, $x + x$ is indeed of type $\mathbb{N}$ and this therefore is a sound definition. In this case $f(21)$ would be by definiton equal to $21 + 21$.

The second possibility is to use λ-abstraction. This method is used when we don't want to introduce a name for the function we want to define. If we want to define the same function as above for some term $\Phi(x)$, we write

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

So if we consider the example frome above, we would write:

$$(\lambda(x : \mathbb{N}).x + x) : \mathbb{N} \to \mathbb{N}$$

While one of those two methods would suffice, since they only differ in notation, both possibilities bring advantages. So in this paper we use both depending on the circumstances.

2.4.2 Dependent function types

In type theory one often has to use a more general kind of functions. The idea is that the type of the value of the function may vary depending on the element of the domain to which the function is applied.

Consider a type $A : \mathcal{U}$ as well as a function B from A into the universe (we call such a function a family over A). Now we can construct the type of dependent functions denoted by

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

Elements of that type are functions F that applied to some $a : A$ yield an element $F(a)$ of the type $B(a)$ i.e. the type of $F(a)$ depends on a. For better readability we will write non dependent functions in small letters while we will begin the name of dependent functions with a capital letter. For example take the dependent function

$$\mathrm{Id} : \prod_{(A:\mathcal{U})} A \to A$$

which we define by $\mathrm{Id}(A) \equiv \mathrm{id}_A$, where id_A is the identity function on the type A. This function Id takes a type A and yields the identity function on that type. Since the type of id_A varies over A, Id is indeed a dependent function. id_A on the other hand always takes values in A and is therefore a non dependent function. One may ask, why we do not simply define a new type FUNC of all functions to avoid the dependency by setting the domain of Id to be that new type FUNC. While this would be an easy task in set theory, in type theory we lose all the information on how to use or construct an element of such a type. Therefore the concept of dependent functions is an important and often used method in type theory. It should be mentioned that we can consider every function as a dependent function, since a ordinary function is simply a dependent function where the function B in the definition above is constant.

As we will see soon, functions take a very important role when defining types, because we can use them to discribe the inherent structure of said type.

2.4.3 Product types

For types $A, B : \mathcal{U}$ we want to introduce the product type $A \times B$. The idea is that elements of that type are pairs (a,b) : $A \times B$. While pairs are defined as particular sets in set theory, in type theory ordered pairs are a primitive concept. There is a natural way how to construct an element of $A \times B$. We need $a : A$ and $b : B$ to construct $(a, b) : A \times B$.

But how can we use elements of $A \times B$? This is just the same as to ask: How can we define functions on elements of $A \times B$? For example if we want to use an element (a, b) to extract the a, this actually a function from $A \times B$ into A. For better understanding we will first consider a non dependent function. The idea is that a function from $A \times B$ into some type C is equal to a function of the type $A \to (B \to C)$, i.e. a function that takes an element of A and yields a function from B to C. Consider, for example, the function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

$$f((x, y)) \equiv x + y.$$

This function could also be written as $g : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$

$$g(x)(y) \equiv x + y$$

i.e. $g(x) = \lambda(y : \mathbb{N}).x + y$. We know, how to construct elements of $A \to (B \to C)$, since we have already studied the function type. So we now want to define functions $f : A \times B \to C$ by defining

$$f((x, y)) \equiv g(x)(y)$$

for some $g : A \to (B \to C)$. In set theory we would argue that this is possible, since every element of $A \times B$ is indeed a pair, i.e. we would define $A \times B$ to be the smallest set that contains these pairs. In type theory we have no notion of "subtypes" so we only know that there are pairs in $A \times B$, but we are not yet sure, whether there are elements in $A \times B$ with a different structure. So we do it the other way round and postulate that we can define functions in the way described above and will later be able to conclude that every element of $A \times B$ is indeed a pair.

We can rephrase that principle into a function

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

with the defining equation

$$\mathrm{Rec}_{A\times B}(C, g, (a, b)) \equiv g(a)(b).$$

We will call this function the recursor and the associated way of defining functions the recursion principle for product types. This might seem a bit strange, since no recursion is taking place. However, the idea of the recursor can be generalized and we will see that for many types the recursor will actually be recursive. The recursion principle of a type states that it is sufficient to define a function on the canonical elements (the pairs) of $A \times B$. In other words, a function that is defined on all the pairs, can be uniquely extended on all elements of $A \times B$

Now we also want to allow dependent functions. Given a family $C : A \times B \to \mathcal{U}$, we can define a dependent function $F : \prod_{(x:A\times B)} C(x)$ by providing a dependent function $G : \prod_{(x:A)} \prod_{(y:B)} C((x, y))$ and setting

$$F((x, y)) \equiv G(x)(y).$$

This is called the induction principle for product types. As before, we can rewrite this principle into a function that we will call induction for product types. Given two types A, B we have

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

Note that the recursor is just a special case of the induction function, namely if we choose C to be constant.

The induction principle expresses that we can prove something for every element of $A \times B$ by proving it for the canonical elements. To see this, consider C as a proposition for elements in $A \times B$. The induction principle now states that to prove C for all elements of $A \times B$ (i.e. find a function of the type $\prod_{x:A\times B} C(x)$) it is sufficient to give a prove that C holds for elements of the form (a, b) with $a : A$ and $b : B$ (i.e. to give a function of the type $\prod_{(a:A)} \prod_{(b:B)} C((a, b))$).

This justifies the name induction since this is exactly what the proof by induction for $\mathbb{N}$ says. We can prove something for all natural numbers by proving it for the canonical elements of $\mathbb{N}$, which is the 0 element together with elements of the form $\mathrm{succ}(n)$ where n is in $\mathbb{N}$.

As a useful example we can define the projection functions

$$\begin{aligned} \mathrm{pr}_1 &: A \times B \to A \\ \mathrm{pr}_2 &: A \times B \to B \end{aligned}$$

with the defining equations

$$\begin{aligned} \mathrm{pr}_1((a, b)) &\equiv a \\ \mathrm{pr}_2((a, b)) &\equiv b. \end{aligned}$$

2.4.4 Dependent pair types

Just as with functions, in type theory it is often useful to generalize the product types such that the type of the second component of a pair may depend on the choice of the first component. This generalized type is called dependent pair type or $\sum$-type. Given a type A and a family B : $A \to \mathcal{U}$ we may form the dependent pair type denoted by

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

The introduction rule, recursion rule and induction rule are just generalisations of the rules given for the product type and we confine ourselves to giving an example.

Consider the dependent pair type $\sum_{(x,y:A)}(x = y)$. While we still have to define the identity type $(x = y)$ in the next section, for now it is sufficient to accept that elements of that type can be considered as paths from x to y. Therefore elements of $\sum_{(x,y:A)}(x = y)$ look like (x, y, p) where x, y are points in A and p is a path connecting these points. The type of p depends both on x and y, since we only consider paths with the same start and end point to be of the same type.

2.4.5 The natural numbers

There is a strong resemblance between the way we define types and the way one defines the natural numbers in set theory.

Remark 2.1. Note that it is not yet clear why we are allowed to talk about THE natural numbers, since there might be several on of them in different universes. However we will later be able to see that all copies of the natural number are equivalent by univalence (2.8) when using the strong recursion principle mentionend in 2.23.

To define the natural numbers in type theory one first gives the introduction rule which states that there is

- a point $0 : \mathbb{N}$, as well as
- a function $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$.

We will adopt the usual notation: $1 \equiv \mathrm{succ}(0)$, $2 \equiv \mathrm{succ}(\mathrm{succ}(0))$ and so on.

The recursion principle for $\mathbb{N}$ states that we can define a function $f : \mathbb{N} \to C$ by "classical" recursion on knows from set theory. This means for every $C : \mathcal{U}$ together with

- a point $c_0 : C$, and
- a function $c_{\mathrm{succ}} : \mathbb{N} \to C \to C$,

there is a function $f : \mathbb{N} \to C$ such that

- $f(0) \equiv c_0$, and
- $f(\mathrm{succ}(n)) \equiv c_{\mathrm{succ}}(n, f(n))$.

For example, we can define the function $\mathrm{double} : \mathbb{N} \to \mathbb{N}$, which doubles its argument, by recursion. For this we take $C \equiv \mathbb{N}$ and $c_0 \equiv 0$. We define $c_{\mathrm{succ}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ by $c_{\mathrm{succ}}(n, y) \equiv \mathrm{succ}(\mathrm{succ}(y))$. By recursion on $\mathbb{N}$ we get $\mathrm{double} : \mathbb{N} \to \mathbb{N}$ such that

- $\mathrm{double}(0) \equiv 0$
- $\mathrm{double}(\mathrm{succ}(n)) \equiv \mathrm{succ}(\mathrm{succ}(\mathrm{double}(n)))$.

Now we consider the dependent elimination rule, i.e. the induction principle for $\mathbb{N}$. Given a type family $C : \mathbb{N} \to \mathcal{U}$ with

- a point $c_0 : C(0)$, and
- a function $C_{\mathrm{succ}} : \prod_{(n:\mathbb{N})} (C(n) \to C(\mathrm{succ}(n)))$

we can construct a function $F : \prod_{(n:\mathbb{N})} C(n)$ with the defining equations

- $F(0) \equiv c_0$,
- $F(\mathrm{succ}(n)) \equiv C_{\mathrm{succ}}(n, (F(n)))$

Here we see the connection between the induction principles in type theory and the classical notion of proof by induction.

The principle of proof by induction (you know from set theory) states that, to prove a property of the natural numbers $C(n)$ for all $n : \mathbb{N}$, it is sufficient to prove $C(0)$ and to prove that $C(n)$ implies $C(\mathrm{succ}(n))$ for all $n : \mathbb{N}$.

Recall that in type theory, we can represent propositions by types. Therefore a property of natural numbers is represented by a family of types $C : \mathbb{N} \to \mathcal{U}$, where the property is true for $n : \mathbb{N}$ if and only if $C(n)$ is inhabited. With this representation principle we can rewrite the principle of proof by induction into the following statement:

To construct a function $F : \prod_{(n:\mathbb{N})} C(n)$ it is sufficient to give a point $c_0 : C(0)$ and a function $C_{\mathrm{succ}} : \prod_{(n:\mathbb{N})} \left(C(n) \to C(\mathrm{succ}(n))\right)$. This is just the induction principle on $\mathbb{N}$.

2.5 Identity types

So far the types we defined, all had their counterpart in set theory and therefore looked somewhat familiar. When it comes to identities however, type theory takes a completey new approach. To use the "propositions as types" concept mentioned in the introduction, we need a type that corresponds to the proposition that $a, b : A$ are equal. We will call this kind of types identity types. If we have a type A and $a, b : A$, we may form the identity type denoted by $a =_A b$. The interesting thing is that the identity type over two elements a, b holds more information than just stating whether a and b are equal or not. One can regard elements of $a =_A b$ as paths between a and b in the space A. Just as there may be several different paths connecting two points in a space, there may be several different witnesses of $a =_A b$.

Even though the interpretation of identities as paths is extremely useful, one has to be careful. Consider a type A with just two elements a, b such that $a = b$. We regard a and b as two different points which are connected by at least one path in A. But what does this path look like? Since the points are isolated, it can not be a path in the classical sense of a continuous map into A. The key observation is that homotopy type theory provides a synthetic description of space. Synthetic geometry describes the approach to geometry where one starts with some basic notions like points and lines and some construction principles for example the line connecting two points. In contrast the analytic approach (one is probably more familiar with) represents points as points in $\mathbb{R}^n$ and lines as certain subsets of $\mathbb{R}^n$. While we won't investigate that topic further, it is important to keep that difference in mind when we consider an element of $(a = b)$ to be a path between the points a and b.

As with the other types, we need an introduction and an elimination rule. The basic way to introduce an element of $a =_A b$ is by simply knowing that a and b are the same. This is exactly what the introduction rule

$$\mathrm{Refl} : \prod_{(a:A)} (a =_A a)$$

states. This dependent function, called reflexivity, gives for each element a of A a path refl_a from a to itself. We consider refl_a to be the constant path at the point a.

The elemination rule of the identity types is called path induction and is of utmost importance to homotopy interpretations in HoTT. The basic idea of path induction is that, to prove a property for all points x, y and all paths $p : x = y$, it's sufficient to only prove that property in the cases where the elements are both x and the path is refl_x. In other words the family of identity types is generated by the constant paths refl_x.

Formally Path induction states that:

Given a family

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

and a function

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

there is a function

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

such that

$$F(x, x, \mathrm{refl}_x) \equiv c(x).$$

The best way to read this expression is to consider the family C as some statement depending on x, y and on the path connecting these two points, and c as a function that for every $x : A$ yields a prove for $C(x, x, \mathrm{refl}_x)$. Path induction now states that the function c can be extended to the general case, i.e. there is a function F that for every $x, y : A$ and $p : x =_A y$ yields a proof for $C(x, y, p)$.

One crucial observation is that path induction is not the elimination rule for the identity type but rather the elemination rule for the family of identity type, in other words, it is the $\sum$-type $\sum_{(x,y:A)}(x = y)$ which is inductively defined and not the identity type $(x = y)$. The type $\sum_{(x,y:A)}(x = y)$ could be interpreted as the space of paths in A with varying endpoints, and it makes sense that every such path can be retracted to the constant path at x (since the endpoints are not fixed). Elements of the identity type (x=y), however, have fixed endpoints (namely x and y) and therefore path induction can not be used to prove things on one specific indentity type.

For example, we can not use path induction to prove that every path $p : x = x$ is equal (i.e. homotopic) to refl_x in the space $(x = x)$, since we can not find a suitable C, however, we can use path induction to prove that the pair (x, p) is equal to the pair (x,refl_x) in the type $\sum_{(x,y:A)}(x = y)$. This again has a topological interpretation. Consider a closed path around a "hole" with the same start and endpoint. If we want to keep both the start and the endpoint of that path fixed, we can not deform that path into the constant path. If we only fixate the startpoint and let the endpoint vary, however, this deformation becomes easy just by contracting the path back to the start point.

2.5.1 Equalities

In type theory there are two different kinds of equalities. First we have the judgmental equality which is used when things are equal by definition and is denoted by $\equiv$. If one interprets $a, b : A$ as a space A with points a and b, then $a \equiv b$ means that a and b define the same point in A.

Propositional equality, denoted by $=$, on the other hand is used when two things are equal under a certain interpretation.

Consider the following two examples.

Let $A : \mathcal{U}$ be a type with two points $a, b : A$. Now imagine a path $p : a =_A b$ from a to b. Let $p^{-1} : b =_A a$ be the inverse of p. Then the path $p \cdot p^{-1}$, where we first go from a to b via p and then back the same way, is homotopic to the constant path at a. While the two paths $p \cdot p^{-1}$ and refl_a, are different objects, different points in the identity type $a =_A a$, they are equal under our interpretation of paths. So we get that $p \cdot p^{-1} =_{(a=a)} \mathrm{refl}_a$, while in general it is not true that $p \cdot p^{-1} \equiv \mathrm{refl}_a$.

Let $A : \mathcal{U}$ be a type with a point $a : A$. We already have seen the identity function $\mathrm{id}_A : A \to A$, which maps every point to itself. We have that $\mathrm{id}_A(a) \equiv a$, since $\mathrm{id}_A(a)$ is just another name for the point a. This means we can replace $\mathrm{id}_A(a)$ by a whenever we want, without worrying about any consequences. Because of that, the identity type $\mathrm{id}_A(a) =_A a$ is just another notation for the type $a =_A a$ and therefore is inhabited by refl_a.

This holds in general and therefore $a \equiv b$ is stronger than $a = b$. While this topic could be explored in more detail (see literature), we will stick to this short explanation. The important conclusion is that if things are jugmentally equal we can replace them with each other without losing any information and without changing the meaning, but if two elements are only propositional equal, one has to take care. For example, for a function $f : A \to B$ it is not clear that $a =_A b$ implies that $f(a) =_B f(b)$. To prove such propositions, one typically uses path induction.

2.5.2 Applications of path induction

In order to characterize higher inductive types, we will prove some basic properties of paths by path induction. Before we get started we need to formalize the inverses of paths as well as the composite of paths.

Lemma 2.2. *(inverse) Let $A : \mathcal{U}$ be a type. For every $x, y : A$ there is a function*

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

denoted by $p \mapsto p^{-1}$. We call p^{-1} the inverse of p. The inverse of refl_x *is* refl_x *itself.*

Proof. We want to use path induction. We fix $A : \mathcal{U}$ and define $C : \prod_{(x,y:A)}(x = y) \to \mathcal{U}$ by

$$C(x, y, p) \equiv \big((x = y) \to (y = x)\big).$$

Now we introduce our $c : \prod_{(x:A)} C(x, x, \mathrm{refl}_x)$ by setting

$$c(x) \equiv \mathrm{id}_{(x=_A x)}.$$

Note that c is well typed since $C(x, x, \mathrm{refl}_x) \equiv (x =_A x) \to (x =_A x)$. Now path induction yields a function $F : \prod_{x,y:A} \prod_{p:(x=y)} C(x, y, p)$. By definition we know that for every $x, y : A$ and $p : x =_A y$ $F(x, y, p)$ is of type $(x = y) \to (y = x)$. To complete our profe it only remains to see that

$$F(x, x, \mathrm{refl}_x)(\mathrm{refl}_x) \equiv c(x)(\mathrm{refl}_x) \equiv \mathrm{refl}_x.$$

□

These proofs in type theory are quite formal and often hard to follow. However there is also an "informal" way to use path induction.

Proof. By path induction we may assume that $y \equiv x$ and $p \equiv \mathrm{refl}_x$. In only remains to show that there is a function of the type $(x =_A x) \to (x =_A x)$ and we have $\mathrm{id}_{(x=_A x)}$ for that. □

In this paper we will use both kinds of proofs, depending on the complexity and importance of the proof.

Lemma 2.3. *(composite) Given $A : \mathcal{U}$ together with $x, y, z : A$ and two paths $p : (x = y)$ and $q : (y = z)$ we have a path*

$$p \centerdot q : (x = z)$$

We call $p \centerdot q$ the composite of p and q. For every $x : A$ we have $\mathit{refl}_x \centerdot \mathit{refl}_x \equiv \mathit{refl}_x$.

Proof. By double path induction we may assume $z \equiv y \equiv x$ and $p \equiv q \equiv \mathrm{refl}_x$ so we only have to prove that there exists a path $\mathrm{refl}_x \centerdot \mathrm{refl}_x : (x = x)$ but we have refl_x for that. □

There are some properties of the inverse and the composite of paths one expects to hold

Lemma 2.4. *For $A : \mathcal{U}$, with $x, y, z, w : A$ and $p : (x = y)$, $q : (y = z)$ and $r : (z = w)$ we have*

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

Proof. By path induction we may assume that $x \equiv y \equiv z \equiv w$ and $p \equiv q \equiv r \equiv \mathsf{refl}_x$.
It remains to prove that

1. $\mathsf{refl}_x = \mathsf{refl}_x \cdot \mathsf{refl}_x$ and $\mathsf{refl}_x = \mathsf{refl}_x \cdot \mathsf{refl}_x$, which holds by definition of the composite.
2. $\mathsf{refl}_x^{-1} \cdot \mathsf{refl}_x = \mathsf{refl}_x$ and $\mathsf{refl}_x \cdot \mathsf{refl}_x^{-1} = \mathsf{refl}_x$, which holds by 1. since $\mathsf{refl}_x^{-1} \equiv \mathsf{refl}_x$.
3. $(\mathsf{refl}_x^{-1})^{-1} = \mathsf{refl}_x$, which holds since $\mathsf{refl}_x^{-1} \equiv \mathsf{refl}_x$ by definition of the inverse.
4. $\mathsf{refl}_x \cdot (\mathsf{refl}_x \cdot \mathsf{refl}_x) = (\mathsf{refl}_x \cdot \mathsf{refl}_x) \cdot \mathsf{refl}_x$, which is true since both sides are just refl_x.

□

Note that these equalities between paths, are paths themselfs. For example that $p^{-1} \cdot p = \mathsf{refl}_y$ just means that there is a path between $p^{-1} \cdot p$ and refl_y. In other words $p^{-1} \cdot p$ and refl_y are homotopic to each other. We already have mentioned that, since functions respect identities, they also must respect paths.

Lemma 2.5. *(Application) Let $A, B : \mathcal{U}$ and $f : A \to B$ be a function. For any $a, b : A$ there is an operation*

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

Moreover, for $x : A$ we have $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$. (see Figure 2.1)

This function, called application of f to a path, is the formalisation of the fact that functions respect paths, i.e. for every path between x and y in A we get a path between $f(x)$ and $f(y)$ in B. Sometimes for better readability instead of ap_f we will simply write $f(p)$ if the context is clear.

Proof. We use path induction on $C : \prod_{(x,y:A)}(x =_A y) \to \mathcal{U}$ defined by

$$C(x, y, p) \equiv (f(x) =_B f(y))$$

By path induction it is sufficient to define a $c : \prod_{(x:A)} C(x, x, p)$. This is easily done by defining

$$c(x) \equiv \mathsf{refl}_{f(x)}.$$

Note that $\mathsf{refl}_{f(x)}$ is indeed of the the type $(f(x) =_B f(x))$, i.e. of the type $C(x, x, p)$.
Path induction now provides the function $F : (x =_A y) \to (f(x) =_B f(y))$ such that $F(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$. This function is exactly ap_f. □

While the result is expected in this case, things get more complicated if we consider a dependent function $F : \prod_{(x:A)} B(x)$. If we now want to apply F to a path $p : (x = y)$, we don't even know whether $F(x)$ and $F(y)$ are of the same type or not. The following lemma shows that the path p itself gives us the information on how to relate the types $B(x)$ and $B(y)$.

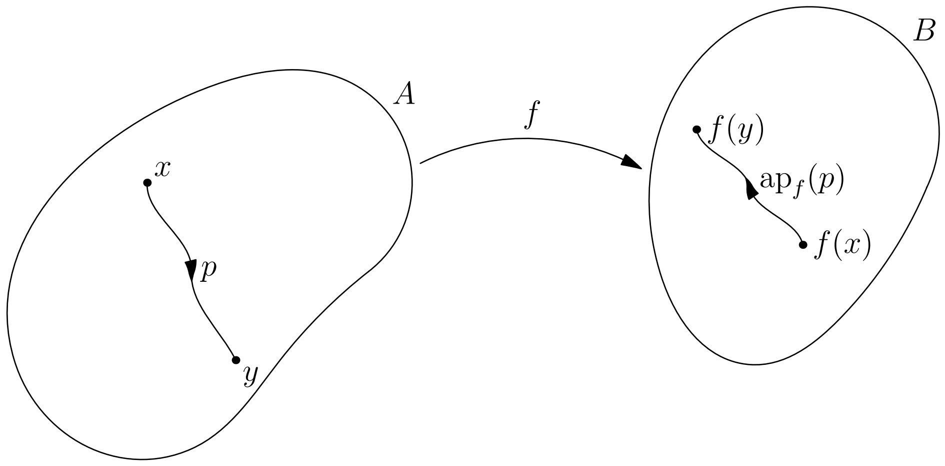

Figure 2.1: Application of a non dependent function $f : A \to B$

Lemma 2.6. *(Transport) Suppose that $B : A \to \mathcal{U}$ is a family over A. Then for every path $p : (x =_A y)$ there is an associated function $p_* : B(x) \to B(y)$. Furthermore $(\mathrm{refl}_x)_* \equiv \mathrm{id}_{B(x)}$.*

Proof. Again we use path induction. This time we set $C : \prod_{(x,y:A)}(x =_A y) \to \mathcal{U}$ by defining

$$C(x, y, p) \equiv B(x) \to B(y).$$

We then consider the function $c : \prod_{(x:A)} C(x, x, \mathrm{refl}_x)$ defined by

$$c(x) \equiv \mathrm{id}_{B(x)}$$

where $\mathrm{id}_{B(x)}$ is the identity on $B(x)$. Note that c is well typed since $\mathrm{id}_{B(x)}$ is of the type $(B(x) \to B(x))$, i.e. of the type $C(x, x, \mathrm{refl}_x)$. Path induction now gives us a function $F : \prod_{(x,y:A)} \prod_{(p:x=y)}(B(x) \to B(y))$, such that $F(x, x, \mathrm{refl}_x) \equiv \mathrm{id}_{B(x)}$. If we apply F on $x, y : A$ and $p : (x = y)$, we get a function from $B(x)$ into $B(y)$ which we name p_* . $\square$

Sometimes we will need to keep track of the type family P. In this case we will use on of the two alternative notations for p_*:

$$\mathrm{transport}^P(p, -) : P(x) \to P(y) \quad \text{or} \quad p_*^P : P(x) \to P(y)$$

Lemma 2.7. *(Path lifting property) Let P be a family over A and assume we have $u : P(x)$ for some $x : A$. Then for every $p : x =_A y$ we have*

$$\mathrm{lift}(u, p) : (x, u) = (y, p_*(u))$$

in $\sum_{(x:A)} B(x)$, such that $\mathrm{ap}_{\mathrm{pr}_1}(\mathrm{lift}(u, p)) = p$.

Proof. To use path induction we define $C : \prod_{(x,y:A)}(x =_A y) \to \mathcal{U}$ by

$$C(x,y,p) \equiv (x,u) =_{\sum_{(x:A)} B(x)} (y, p_*(u)).$$

Now we can set $c : \prod_{(x:A)} C(x,x,\mathrm{refl}_x)$ to be

$$c(x) \equiv \mathrm{refl}_{(x,u)}.$$

This c is well typed, since $\mathrm{refl}_{(x,u)}$ is of the type $((x,u) = (x,u))$, which is by definition equal to $((x,u) = (x, \mathrm{id}_{B(x)}(u)))$ and that is exactly $C(x,x,\mathrm{refl}_x)$. Path induction now yields a function $F : \prod_{(x,y:A)} \prod_{(p:x=y)}((x,u) =_{\sum_{(x:A)} B(x)} (y, p_*(u)))$. This function applied to $x, y : A$ and $p : (x = y)$ yields a path between (x,u) and (y, p_*) in $\sum_{(x:A)} B(x)$ we call $\mathrm{lift}(u,p)$. Note that by path induction we also get that $\mathrm{lift}(u, \mathrm{refl}_x) \equiv \mathrm{refl}_{(x,u)}$.

It remains to show that $\mathrm{ap}_{\mathrm{pr}_1}(\mathrm{lift}(u,p)) = p$. For this we will use path induction a second time. This time consider $D : \prod_{(x,y:A)}(x =_A y) \to \mathcal{U}$ with

$$D(x,y,p) \equiv (\mathrm{pr}_1(\mathrm{lift}(u,p)) =_{x=y} p).$$

This means we set $D(x,y,p)$ to be the identity type over the identity type $(x = y)$. Now we can set $d : \prod_{(x:A)} D(x,x,\mathrm{refl}_x)$ to be

$$d(x) \equiv \mathrm{refl}_{\mathrm{refl}_x}.$$

To show that this is well typed, we have to show that the type $D(x,x,\mathrm{refl}_x)$ is by definition equal to the identity type $(\mathrm{refl}_x =_{(x=x)} \mathrm{refl}_x)$. To see this, first note that

$$D(x,x,\mathrm{refl}_x)$$

is by definition equal to

$$(\mathrm{pr}_1(\mathrm{lift}(u,\mathrm{refl}_x)) =_{x=x} \mathrm{refl}_x).$$

This can be simplified to

$$(\mathrm{pr}_1(\mathrm{refl}_{(x,u)})) =_{x=x} \mathrm{refl}_x)$$

by applying that $\mathrm{lift}(u,\mathrm{refl}_x) \equiv \mathrm{refl}_{(x,u)}$. By the application lemma this is, however, the same as

$$(\mathrm{refl}_x =_{x=x} \mathrm{refl}_x)$$

and therefore our d is well typed. Now we apply path induction and get a $G : \prod_{(x,y:A)} \prod_{(p:x=y)}(\mathrm{pr}_1(\mathrm{lift}(u,p))$ $p)$ to finish the proof. $\square$

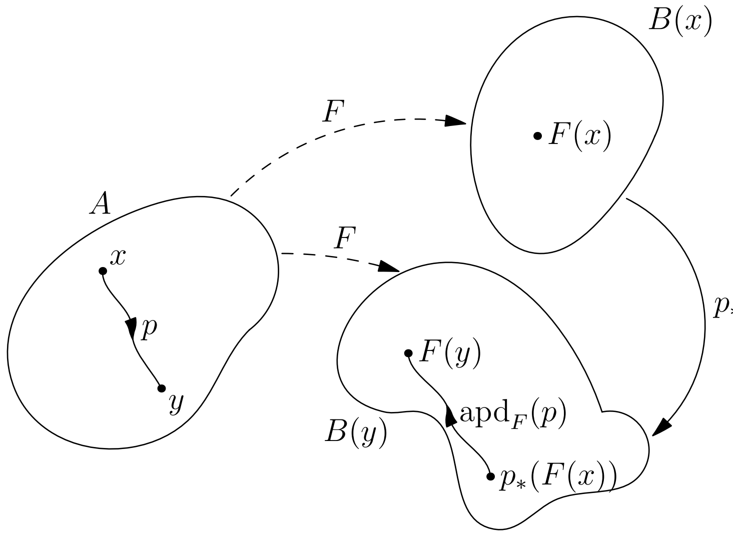

Figure 2.2: Dependent application of a function $F : \prod_{x:A} B(x)$

With this lemma we can give the dependent version of function application:

Lemma 2.8. *(Dependent application) Let* $F : \prod_{x:A} B(x)$ *be a dependent function. Then we have a function*

$$\mathrm{apd}_F : \prod_{p:(x=y)} (p_*(F(x)) =_{B(y)} F(y).$$

(see Figure 2.2)

Proof. Consider $C : \prod_{(x,y:A)}(x =_A y) \to \mathcal{U}$ to be defined by

$$C(x,y,p) \equiv (p_*(F(x)) =_{B(y)} F(y).$$

Now we define $c : \prod_{(x:A)} C(x,x,\mathrm{refl}_x)$ by

$$c(x) \equiv \mathrm{refl}_{F(x)}$$

which is well defined, since $\mathrm{refl}_{F(x)}$ is of the type $F(x) =_{B(x)} F(x)$, which is exactly $C(x,x,\mathrm{refl}_x)$, because $(\mathrm{refl}_x)_*(F(x))$ is by definition equal to $F(x)$. Path induction now provides us with $\mathrm{apd}_F : \prod_{p:(x=y)}(p_*(F(x)) =_{B(y)} F(y)$, where $\mathrm{apd}_F(\mathrm{refl}_x) \equiv \mathrm{refl}_{F(x)}$. ☐

We already have seen that every function $f : A \to B$ can be interpreted as a dependent function $F : \prod_{(x:A)P(x)}$ where $P(x) \equiv B$. In this case ap_f and apd_F are closely related, as seen in the following two lemmata:

Lemma 2.9. *(transportconst) Let* $A : \mathcal{U}$ *and* $P : A \to \mathcal{U}$ *with* $P(x) \equiv B$ *for a fixed* $B : \mathcal{U}$*. For all* $b : B$*,* $x, y : A$ *and for every path* $p : x =_A y$ *we have a path*

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

such that $\mathrm{transportconst}^B_{\mathrm{refl}_x}(b) \equiv \mathrm{refl}_b$

Proof. Fix $b : B$ and define $C : \prod_{x,y:A}(x = y) \to \mathcal{U}$ by

$$C(x, y, p) \equiv (\mathrm{transport}^P(p, b) = b).$$

By definition of transport we have

$$\begin{aligned} C(x, x, \mathrm{refl}_x) &\equiv (\mathrm{transport}^P(\mathrm{refl}_x, b) = b) \\ &\equiv (\mathrm{id}_{P(x)}(b) = b) \\ &\equiv (b = b) \end{aligned}$$

so we can define $c : \prod_{(x:A)} C(x, x, \mathrm{refl}_x)$ by

$$c(x) \equiv \mathrm{refl}_b.$$

Path induction now yields a function $F : \prod_{(x,y:A)} \prod_{p:(x=y)}(\mathrm{transport}^P(p, b) = b)$ with $F(x, x, \mathrm{refl}_x) \equiv \mathrm{refl}_b$, so we can define

$$\mathrm{transportconst}^B_p(b) \equiv F(x, y, p).$$

□

As with transport we introduce an alternative notation. When it is not too confusing, we can omit the type B and the point b and simply write p_{**} instead of $\mathrm{transportconst}^B_p(b)$.

With this result we can prove:

Lemma 2.10. *For $A, B : \mathcal{U}$, $f : A \to B$ and $p : x =_A y$, we have*

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

(see Figure 2.3)

Proof. By path induction we can assume $y \equiv x$ and $p \equiv \mathrm{refl}_x$ so we only have to prove

$$\mathrm{apd}_f(\mathrm{refl}_x) = \mathrm{transportconst}^B_{\mathrm{refl}_x}(f(x)) \cdot \mathrm{ap}_f(\mathrm{refl}_x)$$

By definition we have

- $\mathrm{apd}_f(\mathrm{refl}_x) \equiv \mathrm{refl}_{f(x)}$
- $\mathrm{transportconst}^B_{\mathrm{refl}_x}(f(x)) \equiv \mathrm{refl}_{f(x)}$
- $\mathrm{ap}_f(\mathrm{refl}_x) \equiv \mathrm{refl}_{f(x)}$

so we only have to prove

$$\mathrm{refl}_{f(x)} = \mathrm{refl}_{f(x)} \cdot \mathrm{refl}_{f(x)}$$

but we have $\mathrm{refl}_{\mathrm{refl}_{f(x)}}$ for that. □

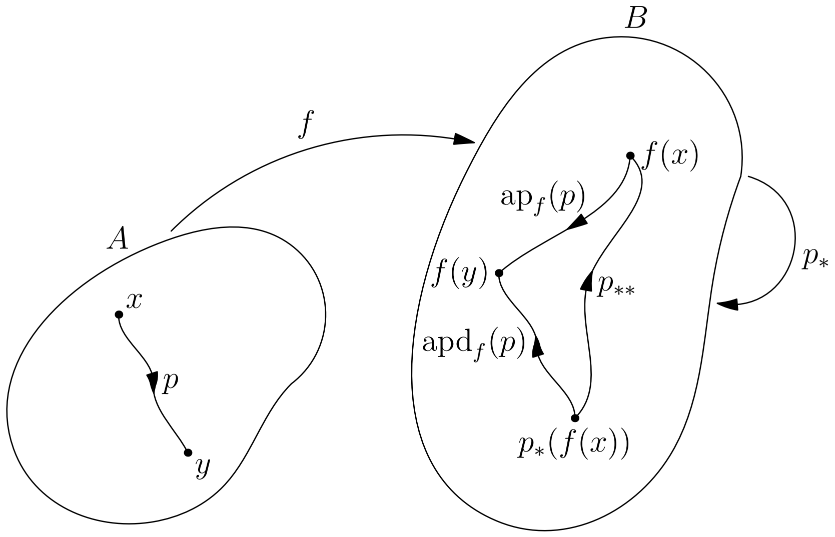

Figure 2.3: The connection between ap and apd

In the next lemmata we will take a closer look at the transport function and give some identities in special cases.

Lemma 2.11. *Given $f, g : A \to B$, paths $p : a_1 =_A a_2$ and $q : f(a_1) =_B g(a_1)$, we have*

$$\mathrm{transport}^{\lambda x.(f(x) =_B g(x))}(p, q) =_{f(a_2)=g(a_2)} (\mathrm{ap}_f(p))^{-1} \cdot q \cdot \mathrm{ap}_g(p)$$

Proof. We fix f, g and then use path induction on $C : \prod_{(a_1,a_2:A)}(a_1 = a_2) \to \mathcal{U}$ with

$$C(a_1, a_2, p) \equiv \left(\mathrm{transport}^{\lambda x.(f(x)=_B g(x))}(p, q) =_{f(a_2)=g(a_2)} (\mathrm{ap}_f(p))^{-1} \cdot q \cdot \mathrm{ap}_g(p)\right)$$

By path induction we may assume $a_1 \equiv a_2$ and that p is refl_{a_1}. So we only have to prove

$$\left(\mathrm{transport}^{\lambda x.(f(x)=_B g(x))}(\mathrm{refl}_{a_1}, q) =_{f(a_1)=g(a_1)} (\mathrm{ap}_f(\mathrm{refl}_{a_1}))^{-1} \cdot q \cdot \mathrm{ap}_g(\mathrm{refl}_{a_1})\right).$$

Since we know that

- $\mathrm{transport}^{\lambda x.(f(x)=_B g(x))}(\mathrm{refl}_{a_1}, -)$ is the identity function on $(f(a_1) =_B g(a_1))$
- $\mathrm{ap}_f(\mathrm{refl}_{a_1}) \equiv \mathrm{refl}_{f(a_1)}$
- $\mathrm{ap}_g(\mathrm{refl}_{a_1}) \equiv \mathrm{refl}_{g(a_1)}$

it is sufficient to prove that

$$q =_{f(a_1)=g(a_1)} (\mathrm{refl}_{f(a_1)})^{-1} \cdot q \cdot \mathrm{refl}_{g(a_1)}.$$

□

There are some important special cases of lemma 2.11:

Lemma 2.12. *Let $A : \mathcal{U}$ be a type with a point $a : A$ and a path $p : x =_A y$. Then we have*

$$\begin{aligned}
&\mathrm{transport}^{\lambda x.(a=x)}(p, q) = q \cdot p && \textit{for } q : a = x\\
&\mathrm{transport}^{\lambda x.(x=a)}(p, q) = p^{-1} \cdot q && \textit{for } q : x = a\\
&\mathrm{transport}^{\lambda x.(x=x)}(p, q) = p^{-1} \cdot q \cdot p && \textit{for } q : x = x.
\end{aligned}$$

Proof. We use lemma 2.11 with f, g as the identity function or a constant function $\lambda x.a$. It only remains to see that $\mathrm{ap}_{\lambda x.a}(l) = \mathrm{refl}_a$ for all $l : x =_A y$. That, however, is easily proven by using path induction on l since for all $x : A$ $\mathrm{ap}_{\lambda x.a}(\mathrm{refl}_x) = \mathrm{refl}_a$. □

Lemma 2.13. *Let $A : \mathcal{U}$ and $P, Q : A \to \mathcal{U}$ be two type families over A. Then*

$$p_*^{\lambda z.(P(z)\to Q(z))}(f) = p_*^Q \circ f \circ (p^{-1})_*^P$$

holds for all $x, y : A$ and $p : x = y$ and $f : P(x) \to Q(x)$.

Proof. First note that this is well typed. $(p^{-1})_*^P$ takes an element from $P(y)$ and yields an element in $P(x)$. Therefore we can apply f and get a point in $Q(x)$. Now we can carry out the transport p_* with respect to Q and get a point in $Q(y)$. So on the right, we have a function from $P(y)$ into $Q(y)$. By definiton $p_*^{\lambda z.(P(z)\to Q(z))}(f)$ is of type $P(y) \to Q(y)$ so the statement above is indeed well typed.

Now we use path induction and define $C : \prod_{x,y:A}(x = y) \to \mathcal{U}$ by

$$C(x, y, p) \equiv \prod_{f:P(x)\to Q(x)} p_*^{\lambda z.(P(z)\to Q(z))}(f) = p_*^Q \circ f \circ (p^{-1})_*^P.$$

By path induction it is sufficient to prove $C(x, x, \mathrm{refl}_x)$.

$$\begin{aligned}
C(x, x, \mathrm{refl}_x) &\equiv \prod_{f:P(x)\to Q(x))} \left((\mathrm{refl}_x)_*^{\lambda x.(P(x)\to Q(x))}(f) = (\mathrm{refl}_x)_*^Q \circ f \circ (\mathrm{refl}_x^{-1})_*^P\right)\\
&= \prod_{f:P(x)\to Q(x)} \left(\mathrm{id}_{P(x)\to Q(x)}(f) = \mathrm{id}_Q \circ f \circ \mathrm{id}_P\right)\\
&= \prod_{f:P(x)\to Q(x)} (f = f).
\end{aligned}$$

But we have $\lambda f.\mathrm{refl}_f : \prod_{f:P(x)\to Q(x)} f = f$ thereby completing the proof. □

Lemma 2.14. *Given $A : \mathcal{U}$, $P : A \to \mathcal{U}$ together with $p : x =_A y$, $q : y =_A z$ and a term $u : P(x)$, we have*

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

Proof. We define $C : \prod_{x,y:A}(x = y) \prod_{z:A}(y = z) \to \mathcal{U}$ by

$$C(x, y, p, z, q) \equiv (q_*(p_*(u)) = (p \cdot q)_*(u)).$$

We prove $C(x, y, p, z, q)$ for all $x, y, z : A$ and $p : x =_A y$, $q : y =_A z$. By path induction we can assume that $y \equiv x$ and that $p \equiv \mathsf{refl}_x$ so we only have to prove $C(x, x, \mathsf{refl}_x, z, q)$ which by definition is

$$q_*((\mathsf{refl}_x)_*(u)) = (\mathsf{refl}_x \cdot q)_*(u).$$

Now we want to prove $C(x, x, \mathsf{refl}_x, z, q)$ for all $x, z : A$ and $q : x = z$. Again we can use path induction and assume that $z \equiv x$ and $q \equiv \mathsf{refl}_x$. With this assumption we only have to prove that for all $x : A$ we have

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

Since by definition we have $(\mathsf{refl}_x \cdot \mathsf{refl}_x) \equiv \mathsf{refl}_x$ as well as $(\mathsf{refl}_x)_* \equiv \mathrm{id}_{P(x)}$, to prove the equation above, is just to prove

$$\mathrm{id}_{P(x)}(\mathrm{id}_{P(x)}(u)) = \mathrm{id}_{P(x)}(u)$$

which by definition is just $u = u$, which trivially holds. $\square$

Lemma 2.15. *Consider $A, B : \mathcal{U}$ with a function $f : A \to B$ together with a type family $P : B \to \mathcal{U}$. Then for any $p : x =_A y$ and $u : P(f(x))$ we have*

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

Proof. First see that the equality above is well typed. $\mathsf{transport}^{P \circ f}(p, -)$ is a function from $P(f(x))$ into $P(f(y))$. And since $\mathsf{ap}_f(p)$ is a path from $f(x)$ to $f(y)$, $\mathsf{transport}^P(\mathsf{ap}_f(p), -)$ is of type $P(f(x)) \to P(f(y))$ as well.

By path induction we may assume that $y \equiv x$ and $p \equiv \mathsf{refl}_x$. So we only need to prove

$$\mathsf{transport}^{P \circ f}(\mathsf{refl}_x, u) = \mathsf{transport}^P(\mathsf{ap}_f(\mathsf{refl}_x), u).$$

Since by definition $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$ and $\mathsf{transport}(\mathsf{refl}_x, u) \equiv u$, it suffices to prove

$$u = u.$$

$\square$

2.6 Equivalences and identities between functions and types

2.6.1 Equivalences

We first need to define what we mean by saying two functions f and g are inverse to each other.

We call two functions $f : A \to B$ and $g : B \to A$ quasi-inverse if we know that:

- $g(f(a)) = a$ for all $a : A$
- $f(g(b)) = b$ for all $b : B$.

In this case we call g a quasi-inverse of f, call f and g equivalences and say that A and B are equivalent, denoted by $A \simeq B$.

Note that we use the propositional equality instead of the stronger judgmental equality here. If one interprets A and B as spaces, this means that the applicaton of $g \circ f$ to a point $a : A$ will not necessarily yield the same point a, but rather some other point $\tilde{a}$ which is connected to the point a via some path.

2.6.2 Identites between functions

We expect that two function are equal if and only if they are equal at every point. In type theory this means that we expect, that for functions $f, g : A \to B$ there is an equivalence

$$(f =_{A \to B} g) \simeq \left(\prod_{x:A} f(x) =_B g(x) \right).$$

Lemma 2.16. *There is a function* $(f =_{A \to B} g) \to \left(\prod_{x:A} f(x) =_B g(x) \right)$.

Proof. Given a $f, g : A \to B$ and a $p : f = g$ we need to give a function of type $\prod_{x:A} f(x) =_B g(x)$. By path induction we may assume $f \equiv g$ and $p \equiv \mathrm{refl}_f$. So we only need to give a function $\left(\prod_{x:A} f(x) =_B f(x) \right)$, which is easily done by

$$(\lambda(x : A).\mathrm{refl}_{f(x)}).$$

□

We will be able to prove that there is also a function

$$\left(\prod_{x:A} f(x) =_B g(x) \right) \to (f =_{A \to B} g)$$

called function extensionality once we have defined the higher interval.

2.6.3 Identities between types

Given $A, B : \mathcal{U}$ we can form the identity type $A =_{\mathcal{U}} B$. We also have a notion for the type of equivalences from A to B denoted by $(A \simeq B)$. It feels natural that those two types are equivalent in a way i.e. that there is an equivalence $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$.

Lemma 2.17. *(idtoeqv) For $A, B : \mathcal{U}$, there is a function,*

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

defined by $\mathrm{idtoeqv} \equiv \mathrm{transport}^{\mathrm{id}_{\mathcal{U}}}$.

Proof. We can consider the type family $P : \mathcal{U} \to \mathcal{U}$ which is induced by the identity function on $\mathcal{U}$. This means we define $P(X) \equiv X$. Now we can take the transport function $\mathrm{transport}^P$. This function maps every path $p : A = B$ to a function $p_* : A \to B$. We claim that p_* is an equivalence. By path induction it suffices to prove that $(\mathrm{refl}_A)_*$ is an equivalence for all $A : \mathcal{U}$. By definition of transport we have $(\mathrm{refl}_A)_* \equiv \mathrm{id}_A$, and the identity function is obviously an equivalence. □

We expect that there is a quasi-inverse to idtoeqv, but type theory can not guarantee this. We will come back to this topic in chapter 2.8.

2.6.4 Applications to the transport function

With these definitions and lemmata we can prove some more interesting properties for transport.

Lemma 2.18. *Let $A : \mathcal{U}$ be a type. For $p : a =_A \tilde{a}$, $q : a =_A a$ and $r : \tilde{a} =_A \tilde{a}$, we have*

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

Proof. By path induction on p it suffices to prove that

$$(q = r) \simeq (q \cdot \mathrm{refl}_a = \mathrm{refl}_a \cdot r).$$

Since this is our first proof of the existence of an equivalence, we will do it in detail. First note that there are two paths

$$k : q \cdot \mathrm{refl}_a = q$$
$$j : r = \mathrm{refl}_a \cdot r.$$

Now we can define $f : (q = r) \to (q \cdot \mathrm{refl}_a = \mathrm{refl}_a \cdot r)$ by

$$f(l) \equiv k \cdot l \cdot j.$$

as well as $g : (q \cdot \mathrm{refl}_a = \mathrm{refl}_a \cdot r) \to (q = r)$ by

$$g(l) \equiv k^{-1} \cdot l \cdot j^{-1}.$$

We claim that f and g are quasi-inverse. First we prove that for every $l : (q = r)$ we have $g(f(l)) = l$. Since

$$g(f(l)) \equiv k^{-1} \cdot k \cdot l \cdot j \cdot j^{-1}$$

and $k^{-1} \cdot k = \mathrm{refl}_q$ as well as $j \cdot j^{-1} = \mathrm{refl}_r$ this is just proving

$$\mathrm{refl}_q \cdot l \cdot \mathrm{refl}_r = l$$

which obviously holds. Now consider some $l : (q \cdot \mathrm{refl}_a = \mathrm{refl}_a \cdot r)$. We prove that $f(g(l)) = l$. By definition we have

$$f(g(l)) \equiv k \cdot k^{-1} \cdot l \cdot j^{-1} \cdot j.$$

Since $k \cdot k^{-1} = \mathrm{refl}_{q \cdot \mathrm{refl}_a}$ and $j^{-1} \cdot j = \mathrm{refl}_{\mathrm{refl}_a \cdot r}$ we only need to see that

$$\mathrm{refl}_{q \cdot \mathrm{refl}_a} \cdot l \cdot \mathrm{refl}_{\mathrm{refl}_a \cdot r} = l.$$

$\square$

We also can prove that the transport functions for p and p^{-1} are quasi-inverse.

Lemma 2.19. *Given $A : \mathcal{U}$ together with $P : A \to \mathcal{U}$ $p : x =_A y$, we have*

$$(p^{-1})^P_* = (p^P_*)^{-1}.$$

Proof. By path induction we can assume that $x \equiv y$ and $p \equiv \mathrm{refl}_x$, so we only need to prove that for every $x : A$, $(\mathrm{refl}_x^{-1})^P_*$ and $(\mathrm{refl}_x)^P_*$ are quasi-inverse. Since $\mathrm{refl}_x^{-1} \equiv \mathrm{refl}_x$ and $(\mathrm{refl}_x)^P_* \equiv \mathrm{id}_{P(x)}$ this is just proving that $\mathrm{id}_{P(x)}$ is a quasi-inverse to itself, which is of course true. $\square$

Lemma 2.20. *Given $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ together with $x, y : A$, a path $p : x =_A y$ and a point $u : B(x)$, we have*

$$\mathrm{transport}^B(p, u) = \mathrm{idtoeqv}(\mathrm{ap}_g(p))(u).$$

Proof. First we rewrite B into $\mathrm{id}_{\mathcal{U}} \cdot B$. Now lemma 2.15 yields

$$\mathrm{transport}^{\mathrm{id}_{\mathcal{U}} \cdot B}(p, u) = \mathrm{transport}^{\mathrm{id}_{\mathcal{U}}}(\mathrm{ap}_B(p), u).$$

By definition we have $\mathrm{transport}^{\mathrm{id}_{\mathcal{U}}}(\mathrm{ap}_B(p), u) \equiv \mathrm{idtoeqv}(\mathrm{ap}_g(p))(u)$, so we are done. $\square$

2.7 Some more types

With our new knowledge of paths and equivalences we can define some more types.

2.7.1 The empty type

Like the empty set in set-theory, there is an empty type, denoted by $\mathbf{0} : \mathcal{U}$, in type theory.

While there is no way to construct elements of $\mathbf{0}$, it is easy to construct functions from the empty type into other sets. Indeed we have a function from $\mathbf{0}$ into every other type. Therefore the recursor for $\mathbf{0}$ is

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

The induction principle on $\mathbf{0}$ states that any property is true for all terms in the type $\mathbf{0}$. Thus the induction function for $\mathbf{0}$ is given by

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

The empty type gives us the oportunity to define the negation of an property A.
If we want to prove that "not A" is true, we give a function $f : A \to \mathbf{0}$.

2.7.2 The unit type

After defining the empty type, we now will define the unit type $\mathbf{1} : \mathcal{U}$ with just one element. The introduction rule is just that there is an term $\star : \mathbf{1}$.

The recursor for the unit type states for every $C : \mathcal{U}$ with a term $c : C$ there is a function $\mathrm{rec}_{\mathbf{1}}(C, c, -) : \mathbf{1} \to C$ such that $\mathrm{rec}_{\mathbf{1}}(C, c, \star) \equiv c$.

The induction principle for $\mathbf{1}$ is given by the function

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

In words this means that to prove a property for every element in $\mathbf{1}$ it is sufficient to prove it only for $\star$. With the induction principle we can now prove the uniqueness principle for $\mathbf{1}$, namely that every element of $\mathbf{1}$ is equal to $\star$.

Lemma 2.21. $\prod_{x:\mathbf{1}} x = \star$.

Proof. We define our $C : \mathbf{1} \to \mathcal{U}$ by

$$C(x) \equiv (x = \star)$$

Now by induction we only need to prove $C(\star)$. But $C(\star) \equiv (\star = \star)$ which holds trivially since we have $\mathrm{refl}_{\star}$ for that. $\square$

2.7.3 The type of booleans

The type of booleans $\mathbf{2}$ is intended to have exactly two elements. Therefore the introduction rule states that there are two elements $0_{\mathbf{2}}, 1_{\mathbf{2}} : \mathbf{2}$. For a type $C : \mathcal{U}$ with two terms $c_0, c_1 : C$ we can define a function $f : \mathbf{2} \to C$ by setting

$$\begin{aligned} f(0_{\mathbf{2}}) &\equiv c_0 \\ f(1_{\mathbf{2}}) &\equiv c_1. \end{aligned}$$

This fact can again be rephrased into the recursor function for $\mathbf{2}$:

$$\mathrm{rec}_{\mathbf{2}} : \prod_{C:\mathbf{U}} (C \to C \to \mathbf{2} \to C)$$

with defining equations

$$\begin{aligned} \mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) &\equiv c_0 \\ \mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) &\equiv c_1. \end{aligned}$$

The induction principle for the type of booleans states that given a $C : \mathbf{2} \to \mathcal{U}$ and terms $c_0 : C(0_{\mathbf{2}})$, $c_1 : C(1_{\mathbf{2}})$ we can define a dependet function $F : \prod_{x:\mathbf{2}} C(x)$ by setting

$$\begin{aligned} F(0_{\mathbf{2}}) &\equiv c_0 \\ F(1_{\mathbf{2}}) &\equiv c_1. \end{aligned}$$

Again we can rewrite this into a function:

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

with defining equations

$$\begin{aligned} \mathrm{ind}_{\mathbf{2}}(C, c_0, c_1, 0_{\mathbf{2}}) &\equiv c_0 \\ \mathrm{ind}_{\mathbf{2}}(C, c_0, c_1, 1_{\mathbf{2}}) &\equiv c_1. \end{aligned}$$

In other words to prove a property for all elements of $\mathbf{2}$, it suffices to prove it for $0_{\mathbf{2}}$ and $1_{\mathbf{2}}$. We expect that $0_{\mathbf{2}}$ and $1_{\mathbf{2}}$ are not equal. Indeed the following lemma holds:

Lemma 2.22. *There is a function $f : (0_{\mathbf{2}} =_{\mathbf{2}} 1_{\mathbf{2}}) \to \mathbf{0}$.*

Proof. By recursion on 2 we can define a function $g : \mathbf{2} \to \mathcal{U}$ such that

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

This means we can use $\mathrm{ap}_g : (0_{\mathbf{2}} = 1_{\mathbf{2}}) \to (\mathbf{0} = \mathbf{1})$ to get an identity between $\mathbf{1}$ and $\mathbf{0}$. But now we can use idtoeqv to get an equivalence $(\mathbf{0} \simeq \mathbf{1})$. This equivalence "contains" a function $h : \mathbf{1} \to \mathbf{0}$. Now we define a function $H : \prod_{x,y:\mathbf{2}} (x =_{\mathbf{2}} y) \to \mathbf{1}$ by

$$H(x, x, \mathrm{refl}_x) \equiv \star.$$

This means $H(0_{\mathbf{2}}, 1_{\mathbf{2}}) : (0_{\mathbf{2}} = 1_{\mathbf{2}}) \to \mathbf{1}$. So in conclusion we have

$$(0_{\mathbf{2}} = 1_{\mathbf{2}}) \xrightarrow{H(0_{\mathbf{2}},1_{\mathbf{2}})} \mathbf{1} \xrightarrow{h} \mathbf{0}.$$

□

Remark 2.23. One has to be careful when using recursion on $\mathbf{2}$ in the beginning to define g. A priori we only can define a function from $\mathbf{2} : \mathcal{U}$ into $\mathcal{U}'$ by recursion, if $\mathcal{U}' : \mathcal{U}$. First of all this means that the recursion principle of $\mathbf{2} : \mathcal{U}$, and therefore the type itself, depends on the $\mathcal{U}$ we fixed when we defined $\mathbf{2}$. And it also means that our proof does not work since $\mathcal{U}$ is not a term in itself.

One solution is to use a stronger recursion principle for all our inductive types. This means that for any inductive type $A : \mathcal{U}$ we can use recursion to define a function from A into any other type $B : \mathcal{U}'$, where $\mathcal{U}'$ can be any universe. In the case of $\mathbf{2}$ this would mean that we can define a function f from $\mathbf{2} : \mathcal{U}$ into any other type $B : \mathcal{U}'$ with two points $b_0, b_1 : B$, where $\mathcal{U}'$ can be any universe, such that

$$\begin{aligned} f(0_{\mathbf{2}}) &\equiv b_0 \\ f(1_{\mathbf{2}}) &\equiv b_1. \end{aligned}$$

This fixes our proof from above.

However, one can use a weaker assumption as well. First we introduce a new notation for the different types of booleans. We will write $\mathbf{2}_i$ for the type of booleans associated with the universe $\mathcal{U}_i$. We will also apply this notation to all the other types we have defined so far if it is necessary. Now we postulate a new recursion principle. For a inductive type $A_i : \mathcal{U}_i$ we can use recursion to define a function into any other type $B : \mathcal{U}_i$ as well as to define a function into the type $A_{i+1} : \mathcal{U}_{i+1}$. Now we will give a second proof for the lemma with this notation.

Lemma 2.24. *For every i there is a function $f_i : (0_{\mathbf{2}_i} =_{\mathbf{2}_i} 1_{\mathbf{2}_i}) \to \mathbf{0}_i$.*

Proof. We fix i and use our assumption do define a function $\tilde{g} : \mathbf{2}_i \to \mathbf{2}_{i+1}$ such that

$$\begin{aligned} \tilde{g}(0_{\mathbf{2}_i}) &\equiv 0_{\mathbf{2}_{i+1}} \\ \tilde{g}(1_{\mathbf{2}_i}) &\equiv 1_{\mathbf{2}_{i+1}}. \end{aligned}$$

Now we can use the "standard" recursion on $\mathbf{2}_{i+1}$ to define a $g : \mathbf{2}_{i+1} \to \mathcal{U}_i$ such that

$$\begin{aligned} g(0_{\mathbf{2}_{i+1}}) &\equiv \ \mathbf{0}_i \\ g(1_{\mathbf{2}_{i+1}}) &\equiv \ \mathbf{1}_i. \end{aligned}$$

Therefore we can use the function $\mathrm{ap}_g : (0_{\mathbf{2}_{i+1}} = 1_{\mathbf{2}_{i+1}}) \to (\mathbf{0}_i = \mathbf{1}_i)$, to get an identity between $\mathbf{0}_i$ and $\mathbf{1}_i$. Now we can use $\mathrm{idtoeqv}_{\mathcal{U}_i}$ to get an equivalence $(\mathbf{0}_i \simeq \mathbf{1}_i)$. This equivalence contains a function $h : \mathbf{1}_i \to \mathbf{0}_i$.

Now we define a function $H : \prod_{x,y:\mathbf{2}_{i+1}} (x =_{\mathbf{2}_{i+1}} y) \to \mathbf{1}_i$ by

$$H(x, x, \mathrm{refl}_x) \equiv \star_i.$$

This means $H(0_{\mathbf{2}_{i+1}}, 1_{\mathbf{2}_{i+1}}) : (\mathbf{0}_{i+1} = \mathbf{1}_{i+1}) \to \mathbf{1}_i$. So in conclusion we have

$$(\mathbf{0}_i = \mathbf{1}_i) \xrightarrow{\mathrm{ap}_{\tilde{g}}} (\mathbf{0}_{i+1} = \mathbf{1}_{i+1}) \xrightarrow{H(0_{\mathbf{2}_{i+1}}, 1_{\mathbf{2}_{i+1}})} \mathbf{1}_i \xrightarrow{h} \mathbf{0}_i.$$

□

As one can see this notation can be quite confusing. Therefore we will simply assume that we can use recursion on an inductive type A to define a function into any other type B.

2.7.4 The integers

The introduction rule on $\mathbb{Z}$ says that there is

- a point $0 : \mathbb{Z}$
- a function $\mathrm{pos} : \mathbb{N} \to \mathbb{Z}$
- a function $\mathrm{neg} : \mathbb{N} \to \mathbb{Z}$

The recursion principle for $\mathbb{Z}$ is that for every $C : \mathcal{U}$ with

- $c_0 : C$
- $c_{\mathrm{pos}} : \mathbb{N} \to C$
- $c_{\mathrm{neg}} : \mathbb{N} \to C$

we get a function $f : \mathbb{Z} \to C$, such that $f(0_{\mathbb{Z}}) \equiv c_0$, $f(\mathrm{pos}(n)) \equiv c_{\mathrm{pos}}(n)$ and $f(\mathrm{neg}(n)) \equiv c_{\mathrm{neg}}(n)$.

In the more general case where $C : \mathbb{Z} \to \mathcal{U}$ is a type family we get the the induction principle for $\mathbb{Z}$. If we have

- $c_0 : C(0)$
- $c_{\mathrm{pos}} : \prod_{(n:\mathbb{N})} C(\mathrm{pos}(n))$
- $c_{\mathrm{neg}} : \prod_{(n:\mathbb{N})} C(\mathrm{neg}(n))$

there is a function $F : \prod_{(z:\mathbb{Z})} C(z)$, such that $F(0_{\mathbb{Z}}) \equiv c_0$, $F(\mathrm{pos}(n)) \equiv c_{\mathrm{pos}}(n)$ and $F(\mathrm{neg}(n)) \equiv c_{\mathrm{neg}}(n)$.

Note that in this definition of $\mathbb{Z}$ one should think of pos(0) as 1 and of neg(0) as -1.

One expects that there is a successor function on $\mathbb{Z}$ and that furthermore this function is an equivalence.

Lemma 2.25. *There is an equivalence* $\mathrm{succ}_{\mathbb{Z}} : \mathbb{Z} \simeq \mathbb{Z}$.

Proof. We use induction for $\mathbb{Z}$ and define:

- $\mathrm{succ}_{\mathbb{Z}}(0_{\mathbb{Z}}) \equiv \mathrm{pos}(0_{\mathbb{N}})$

- $\text{succ}_{\mathbb{Z}}(\text{pos}(n)) \equiv \text{pos}(\text{succ}_{\mathbb{N}}(n))$
- $\text{succ}_{\mathbb{Z}}(\text{neg}(n)) \equiv u(n)$.

Where $u : \mathbb{N} \to \mathbb{Z}$ is defined by recursion $\mathbb{N}$ by

- $u(0_{\mathbb{N}}) \equiv 0_{\mathbb{Z}}$
- $u(\text{succ}_{\mathbb{N}}(n)) \equiv \text{neg}(n)$

Similary we define $\text{succ}_{\mathbb{Z}}^{-1}$ by

- $\text{succ}_{\mathbb{Z}}^{-1}(0_{\mathbb{Z}}) \equiv \text{neg}(0_{\mathbb{N}})$
- $\text{succ}_{\mathbb{Z}}^{-1}(\text{pos}(n))) \equiv v(n)$
- $\text{succ}_{\mathbb{Z}}^{-1}(\text{neg}(n)) \equiv \text{neg}(\text{succ}_{\mathbb{N}}(n))$.

Where $v : \mathbb{N} \to \mathbb{Z}$ is defined by recursion $\mathbb{N}$ by

- $v(0_{\mathbb{N}}) \equiv 0_{\mathbb{Z}}$
- $v(\text{succ}_{\mathbb{N}}(n)) \equiv \text{pos}(n)$

We now prove $(\text{succ}_{\mathbb{Z}} \cdot \text{succ}_{\mathbb{Z}}^{-1})(z) = z$ for all $z : \mathbb{Z}$

By induction on $\mathbb{Z}$ we can take cases:

1. If $z \equiv 0_{\mathbb{Z}}$, we have

$$\text{succ}_{\mathbb{Z}} \cdot \text{succ}_{\mathbb{Z}}^{-1}(0_{\mathbb{Z}}) \equiv \text{succ}_{\mathbb{Z}}(\text{neg}(0_{\mathbb{N}})) \equiv u(0_{\mathbb{N}}) \equiv 0_{\mathbb{Z}}$$

2. If $z \equiv \text{pos}(n)$ for some $n : \mathbb{N}$, we prove $\text{succ} \cdot \text{succ}_{\mathbb{Z}}^{-1}(\text{pos}_{\mathbb{Z}}(n)) = \text{pos}(n)$ by induction on $\mathbb{N}$.

   If $n \equiv 0_{\mathbb{N}}$, we get:

$$\text{succ}_{\mathbb{Z}} \cdot \text{succ}_{\mathbb{Z}}^{-1}(\text{pos}(0)) \equiv \text{succ}_{\mathbb{Z}}(v(0)) \equiv \text{succ}_{\mathbb{Z}}(0_{\mathbb{Z}}) \equiv \text{pos}(0).$$

   If $n \equiv \text{succ}_{\mathbb{N}}(m)$ for some $m : \mathbb{N}$, we have:

$$\begin{aligned}\text{succ}_{\mathbb{Z}} \cdot \text{succ}_{\mathbb{Z}}^{-1}(\text{pos}(\text{succ}_{\mathbb{N}}(m))) &\equiv \text{succ}_{\mathbb{Z}}(v(\text{succ}_{\mathbb{N}}(m))) \\ &\equiv \text{succ}_{\mathbb{Z}}(\text{pos}(m)) \equiv \text{pos}(\text{succ}_{\mathbb{N}}(m))\end{aligned}$$

3. If $z \equiv \text{neg}(n)$ for some $n : \mathbb{N}$, we again use induction on $\mathbb{N}$ to prove $\text{succ}_{\mathbb{Z}} \cdot \text{succ}_{\mathbb{Z}}^{-1}(\text{neg}(n)) = \text{pos}(n)$.

   If $n \equiv 0_{\mathbb{N}}$, we get:

$$\text{succ}_{\mathbb{Z}} \cdot \text{succ}_{\mathbb{Z}}^{-1}(\text{neg}(0_{\mathbb{N}})) \equiv \text{succ}_{\mathbb{Z}}(\text{neg}(\text{succ}_{\mathbb{N}}(0_{\mathbb{N}}))) \equiv u(\text{succ}_{\mathbb{N}}(0_{\mathbb{N}})) \equiv \text{neg}(0_{\mathbb{N}})$$

   If $n \equiv \text{succ}_{\mathbb{N}}(m)$ for some $m : \mathbb{N}$, we have:

$$\begin{aligned}\text{succ}_{\mathbb{Z}} \cdot \text{succ}_{\mathbb{Z}}^{-1}(\text{neg}(\text{succ}_{\mathbb{N}}(m))) &\equiv \text{succ}_{\mathbb{Z}}(\text{neg}(\text{succ}_{\mathbb{N}}(\text{succ}_{\mathbb{N}}(m)))) \\ &\equiv u(\text{succ}_{\mathbb{N}}(\text{succ}_{\mathbb{N}}(m))) \equiv \text{neg}(\text{succ}_{\mathbb{N}}(m))\end{aligned}$$

The prove that $(\mathrm{succ}_{\mathbb{Z}}^{-1} \cdot \mathrm{succ}_{\mathbb{Z}})(z) = z$ for all $z : \mathbb{Z}$ works similar and is therefore omitted. ☐

Next we are going to prove that, when working with the integers, propositional equality and judgmental equality are equivalent. This means that if and only if $x \equiv y$ we have $x = y$ and even further that all the paths $x = y$ are equal as elements in the identity type.

To see this, we will use the so called encode-decode method. First we define a type $\mathrm{code} : \mathbb{Z} \to \mathbb{Z} \to \mathcal{U}$ and then we will show that for every $x, y : \mathbb{Z}$ we have $(x =_{\mathbb{Z}} y) \simeq \mathrm{code}(x, y)$.

Definition 2.26. We define $\mathrm{code} : \mathbb{Z} \to \mathbb{Z} \to \mathcal{U}$ by

$$\begin{aligned}
\mathrm{code}(0_{\mathbb{Z}}, 0_{\mathbb{Z}}) &\equiv \mathbf{1}\\
\mathrm{code}(\mathrm{pos}(0_{\mathbb{N}}), \mathrm{pos}(0)_{\mathbb{N}}) &\equiv \mathbf{1}\\
\mathrm{code}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m)) &\equiv \mathrm{code}(\mathrm{pos}(n), \mathrm{pos}(m))\\
\mathrm{code}(\mathrm{neg}(0_{\mathbb{N}}), \mathrm{neg}(0)_{\mathbb{N}}) &\equiv \mathbf{1}\\
\mathrm{code}(\mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(n), \mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(m)) &\equiv \mathrm{code}(\mathrm{neg}(n), \mathrm{neg}(m))\\
\mathrm{code}(\mathrm{pos}(n), 0_{\mathbb{Z}}) &\equiv \mathbf{0} \equiv \mathrm{code}(0_{\mathbb{Z}}, \mathrm{pos}(m))\\
\mathrm{code}(\mathrm{neg}(n), 0_{\mathbb{Z}}) &\equiv \mathbf{0} \equiv \mathrm{code}(0_{\mathbb{Z}}, \mathrm{neg}(m))\\
\mathrm{code}(\mathrm{pos}(\mathrm{succ}(n)), \mathrm{pos}(0_{\mathbb{Z}})) &\equiv \mathbf{0} \equiv \mathrm{code}(\mathrm{pos}(0_{\mathbb{Z}}), \mathrm{pos}(\mathrm{succ}(m)))\\
\mathrm{code}(\mathrm{neg}(\mathrm{succ}(n)), \mathrm{neg}(0_{\mathbb{Z}})) &\equiv \mathbf{0} \equiv \mathrm{code}(\mathrm{neg}(0_{\mathbb{Z}}), \mathrm{neg}(\mathrm{succ}(m)))\\
\mathrm{code}(\mathrm{pos}(n), \mathrm{neg}(m)) &\equiv \mathbf{0} \equiv \mathrm{code}(\mathrm{pos}(n), \mathrm{pos}(m)).
\end{aligned}$$

Remark 2.27. Here we need the "strong" recursion mentioned in remark 2.23.

Lemma 2.28. *For all $x, y : \mathbb{Z}$, $p : x = y$ we have*

$$p_*^{z \mapsto \mathrm{code}(x,z)}(-) = p_*^{z \mapsto \mathrm{code}(\mathrm{succ}_{\mathbb{Z}}(x), \mathrm{succ}_{\mathbb{Z}}(z))}(-).$$

Proof. By path induction it is sufficient to prove that for all $x : \mathbb{Z}$

$$(\mathrm{refl}_x)_*^{z \mapsto \mathrm{code}(x,z)}(-) = (\mathrm{refl}_x)_*^{z \mapsto \mathrm{code}(\mathrm{succ}_{\mathbb{Z}}(x), \mathrm{succ}_{\mathbb{Z}}(z))}(-)$$

holds. By definition this is just to prove that

$$\mathrm{id}_{\mathrm{code}(x,x)} = \mathrm{id}_{\mathrm{code}(\mathrm{succ}_{\mathbb{Z}}(x), \mathrm{succ}_{\mathbb{Z}}(x))}$$

holds for all $x : \mathbb{Z}$. Now by induction on $\mathbb{Z}$ we can take cases:

1. If $x \equiv 0_{\mathbb{Z}}$ by definition of code, we have

   $$\begin{aligned}
   &\mathrm{code}(0_{\mathbb{Z}}, 0_{\mathbb{Z}}) \equiv \mathbf{1} \quad \text{and,}\\
   &\mathrm{code}(\mathrm{succ}_{\mathbb{Z}}(0_{\mathbb{Z}}), \mathrm{succ}_{\mathbb{Z}}(0_{\mathbb{Z}})) \equiv \mathrm{code}(\mathrm{pos}(0_{\mathbb{N}}), \mathrm{pos}(0_{\mathbb{N}})) \equiv \mathbf{1}.
   \end{aligned}$$

   So we need to prove $\mathrm{id}_{\mathbf{1}} = \mathrm{id}_{\mathbf{1}}$ which is trivially true.

2. If $x \equiv \mathrm{pos}(n)$ for some $n : \mathbb{N}$, we use induction on $\mathbb{N}$.

   If $n \equiv 0_{\mathbb{N}}$, we get

$$\mathrm{code}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(0_{\mathbb{N}})), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(0_{\mathbb{N}}))) \equiv \mathrm{code}(\mathrm{pos}(0_{\mathbb{N}})), \mathrm{pos}(0_{\mathbb{N}})) \equiv \mathbf{1}$$

   so again we only have to prove $\mathrm{id}_{\mathbf{1}} = \mathrm{id}_{\mathbf{1}}$ which is trivially true.

3. If $n \equiv \mathrm{succ}_{\mathbb{N}}(m)$ for some $m : \mathbb{N}$ by definition of code, we get

$$\mathrm{code}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))) \equiv \mathrm{code}(\mathrm{pos}(n), \mathrm{pos}(n)).$$

   So we only have to prove $\mathrm{id}_{\mathrm{code}(\mathrm{pos}(n),\mathrm{pos}(n))} = \mathrm{id}_{\mathrm{code}(\mathrm{pos}(n),\mathrm{pos}(n))}$ which is obviously true.

4. If $z \equiv \mathrm{neg}(n)$, the prove is similar to the 2. and 3. case.

□

Theorem 2.29. *There is a function* $\prod_{x,y:\mathbb{Z}} ((x =_{\mathbb{Z}} y) \simeq \mathrm{code}(x, y))$.

Proof. Combining the two lemmata 2.32 and 2.33 below. □

Definition 2.30. We define a function encode : $\prod_{x,y:\mathbb{Z}}(x =_{\mathbb{Z}} y) \to \mathrm{code}(x, y)$ by

$$\mathrm{encode}(x, y, p) \equiv \mathrm{transport}^{\mathrm{code}(x,-)}(p, k(x)),$$

where $k : \prod_{z:\mathbb{Z}} \mathrm{code}(z, z)$ is defined by recursion on $\mathbb{Z}$ by

$$\begin{aligned} k(0_{\mathbb{Z}}) \equiv k(\mathrm{pos}(0_{\mathbb{N}})) \equiv k(\mathrm{neg}(0_{\mathbb{N}})) &\equiv \star \\ k(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))) &\equiv k(\mathrm{pos}(n)) \\ k(\mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(n))) &\equiv k(\mathrm{neg}(n)) \end{aligned}$$

Definition 2.31. We define a function decode : $\prod_{x,y:\mathbb{Z}} \mathrm{code}(x, y) \to (x =_{\mathbb{Z}} y)$ by double induction on $\mathbb{Z}$. For every pair of integers $x, y : \mathbb{Z}$ we need to give a function $\mathrm{code}(x, y) \to (x =_{\mathbb{Z}} y)$.

1. If $x \equiv y \equiv 0_{\mathbb{Z}}$, we need to give a function $\mathbf{1} \to (0_{\mathbb{Z}} = 0_{\mathbb{Z}})$. Such a function is easily defined by induction on $\mathbf{1}$ by $\mathrm{decode}(0_{\mathbb{Z}}, 0_{\mathbb{Z}})(\star) \equiv \mathrm{refl}_0$.
2. If $x \equiv y \equiv \mathrm{pos}(0_{\mathbb{N}})$, we define decode by $\mathrm{decode}(\mathrm{pos}(0_{\mathbb{N}}), \mathrm{pos}(0_{\mathbb{N}}))(\star) \equiv \mathrm{refl}_{\mathrm{pos}(0)}$.
3. If $x \equiv y \equiv \mathrm{neg}(0_{\mathbb{N}})$, we define decode by $\mathrm{decode}(\mathrm{neg}(0_{\mathbb{N}}), \mathrm{neg}(0_{\mathbb{N}}))(\star) \equiv \mathrm{refl}_{\mathrm{neg}(0)}$.
4. If $x \equiv \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))$, $y \equiv \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m))$ and we already have defined

$$\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m)) : \mathrm{code}(\mathrm{pos}(n), \mathrm{pos}(m)) \to (\mathrm{pos}(n) = \mathrm{pos}(m))$$

   we define decode by

$$\mathrm{decode}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m))) \equiv \mathrm{ap}_{\mathrm{succ}_{\mathbb{Z}}} \cdot \mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m)).$$

Since by definition we have

$$\mathrm{code}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m))) \equiv \mathrm{code}(\mathrm{pos}(n), \mathrm{pos}(m)),$$

this is well typed.

5. If $x \equiv \mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(n))$, $y \equiv \mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(m))$ and we already have defined

$$\mathrm{decode}(\mathrm{neg}(n), \mathrm{neg}(m)) : \mathrm{code}(\mathrm{neg}(n), \mathrm{neg}(m)) \to (\mathrm{neg}(n) = \mathrm{neg}(m))$$

we define decode by

$$\mathrm{decode}(\mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(m))) \equiv \mathrm{ap}_{\mathrm{succ}_{\mathbb{Z}}^{-1}} \centerdot \mathrm{decode}(\mathrm{neg}(n), \mathrm{neg}(m)).$$

6. In all the other cases $\mathrm{code}(x, y) \equiv \mathbf{0}$ so we can use induction on $\mathbf{0}$ to get a function $\mathrm{code}(x, y) \to (x = y)$.

Lemma 2.32. *For every $x, y : \mathbb{Z}$ and every path $p : x = y$ we have*

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

Proof. By path induction it suffices to prove

$$\mathrm{decode}(x, x, \mathrm{encode}(x, x, \mathrm{refl}_x)) = \mathrm{refl}_x.$$

By definition of encode and decode we get

$$\begin{aligned}\mathrm{decode}(x, x, \mathrm{encode}(x, x, \mathrm{refl}_x)) &\equiv \mathrm{decode}(x, x, \mathrm{transport}^{\mathrm{code}(x,-)}(\mathrm{refl}_x, k(x))) \\ &\equiv \mathrm{decode}(x, x, k(x)).\end{aligned}$$

So it remains to prove that $\mathrm{decode}(x, x, k(x)) = \mathrm{refl}_x$ for all $x : \mathbb{Z}$. This is clear by definition and induction over $\mathbb{Z}$. □

Lemma 2.33. *For every $x, y : \mathbb{Z}$ and $c : \mathrm{code}(x, y)$ we have*

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

Proof. We use double induction on $\mathbb{Z}$

1. If $x \equiv y \equiv 0_{\mathbb{Z}}$ or $x \equiv y \equiv \mathrm{pos}(0_{\mathbb{N}})$ or $x \equiv y \equiv \mathrm{neg}(0_{\mathbb{N}})$ we know that by definition $\mathrm{code}(x, x) \equiv \mathbf{1}$ and that $\mathrm{decode}(x, x)(\star) \equiv \mathrm{refl}_x$. We now want to prove that for every $c : \mathrm{code}(x, x)$

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

but by induction on $\mathbf{1}$ we only need to prove $\mathrm{encode}(x, x, \mathrm{decode}(x, x, \star)) = \star$. By definition we have

$$\begin{aligned}\mathrm{encode}(x, x, \mathrm{decode}(x, x, \star)) &\equiv \mathrm{encode}(x, x, \mathrm{refl}_x) \\ &\equiv \mathrm{transport}^{\mathrm{code}(x,-)}(\mathrm{refl}_x, k(x)) \\ &\equiv \mathrm{transport}^{\mathrm{code}(x,-)}(\mathrm{refl}_x, \star) \\ &\equiv \star\end{aligned}$$

2. Assume $x \equiv \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))$, $y \equiv \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m))$ and

$$\mathrm{encode}(\mathrm{pos}(n), \mathrm{pos}(m), \mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c)) = c$$

for all $c : \mathrm{code}(\mathrm{pos}(n), \mathrm{pos}(m))$. We need to prove

$$\mathrm{encode}\Big(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m)), \mathrm{decode}\Big(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m)), c\Big)\Big)$$

is equal to c. By definition of decode and encode we have

$$\begin{aligned}
&\mathrm{encode}\Big(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m)), \mathrm{decode}\Big(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m)), c\Big)\Big) \\
&\equiv \mathrm{encode}\Big(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)), \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(m)), \mathrm{ap}_{\mathrm{succ}_{\mathbb{Z}}}(\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c))\Big) \\
&\equiv \Big(\mathrm{ap}_{\mathrm{succ}_{\mathbb{Z}}}(\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c))\Big)_*^{\mathrm{code}((\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)),-)}(k(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)))).
\end{aligned}$$

Next we use the definition of k and lemma 2.15, followed by the definition of $\mathrm{succ}_{\mathbb{Z}}$ to see that

$$\begin{aligned}
&\Big(\mathrm{ap}_{\mathrm{succ}_{\mathbb{Z}}}(\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c))\Big)_*^{\mathrm{code}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)),-)}(k(\mathrm{pos}(n))) \\
&= (\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c))_*^{\mathrm{code}((\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)),\mathrm{succ}_{\mathbb{Z}}(-))}(k(\mathrm{pos}(n))) \\
&= (\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c))_*^{\mathrm{code}(\mathrm{succ}_{\mathbb{Z}}(\mathrm{pos}(n)),\mathrm{succ}_{\mathbb{Z}}(-))}(k(\mathrm{pos}(n))).
\end{aligned}$$

With lemma 2.28 we can rewrite this, to see that

$$\begin{aligned}
&(\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c))_*^{\mathrm{code}(\mathrm{succ}_{\mathbb{Z}}(\mathrm{pos}(n)),\mathrm{succ}_{\mathbb{Z}}(-))}(k(\mathrm{pos}(n))) \\
&= (\mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c))_*^{\mathrm{code}(\mathrm{pos}(n),-)}(k(\mathrm{pos}(n))) \\
&= \mathrm{encode}(\mathrm{pos}(n), \mathrm{pos}(m), \mathrm{decode}(\mathrm{pos}(n), \mathrm{pos}(m), c)) \\
&= c,
\end{aligned}$$

where we used the induction assumption in the last step.

3. If $x \equiv \mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(n))$ and $y \equiv \mathrm{neg}(\mathrm{succ}_{\mathbb{N}}(m))$ the proof works the same as the one we just did.

4. In all the other cases we get that $\mathrm{code}(x, y) \equiv \mathbf{0}$. So we need to prove something for all $c : \mathbf{0}$ which is trivial by induction on $\mathbf{0}$.

$\square$

This means that

Lemma 2.34. *For all $x, y : \mathbb{Z}$ and paths $p, q : x = y$ we have $p =_{(x=y)} q$.*

Remark 2.35. One might try to use path induction to prove this. However, after using path induction for p, one has to prove that for all $x : \mathbb{Z}$ and all paths $q : x = x$ one has $\mathrm{refl}_x = q$. At this point one can **not** use path induction a second time, because the statement one wants to prove is not of the form for all $x, y : \mathbb{Z}$ and $q : x = y$ we have some proposition $C(x, y, q)$.

Proof. For all $x, y : \mathbb{Z}$ we have an equivalence between $x = y$ and $\mathrm{code}(x, y)$. By induction on $\mathbb{Z}$ there are only two cases we have to consider:

1. $\mathrm{code}(x, y) \equiv \mathbf{0}$: We name our quasi-inverses $f : (x = y) \to \mathrm{code}(x, y)$ and f^{-1}. We have

$$p = f^{-1}(f(p)) = f^{-1}(f(q)) = q,$$

where we used that f and f^{-1} are quasi-inverse and that for all elements $a, b : \mathbf{0}$ we have $a = b$.

2. $\mathrm{code}(x, y) \equiv \mathbf{1}$: We name our functions $f : (x = y) \to \mathrm{code}(x, y)$ and f^{-1} and get

$$p = f^{-1}(f(p)) = f^{-1}(f(q)) = q,$$

where we used that f and f^{-1} are quasi-inverse and that, by lemma 2.21, for all elements $a, b : \mathbf{1}$ we have $a = b$.

$\square$

2.8 Univalence axiom

In chapter 2.6.3 we talked about identies between types and were able to prove that there is a function $\mathrm{idtoeqv} : (A =_{\mathcal{U}} B) \to B) \to (A \simeq B)$. We expect that there is also a function $(A \simeq B) \to (A =_{\mathcal{U}} B)$ which is quasi-inverse to idtoeqv. However type theory is not able to guarantee the existence of such a function. Therefore we have to take this property as an axiom:

Axiom 2.36. (Univalence axiom) For any $A, B : \mathcal{U}$ idtoeqv is an equivalence.

Definition 2.37. We call the quasi-inverse of idtoeqv $\mathrm{ua} : (A \simeq B) \to (A =_{\mathcal{U}} B)$.

3 Higher inductive Types

3.1 The higher interval

In this chapter we study higher inductive types (for further reading see [6]). When we say a type is inductively defined, we mean that this type is generated by some constructors. Just like the natural numbers, which are generated by the 0 element together with the succsessor function. In all the examples we have seen so far the constructor only generated objects of said type and nothing more. But what, if we want to define something like the interval I. The structure of the interval is more than just an accumulation of points. One can define the higher interval as the type with two points and a path connecting those points. This means our introduction rule is given by

- a point $0_I : I$
- a point $1_I : I$
- a path $\mathrm{seg} : 0_I =_I 1_I$.

In this inductive definition we generate not only two points $0_I : I$ and $1_I : I$, but also a path in the type we are defining. We will call ordinary contructors (like the one that constructs $0_I : I$) point constructors, and the others (like seg) path constructors. When the definition of a type uses one (or more) path constructors, it is considered a higher inductive type.

Note that, regardless of the structure of a type, there are some operations like inversion that act on paths and higher paths on that type. Therefore, if a path is generated by a certain constructor, these operations on paths yield more paths that may not come from the constructor itself. In the example mentioned above we would not only have seg but also seg^{-1}.

One could think that a higher inductive type actually defines more than one type. For example, one could think that our definition of I not only defines I itself but also the identity type over I. On the first look that idea makes some sense, since our definition contains a path constructor. However, it is the higher inductive type itself which will be equipped with some induction principle (and therefore a universal property) and not the

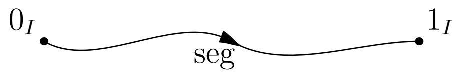

Figure 3.1: The higher interval

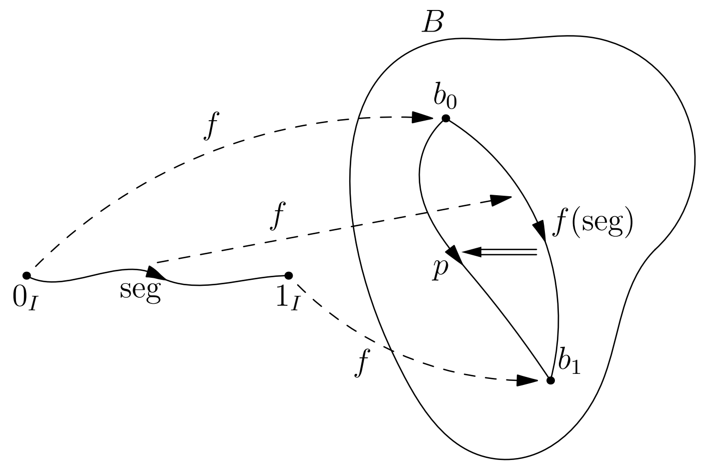

Figure 3.2: The recursion principle of the higher interval

identity type as one may think. In particular the identity type over a higher inductive type still has the same properties as any other identity type (and not any more).

Note that there are some inconspicuous constructors that yield paths or even higher paths (i.e. paths between paths). If we, for example, define a type B with a constructor of the type $A \to B$, every path in A generates a path in B and every higher path in A generates a higher path in B. Therefore, with a suitable A, this simple constructor generates higher paths in B, making this constructor a path constructor.

The recursion principle for a type A generally states that, if we have another type with the same structure, there is a function from A into that type that respects that structure. This idea can be extended to higher inductive types. In our example I the recursion principle would look like this. Given any type B with two points $b_0, b_1 : B$ and a path $p : b_0 =_B b_1$.,there is a function $f : I \to B$ such that $f(0_I) \equiv b_0$, $f(1_I) \equiv b_1$ and $\mathrm{ap}_f(\mathrm{seg}) = p$.

These three equalities are the computation rules. Note that, while we say that $f(0_I)$ is by definition the same as b_0, we used the weaker propositional equality the last time, i.e. we only say that there is a path between $\mathrm{ap}_f(\mathrm{seg})$ and p. But why do we not use judgemental equalities both times? The problem is that the application function is no fundamental part of type theory, but something we defined. It seems a bit weird to call two things judgementally equal, if everything depends on how we define a certain function. Jugemental equalities are part of the deductive system and should therefore not depend on choices that we make within that system. While these questions are part of current research, we will not dive deeper into this topic and simply accept that the computation rules for point constructors yield judgemental equalities and the computation rules for path constructors only yield propositional equalities.

As with ordinary inductive types we now want to give an induction principle for I, i.e. a dependent eliminator. Recall that the induction principle for a type A states that to prove some given proposition P for all $x : A$ we only need to consider the elements directly generated by the constructors over A. For a better understanding first recall the induction principle for $\mathbb{N}$. To prove a proposition P for all $n : \mathbb{N}$ we need to give:

- a prove of $P(0)$, by giving an element $\mathrm{b} : P(0)$
- for each $n : \mathbb{N}$ a prove of that $P(n)$ implies $P(\mathrm{succ}(n))$, by giving a function $P(n) \to P(\mathrm{succ}(n))$.

The second can be seen as a function $P \to P$ respecting the constructor $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$ or (topologically) lying over the succesor function. Analogously to prove P for all $x : I$ we need to give:

- a prove of $P(0_I)$ and $P(1_I)$ by giving elements $b_0 : P(0_I)$ and $b_1 : P(1_I)$
- a path from b_0 to b_1 respecting (or lying over) the constructor $\mathrm{seg} : 0_I = 1_I$.

But what do we mean by a path lying over another path? And how can we even connect two elemnts b_0 and b_1 with a path, if they dont even have to be of the same type?

We already have discussed that topic in Lemma 2.8 about the dependent application function. The solution was that a path from $u : P(x)$ to $v : P(y)$ lying over $p : (x = y)$ can be interpreted as a path from $p_*(u)$ to v in $P(y)$. We call such paths, lying over other paths, dependent paths and introduce a short notation for the type of dependent paths by defining

$$(u =_p^P v) \equiv (\mathrm{transport}^P(p, u) = v) \equiv p_*^P(u) = v$$

Now we can define the induction principle for I, which states that given $P : I \to \mathcal{U}$ together with

- a point $b_0 : P(0)$
- a point $b_1 : P(1)$
- a path $p : b_0 =_{\mathrm{seg}}^P b_1$.

there is a function $F : \prod_{(x:I)} P(x)$ such that $F(0_I) \equiv b_0$, $F(1_I) \equiv b_1$ and $\mathrm{apd}_F(\mathrm{seg}) = p$.

With the induction principle we can prove the recursion principle.

Lemma 3.1. *(Recursion principle) Given a type B with*

- *a point $b_0 : B$*
- *a point $b_1 : B$*
- *a path $p : b_0 =_B b_1$.*

there is a function $f : I \to B$ such that $f(0_I) \equiv b_0$, $f(1_I) \equiv b_1$ and $\mathrm{ap}_f(\mathrm{seg}) = p$.

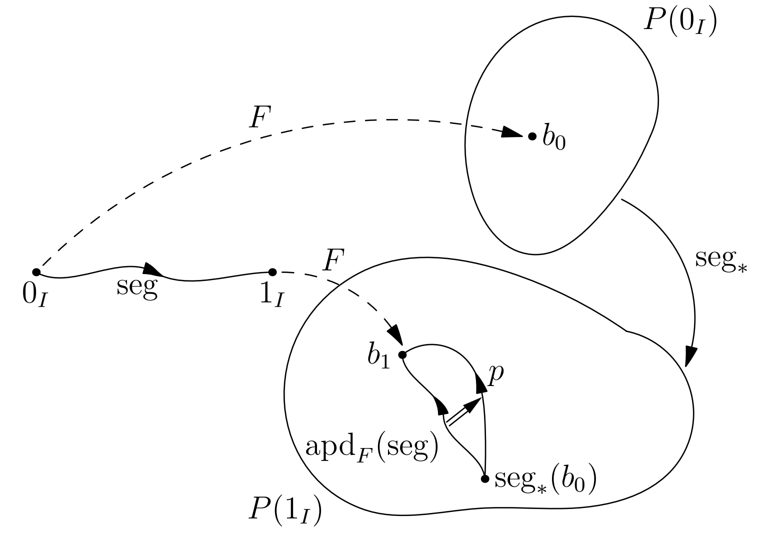

Figure 3.3: The induction principle for the higher interval

Proof. We want to use the induction principle for I with $P(x) \equiv B$. We have $b_0 : P(0)$ and $b_1 : P(1)$, so it remains to give a path $l : \text{transport}^P(\text{seg}, b_0) = b_1$. Since we have $\text{transportconst}^B_{\text{seg}}(b_0) : \text{transport}^P(\text{seg}, b_0) = b_0$, we get

$$\text{transportconst}^B_{\text{seg}}(b_0) \cdot p : \text{transport}^P(\text{seg}, b_0) = b_1$$

Now the induction principle yields a function $F : \prod_{(x:I)} P(x)$ such that $F(0_I) \equiv b_0$, $F(1_I) \equiv b_1$ and $\text{apd}_F(\text{seg}) = \text{transportconst}^P_{\text{seg}}(b_0) \cdot p$. Since we can interpret F as a non dependent function $f : A \to B$ with $f(0_I) \equiv b_0$, $f(1_I) \equiv b_1$ and $\text{apd}_f(\text{seg}) = \text{transportconst}^B_{\text{seg}}(b_0) \cdot p$ it only remains to show

$$\text{ap}_f(\text{seg}) = p$$

To see this we use the connection between ap and apd given by lemma 2.10 namely

$$\text{apd}_f(\text{seg}) = \text{transportconst}^B_{\text{seg}} \cdot \text{ap}_f(p)$$

so we get that

$$\text{transportconst}^B_{\text{seg}} \cdot \text{ap}_f(p) = \text{transportconst}^P_{\text{seg}}(b_0) \cdot p.$$

By cancelling $\text{transportconst}^B_{\text{seg}}$ we get $\text{ap}_f(\text{seg}) = p$. □

Our conception of the interval is that all the points lie on the path from 0_I to 1_I. Therefore all elements in I should be connected by a path.

Lemma 3.2. *The type I is contractible, for example with centre of contraction 1_I, i.e. there is a function $F : \prod_{(x:I)}(x =_I 1_I)$.*

Proof. We use the induction principle on I and start by defining $P : I \to \mathcal{U}$ by $P(x) \equiv (x =_I 1_I)$. We already have

$$\begin{aligned} \mathrm{seg} &: P(0) \\ \mathrm{refl}_{1_I} &: P(1) \end{aligned}$$

Now we must find an element of the type $\mathrm{seg} =^{\lambda x.(x=_I 1_I)}_{\mathrm{seg}} \mathrm{refl}_{1_I}$ or in other words, we have to prove that this equality holds. By definition this is the type

$$\mathrm{transport}^{\lambda x.(x=_I 1_I)}(\mathrm{seg}, \mathrm{seg}) = \mathrm{refl}_{1_I}.$$

By lemma 2.12 we know that

$$\mathrm{transport}^{\lambda x.(x=_I 1_I)}(\mathrm{seg}, \mathrm{seg}) = \mathrm{seg}^{-1} \cdot \mathrm{seg}$$

So we only need to prove that

$$\mathrm{seg}^{-1} \cdot \mathrm{seg} = \mathrm{refl}_{1_I}.$$

This, however, holds as we have seen when discussing paths. □

We can use the interval to prove function extensionality.

Lemma 3.3. *Given two functions $f, g : A \to B$ such that for all $x : A$ we have $f(x) = g(x)$, f, then $f = g$ in the type $A \to B$*

Proof. By assumption we have a function $P : \prod_{x:A}(f(x) = g(x))$ and our goal is to find an element of $f =_{(A\to B)} g$. First for all $x : A$ we define an auxiliary function $p_x : I \to B$ by induction on I with the defining definitions

$$\begin{aligned} p_x(0_I) &\equiv f(x) \\ p_x(1_I) &\equiv g(x) \\ \mathrm{ap}_{p_x}(\mathrm{seg}) &= P(x) \end{aligned}$$

Now we can define $q : I \to (A \to B)$ by

$$\begin{aligned} q(0_I) &\equiv (\lambda x.p_x(0_I)) \\ q(1_I) &\equiv (\lambda x.p_x(1_I)) \\ \mathrm{ap}_q(\mathrm{seg}) &\equiv (\lambda x.\mathrm{ap}_{p_x}(\mathrm{seg})) \end{aligned}$$

By this definition we have that $q(0_I)$ is simply f and $q(1_I)$ is g. This means q maps seg on a path between f and g, i.e.

$$q(\mathrm{seg}) : f =_{(A\to B)} g$$

□

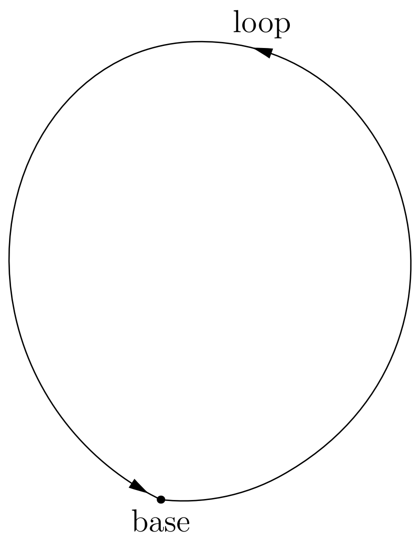

Figure 3.4: The higher circle

3.2 The higher circle

Now that we understood the basic properties of higher inductive types, we will define the higher circle $\mathbb{S}^1$. Just as we consider the interval as two points connected by a path, we can define the circle inductively, in a natural way by

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

This means we understand the circle as a basepoint called base and a path from that basepoint to itself. This of course is a homotopical representation of the circle in the sense that we do not care about distances and the like.

To define a type, we need to know how to construct and how to use objects of that type. Having discussed the introduction rule for $\mathbb{S}^1$ we now consider the elimination rules, i.e. the recursion and induction principle.

The induction principle for $\mathbb{S}^1$ states that given $P : \mathbb{S}^1 \to \mathcal{U}$ as well as

- an element $b : P(\text{base})$
- a path $l : b =^P_{\text{loop}} b$

there is a function $F : \prod_{(x:\mathbb{S}^1)} P(x)$ such that $F(\text{base}) \equiv b$ and $\text{apd}_F(\text{loop}) = l$. This means that to prove a proposition for all elements of the circle, we need to prove it for base and we have to prove that the proposition holds as x varies along the path loop.

As with the other types we expect the recursion principle to be provable from the induction principle.

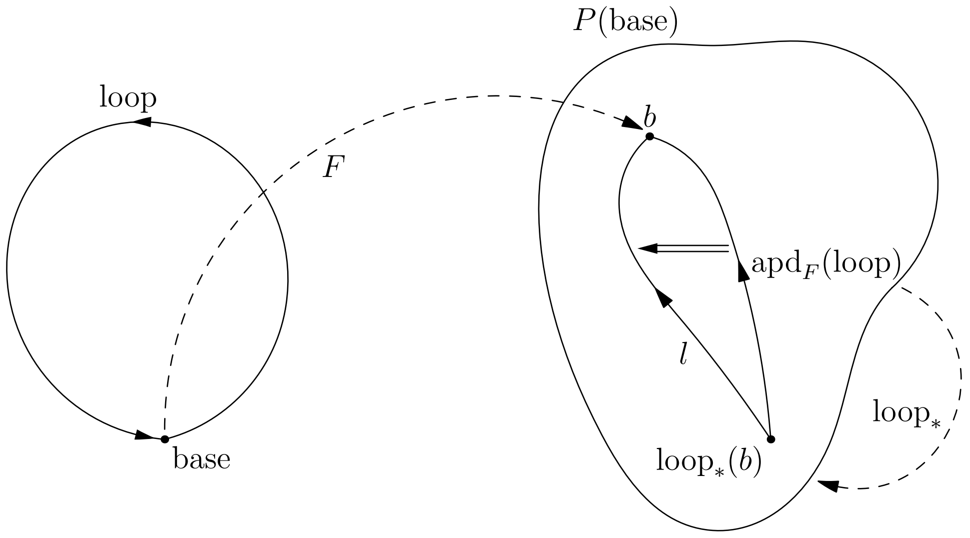

Figure 3.5: The induction principle for the higher circle

Lemma 3.4. *(Recursion principle) Given a $B : \mathcal{U}$ with a point $b : B$ and a path $l : b{=}b$, there is a function $f : \mathbb{S}^1 \to B$ such that $f(\text{base}) \equiv b$ and $\text{ap}_f(\text{loop}) = l$.*

Proof. We use the induction principle with $P : \mathbb{S}^1 \to \mathcal{U}$, where $P(x) \equiv B$. We have $b : P(\text{base})$ and need a path $b =^P_{\text{loop}} b$, which by definition is $\text{transport}^P(\text{loop}, b) = b$. From lemma 2.9 we know that we have $\text{transportconst}^B_{\text{loop}} : \text{transport}^P(\text{loop}, b) = b$. Since $l : b = b$ we can consider the composite

$$\text{transportconst}^B_{\text{loop}} \cdot l : (b =^P_{\text{loop}} b)$$

Now the induction principle on $\mathbb{S}^1$ yields a function $F : \prod_{(x:\mathbb{S}^1)} P(x)$ such that $F(\text{base}) \equiv b$ and $\text{apd}_F(\text{loop}) = \text{transportconst}^B_{\text{loop}} \cdot l$. Note that F can also be seen as a non dependent function $f : A \to B$.

It remains to show that $\text{ap}_F(\text{loop}) = l$. From lemma 2.10 we know that

$$\text{apd}_F(\text{loop}) = \text{transportconst}^B_{\text{loop}} \cdot \text{ap}_F(\text{loop})$$

so we get that

$$\text{transportconst}^B_{\text{loop}} \cdot \text{ap}_F(\text{loop}) = \text{transportconst}^B_{\text{loop}} \cdot l.$$

Now we can cancel transportconst and get $\text{ap}_F(\text{loop}) = l$. $\square$

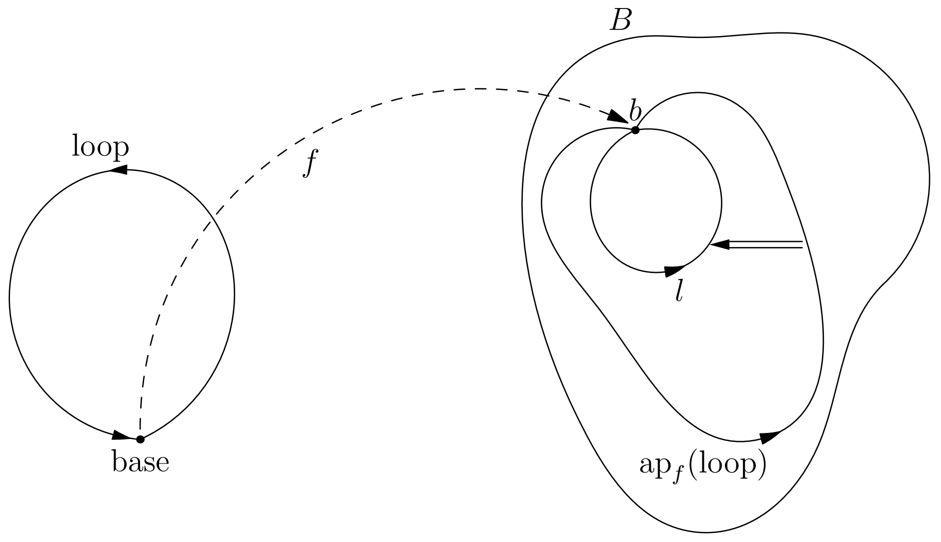

Figure 3.6: The recursion principle for the higher circle

This induction principle has a corresponding uniqueness principle, namely

Lemma 3.5. *If A is a type and $f, g : \mathbb{S}^1 \to A$ are two maps such that there are two equalities p, q*

$$p : f(\text{base}) =_A g(\text{base})$$
$$q : \mathrm{ap}_f(\text{loop}) =_p^{\lambda x.(x =_A x)} \mathrm{ap}_g(\text{loop})$$

we have $f(x) = g(x)$ for all $x : \mathbb{S}^1$.

Proof. We use the induction principle for $\mathbb{S}^1$ on the type family $P(x) \equiv (f(x) = g(x))$. First we need to give an element in $P(base)$, i.e. in $(f(\text{base}) =_A g(\text{base}))$. Here we can simply use p as it is exactly of the right type. Next we have to prove that $p =_{\text{loop}}^P p$. For this we rewrite this equality until we see that q is sufficient to prove it. For this, first note that $p =_{\text{loop}}^P p$ is by definition

$$\text{transport}^{\lambda x.f(x)=g(x)}(\text{loop}, p) = p$$

From lemma 2.11 we know that

$$\text{transport}^{\lambda x.f(x)=g(x)}(\text{loop}, p) = \left(\mathrm{ap}_f(\text{loop})\right)^{-1} \cdot p \cdot \mathrm{ap}_g(\text{loop})$$

So now it is sufficient to prove that

$$\left(\mathrm{ap}_f(\text{loop})\right)^{-1} \cdot p \cdot \mathrm{ap}_g(\text{loop}) = p$$

which can easily be rewritten into

$$\mathrm{ap}_f(\text{loop}) \cdot p = p \cdot \mathrm{ap}_g(\text{loop}).$$

Here we can apply lemma 2.18 to see that

$$\mathrm{ap}_f(\mathrm{loop}) \cdot p = p \cdot \mathrm{ap}_g(\mathrm{loop}) \simeq \mathrm{transport}^{\lambda x.(x=x)}(p, \mathrm{ap}_f(\mathrm{loop})) = \mathrm{ap}_g(\mathrm{loop}).$$

This means we have a function that maps every proof of

$$\mathrm{transport}^{\lambda x.(x=x)}(p, \mathrm{ap}_f(\mathrm{loop})) = \mathrm{ap}_g(\mathrm{loop})$$

onto a proof of

$$\mathrm{ap}_f(\mathrm{loop}) \cdot p = p \cdot \mathrm{ap}_g(\mathrm{loop}).$$

So it only remains to prove that the type $\mathrm{transport}^{\lambda x.(x=x)}(p, \mathrm{ap}_f(\mathrm{loop})) = \mathrm{ap}_g(\mathrm{loop})$ is inhabited. This type, however, is by definition $\mathrm{ap}_f(\mathrm{loop}) =_p^{\lambda x.(x=_A x)} \mathrm{ap}_g(\mathrm{loop})$ which is proven by q. ☐

With this lemma we can now prove the universal property of the circle:

Lemma 3.6. *For any type A we have*

$$(\mathbb{S}^1 \to A) \simeq \sum_{x:A}(x =_A x)$$

Proof. There is a natural choice for a function $f : (\mathbb{S}^1 \to A) \to \sum_{x:A}(x =_A x)$ namely to define

$$f(h) \equiv (h(\mathrm{base}), \mathrm{ap}_h(\mathrm{loop}))).$$

Now we need a function $g : \sum_{x:A}(x =_A x) \to (\mathbb{S}^1 \to A)$. By recursion on the dependent pair type we can define $g(x, p) \equiv h$, where $h : \mathbb{S}^1 \to A$ is defined by recursion on $\mathbb{S}^1$ by

$$h(\mathrm{base}) \equiv x$$
$$h(\mathrm{loop}) = p.$$

Next we prove that f and g are quasi-inverse.

For all $u : \sum_{x:A}(x =_A x)$ we have $f(g(u)) = u$. By induction on $\sum_{x:A}(x =_A x)$ we can assume that u has the form (x, l) with $x : A$ and $l : x = x$. By definition we have $f(g(u)) \equiv (g(u)(\mathrm{base}), g(u)(\mathrm{loop}))$. But $g(u)(\mathrm{base}) \equiv x$ and $g(u)(\mathrm{loop}) = l$ so $f(g(u)) = u$.

Now consider a function $h : \mathbb{S}^1 \to A$. We show that $g(f(h)) = h$. By the uniqueness principle of $\mathbb{S}^1$ it is sufficient to prove that there is

$$p : g(f(h))(\mathrm{base}) = h(\mathrm{base})$$
$$q : p_*^{(x \mapsto x=x)}(g(f(h))(\mathrm{loop})) = h(\mathrm{loop}).$$

First we define p. Since by definition $g(f(h))(\mathrm{base}) \equiv h(\mathrm{base})$, we can take p to be $\mathrm{refl}_{h(\mathrm{base})}$. It remains to prove that $g(f(h))(\mathrm{loop}) = h(\mathrm{loop})$ but this again simply holds by definition of f and g. ☐

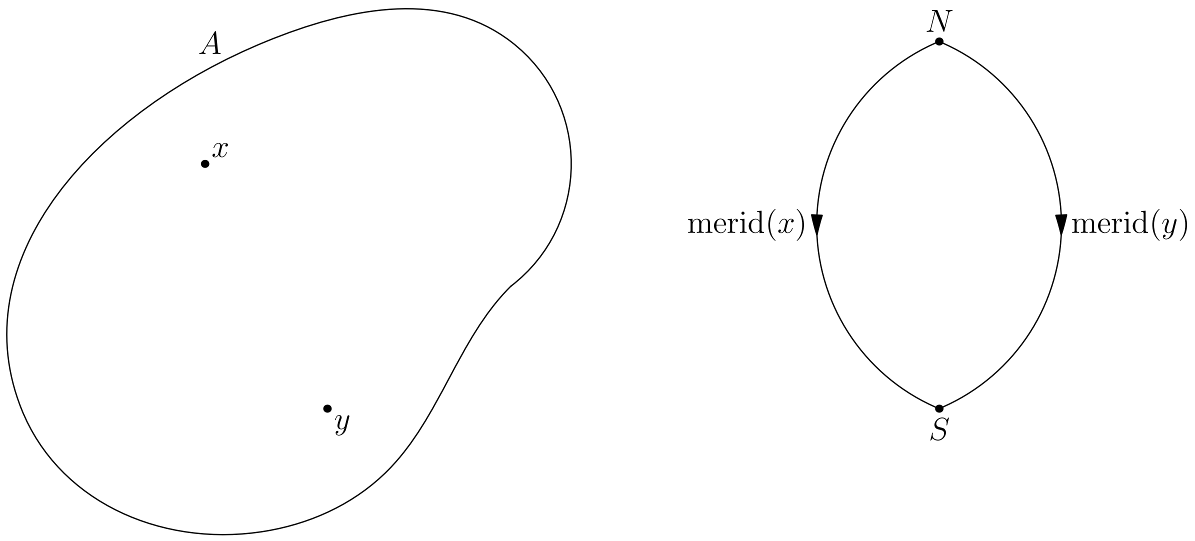

Figure 3.7: The suspension of the type A

3.3 Suspensions

The suspension of a type A

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

This notation is used to suggest a globe of sorts with a north pole N, a south pole S and for each element of A a meridian, i.e. a path between those two.

The induction principle states that given $P : \Sigma A \to \mathcal{U}$ together with

- a point $n : P(N)$,
- a point $s : P(S)$ and
- for each $a : A$ a path $m(a) : n =^{P}_{\mathrm{merid}(a)} s$,

we get a function $F : \prod_{(x:\Sigma A)} P(x)$ such that $F(N) \equiv n$, $F(S) \equiv s$ and for all $a : A$ we have $\mathrm{apd}_F(\mathrm{merid}(a)) = m(a)$. With the induction principle we can prove the recursion principle for the suspension type.

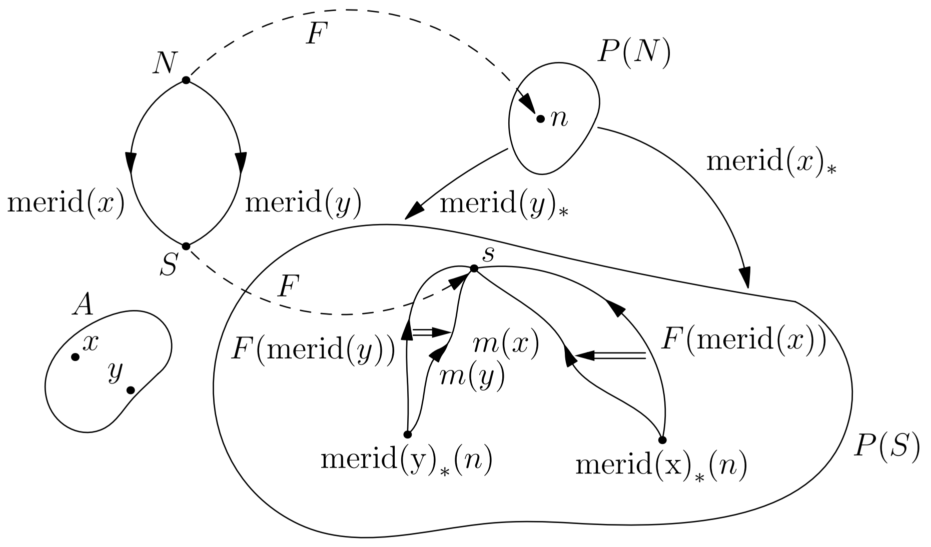

Figure 3.8: The induction principle for the suspension of the type A

Lemma 3.7. *(Recursion principle) For each B with*

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

we have a function $f : \Sigma A \to B$ such that $f(N) \equiv n$ and $f(S) \equiv s$, as well as for all $a : A$ $f(\mathit{merid}(a)) = m(a)$.

Proof. We use the induction principle with $P : \Sigma A \to \mathcal{U}$, where $P(x) \equiv B$. We have $n : P(N)$ and $s : P(S)$. Now we need a path $m'(a) : \text{merid}(a)^P_*(n) = s$ for every $a : A$. To see this fix an $a : A$. From lemma 2.9 we know that we have

$$\text{transportconst}^B_{\text{merid}(a)} : \text{merid}(a)^P_*(n) = n.$$

Since $m(a) : n = s$ we can consider the composite

$$\text{transportconst}^B_{\text{merid}(a)} \cdot m(a) : \text{merid}(a)^P_*(n) = s.$$

Now the induction principle on ΣA yields a function $F : \prod_{(x:\Sigma A)} P(x)$ such that $F(N) \equiv n$, $F(S) \equiv s$ and $\text{apd}_F(\text{merid}(a)) = \text{transportconst}^B_{\text{loop}} \cdot m(a)$ for all $a : A$. Note that F can also be seen as a non dependent function $f : A \to B$.

It remains to show that $\text{ap}_F(\text{merid}(a)) = m(a)$ for all $a : A$. From lemma 2.10 we know that

$$\text{apd}_F(\text{merid}(a)) = \text{transportconst}^B_{\text{merid}(a)} \cdot \text{ap}_F(\text{merid}(a))$$

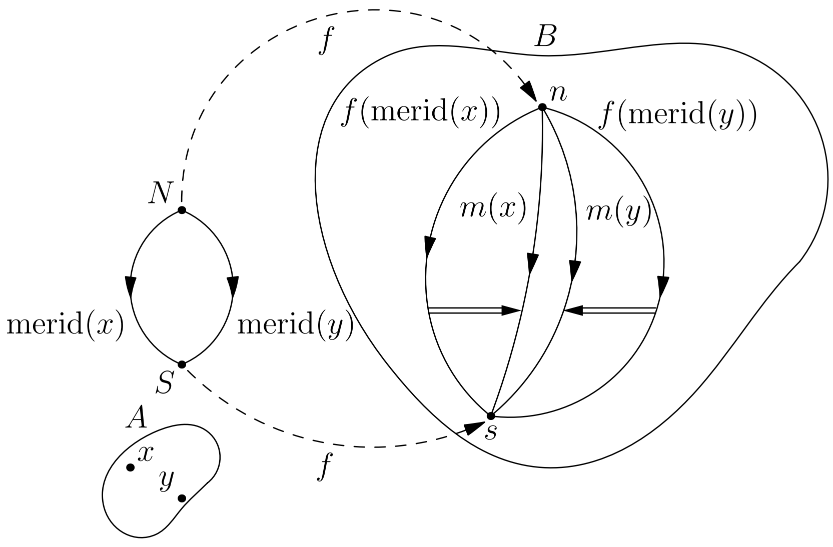

Figure 3.9: The recursion principle for the suspension of the type A

so we get that

$$\text{transportconst}^B_{\text{merid}(a)} \cdot \text{ap}_F(\text{merid}(a)) = \text{transportconst}^B_{\text{merid}(a)} \cdot m(a).$$

Now we can cancel transportconst and get $\text{ap}_F(\text{merid}(a)) = m(a)$. $\square$

Lemma 3.8. $\Sigma\mathbf{1} \simeq \mathrm{I}$

Proof. Since we have $0_I, 1_I : I$ and can define a function $m : \mathbf{1} \to (0_I =_I 1_I)$ by setting $m(\star) \equiv \text{seg}$, by recursion on $\Sigma\mathbf{1}$, we get a function $f : \Sigma\mathbf{1} \to I$ such that

- $f(N) \equiv 0_I$,
- $f(S) \equiv 1_I$ and
- $f(\text{merid}(1)) = \text{seg}$.

To define the inverse of f, we use recursion on I. We have $N, S : \Sigma\mathbf{1}$ and $\text{merid}(\star) : N = S$ so the recursion principle on I yields a function $g : I \to \Sigma\mathbf{1}$ such that

- $g(0_I) \equiv N$
- $g(1_I) \equiv S$
- $g(\text{seg}) = \text{merid}(\star)$

It remains to prove that f and g are indeed quasi-inverse.

First consider $f \circ g$. We use induction on I to prove $(f \circ g)(x) = x$ for all $x : I$. Define $P : I \to \mathcal{U}$ by $P(x) \equiv (f \circ g)(x) = x$. Since we have that

$$\begin{aligned} P(0_I) &\equiv ((f \circ g)(0_I) = 0_I) \equiv (f(N) = 0_I) \equiv (0_I = 0_I) \\ P(1_I) &\equiv ((f \circ g)(1_I) = 1_I) \equiv (f(S) = 1_I) \equiv (1_I = 1_I) \end{aligned}$$

we have $\mathrm{refl}_{0_I} : P(0_I)$ and $\mathrm{refl}_{1_I} : P(1_I)$. Now we need to give a path $\mathrm{refl}_{0_I} =^P_{\mathrm{seg}} \mathrm{refl}_{1_I}$. By definition this is just proving $\mathrm{transport}^{\lambda x.(f \circ g)(x)=x}(\mathrm{seg}, \mathrm{refl}_{0_I}) = \mathrm{refl}_{1_I}$. By lemma 2.11 we have that

$$\mathrm{transport}^{\lambda x.(f \circ g)(x)=x}(\mathrm{seg}, \mathrm{refl}_{0_I}) = (\mathrm{ap}_{f \circ g}(\mathrm{seg}))^{-1} \cdot \mathrm{refl}_{0_I} \cdot \mathrm{seg}$$

so, since we can simply cancel refl_{0_I}, it is sufficient to prove

$$(\mathrm{ap}_{f \circ g}(\mathrm{seg}))^{-1} \cdot \mathrm{seg} = \mathrm{refl}_{1_I}.$$

But since

$$\begin{aligned} \mathrm{ap}_{f \circ g}(\mathrm{seg}))^{-1} \cdot \mathrm{seg} &\equiv (\mathrm{ap}_f(\mathrm{ap}_g(\mathrm{seg})))^{-1} \cdot \mathrm{seg} \\ &\equiv (\mathrm{ap}_f(\mathrm{merid}(\star)))^{-1} \cdot \mathrm{seg} \\ &\equiv \mathrm{seg}^{-1} \cdot \mathrm{seg} \end{aligned}$$

this is just proving $\mathrm{seg}^{-1} \cdot \mathrm{seg} = \mathrm{refl}_{1_I}$ which is trivially true. Induction on I now yields a function of the type $\prod_{x:I}((f \circ g)(x) = x)$ thereby proving $(f \circ g)(x) = x$ for all $x : I$. Now we use induction on $\Sigma\mathbf{1}$ to prove $(g \circ f)(x) = x$ for all $x : \Sigma\mathbf{1}$. Define $Q : \Sigma\mathbf{1} \to \mathcal{U}$ by $Q(x) \equiv (g \circ f)(x) = x$. Since by definition

$$\begin{aligned} Q(N) &\equiv ((g \circ f)(N) = N) \equiv (g(0_I) = N) \equiv (N = N) \\ Q(S) &\equiv ((g \circ f)(S) = S) \equiv (g(1_I) = S) \equiv (S = S) \end{aligned}$$

we have $\mathrm{refl}_N : Q(N)$ and $\mathrm{refl}_S : Q(S)$. Now we need for each element $a : \mathbf{1}$ a prove of $\mathrm{refl}_N =^Q_{\mathrm{merid}(a)} \mathrm{refl}_S$. By induction on $\mathbf{1}$ it is sufficient to prove $\mathrm{refl}_N =^Q_{\mathrm{merid}(1)} \mathrm{refl}_S$. By definition this is the same as proving

$$\mathrm{transport}^{\lambda x.(g \circ f)(x)=x}(\mathrm{merid}(\star), \mathrm{refl}_N) = \mathrm{refl}_S$$

Again by lemma 2.11 we have that

$$\mathrm{transport}^{\lambda x.(g \circ f)(x)=x}(\mathrm{merid}(\star), \mathrm{refl}_N) = (\mathrm{ap}_{g \circ f}(\mathrm{merid}(\star))^{-1} \cdot \mathrm{refl}_N \cdot \mathrm{merid}(\star)$$

so, after cancelling refl_N it is sufficient to prove that

$$(\mathrm{ap}_{g \circ f}(\mathrm{merid}(\star))^{-1} \cdot \mathrm{merid}(\star) = \mathrm{refl}_S.$$

To see that this equality holds, we simplify the left side:

$$\begin{aligned} (\mathrm{ap}_{g \circ f}(\mathrm{merid}(\star))^{-1} \cdot \mathrm{merid}(\star) &\equiv (\mathrm{ap}_g(\mathrm{ap}_f(\mathrm{merid}(\star))))^{-1} \cdot \mathrm{merid}(\star) \\ &\equiv (\mathrm{ap}_g(\mathrm{seg}))^{-1} \cdot \mathrm{meird}(\star) \\ &\equiv (\mathrm{merid}(\star))^{-1} \cdot \mathrm{merid}(\star). \end{aligned}$$

That $(\mathrm{merid}(\star))^{-1} \cdot \mathrm{merid}(\star) = \mathrm{refl}_S$ is clear, so induction on $\Sigma\mathbf{1}$ yields a function of the type $\prod_{x:\Sigma\mathbf{1}}((g \circ f)(x) = x)$ thereby proving that g and f are really quasi-inverse.

□

Lemma 3.9. $\Sigma\mathbf{2} \simeq \mathbb{S}^1$

Proof. We use recursion on $\Sigma\mathbf{2}$ to define a function $f : \Sigma\mathbf{2} \to \mathbb{S}^1$. We need two points in $\mathbb{S}^1$ so we take base for both of them. Now we need a function $m : \mathbf{2} \to (\mathrm{base} =_{\mathbb{S}^1} \mathrm{base})$. We define this by recursion on $\mathbf{2}$ by setting $m(0_{\mathbf{2}}) \equiv \mathrm{loop}$ and $m(1_{\mathbf{2}}) \equiv \mathrm{refl}_{\mathrm{base}}$. Recursion on $\Sigma\mathbf{2}$ now yields a function $f : \Sigma\mathbf{2} \to \mathbb{S}^1$ such that

$$\begin{aligned} f(N) &\equiv \mathrm{base} \\ f(S) &\equiv \mathrm{base} \\ f(\mathrm{merid}(0_{\mathbf{2}})) &= \mathrm{loop} \\ f(\mathrm{merid}(1_{\mathbf{2}})) &= \mathrm{refl}_{\mathrm{base}}. \end{aligned}$$

Now we use recursion on $\mathbb{S}^1$ to define the inverse function $g : \mathbb{S}^1 \to \Sigma\mathbf{2}$. We have a point $N : \Sigma\mathbf{2}$ and a path $\mathrm{merid}(0_{\mathbf{2}}) \cdot \mathrm{merid}(1_{\mathbf{2}})^{-1}$ from N to itself. By recursion we get $g : \mathbb{S}^1 \to \Sigma\mathbf{2}$ such that

$$\begin{aligned} g(\mathrm{base}) &\equiv N \\ g(\mathrm{loop}) &= \mathrm{merid}(0_{\mathbf{2}}) \cdot \mathrm{merid}(1_{\mathbf{2}})^{-1}. \end{aligned}$$

It remains to show that f and g are quasi-inverses. First we prove by induction on $\Sigma\mathbf{2}$ that for all $x : \Sigma\mathbf{2}$ we have $g(f(x)) = x$. For this define $P : \Sigma\mathbf{2} \to \mathcal{U}$ by $P(x) \equiv ((g(f(x)) = x)$. Since $P(N) \equiv (N = N)$ and $P(S) \equiv (N = S)$ we have

- $\mathrm{refl}_N : P(N)$
- $\mathrm{merid}(1_{\mathbf{2}}) : P(S)$

It remains to give for each $x : \mathbf{2}$ a term $m(x) : \mathrm{refl}_N =^P_{\mathrm{merid}(x)} \mathrm{merid}(1_{\mathbf{2}})$. By definition this amounts to:

$$\mathrm{transport}^{\lambda x. g(f(x))=x}(\mathrm{merid}(x), \mathrm{refl}_N) = \mathrm{merid}(1_{\mathbf{2}})$$

By lemma 2.11 we get that

$$\mathrm{transport}^{\lambda x.(g\circ f)(x)=x}(\mathrm{merid}(x), \mathrm{refl}_N) = (\mathrm{ap}_{(g\circ f)}(\mathrm{merid}(x)))^{-1} \cdot \mathrm{refl}_N \cdot \mathrm{merid}(x)$$

and since we can cancel refl_N, it now only remains to prove that

$$(\mathrm{ap}_{(g\circ f)}(\mathrm{merid}(x)))^{-1} \cdot \mathrm{merid}(x) = \mathrm{merid}(1_{\mathbf{2}}).$$

By induction on $\mathbf{2}$ we take cases:

1. If $x \equiv 0_{\mathbf{2}}$, we get

$$\begin{aligned}(\mathrm{ap}_{(g\circ f)}(\mathrm{merid}(0_{\mathbf{2}})))^{-1} \cdot \mathrm{merid}(0_{\mathbf{2}}) &= (g(f(\mathrm{merid}(0_{\mathbf{2}})))^{-1} \cdot \mathrm{merid}(0_{\mathbf{2}}) \\ &= (g(\mathrm{loop}))^{-1} \cdot \mathrm{merid}(0_{\mathbf{2}}) \\ &= (\mathrm{merid}(0_{\mathbf{2}}) \cdot \mathrm{merid}(1_{\mathbf{2}})^{-1})^{-1} \cdot \mathrm{merid}(0_{\mathbf{2}}) \\ &= (\mathrm{merid}(1_{\mathbf{2}})^{-1})^{-1} \cdot \mathrm{merid}(0_{\mathbf{2}})^{-1} \cdot \mathrm{merid}(0_{\mathbf{2}}) \\ &= \mathrm{merid}(1_{\mathbf{2}})\end{aligned}$$

2. If $x \equiv 1_{\mathbf{2}}$, we get

$$\begin{aligned}(\mathrm{ap}_{(g\circ f)}(\mathrm{merid}(1_{\mathbf{2}})))^{-1} \cdot \mathrm{merid}(1_{\mathbf{2}}) &= (g(f(\mathrm{merid}(1_{\mathbf{2}})))^{-1} \cdot \mathrm{merid}(1_{\mathbf{2}}) \\ &= (g(\mathrm{refl}_{\mathrm{base}}))^{-1} \cdot \mathrm{merid}(1_{\mathbf{2}}) \\ &= \mathrm{refl}_N \cdot \mathrm{merid}(1_{\mathbf{2}}) \\ &= \mathrm{merid}(1_{\mathbf{2}})\end{aligned}$$

Therefore we have $(g(f(x)) = x$, for all $x : \Sigma\mathbf{2}$.

Now we use induction on $\mathbb{S}^1$ to prove $f(g(x)) = x$ for all $x : \mathbb{S}^1$. For this we define $Q : \mathbb{S}^1 \to \Sigma\mathbf{2}$ by $Q(x) \equiv (f(g(x)) = x)$. Since $Q(\mathrm{base}) \equiv (\mathrm{base} = \mathrm{base})$ we have $\mathrm{refl}_N : Q(\mathrm{base})$. It remains to give an path $\mathrm{refl}_{\mathrm{base}} =^{Q}_{\mathrm{loop}} \mathrm{refl}_{\mathrm{base}}$. By definition this amounts to

$$\mathrm{transport}^{\lambda x.(f\circ g(x)=x)}(\mathrm{loop}, \mathrm{refl}_{\mathrm{base}}) = \mathrm{refl}_{\mathrm{base}}.$$

Again we use lemma 2.11 and we get that

$$\mathrm{transport}^{\lambda x.(f\circ g(x)=x)}(\mathrm{loop}, \mathrm{refl}_{\mathrm{base}}) = (\mathrm{ap}_{(f\circ g)}(\mathrm{loop}))^{-1} \cdot \mathrm{refl}_{\mathrm{base}} \cdot \mathrm{loop}$$

so it only remains to show

$$(\mathrm{ap}_{(f\circ g)}(\mathrm{loop}))^{-1} \cdot \mathrm{refl}_{\mathrm{base}} \cdot \mathrm{loop} = \mathrm{refl}_{\mathrm{base}}$$

We use lemma 2.4 to see that

$$\begin{aligned}(\mathrm{ap}_{(f\circ g)}(\mathrm{loop}))^{-1} \cdot \mathrm{refl}_{\mathrm{base}} \cdot \mathrm{loop} &= f(g(\mathrm{loop}))^{-1} \cdot \mathrm{loop} \\ &= f(\mathrm{merid}(0_{\mathbf{2}}) \cdot \mathrm{merid}(1_{\mathbf{2}})^{-1})^{-1} \cdot \mathrm{loop} \\ &= (f(\mathrm{merid}(0_{\mathbf{2}})) \cdot f(\mathrm{merid}(1_{\mathbf{2}}))^{-1})^{-1} \cdot \mathrm{loop} \\ &= f(\mathrm{merid}(1_{\mathbf{2}})) \cdot f(\mathrm{merid}(0_{\mathbf{2}}))^{-1} \cdot \mathrm{loop} \\ &= \mathrm{refl}_{\mathrm{base}} \cdot \mathrm{loop}^{-1} \cdot \mathrm{loop} \\ &= \mathrm{refl}_{\mathrm{base}}\end{aligned}$$

By induction on $\mathbb{S}^1$ we get that $f(g(x)) = x$ for all $x : \mathbb{S}^1$. □

3.4 The fundamental group of the circle

In this section we will prove that the type of paths base $=_{\mathbb{S}^1}$ base is equivalent to the integers $\mathbb{Z}$.

Lemma 3.10. *There is a function* wind : $\mathbb{Z} \to (\text{base} =_{\mathbb{S}^1} \text{base})$*, that maps* $\text{pos}(0_{\mathbb{N}})$ *to* loop

Proof. Define $\text{wind}(0) \equiv \text{refl}_{\text{base}}$, define $\text{wind}(\text{pos}(0_{\mathbb{N}})) \equiv \text{loop}$ and define

$$\text{wind}(\text{pos}(\text{succ}_{\mathbb{N}}(n))) \equiv \text{wind}(\text{pos}(n)) \cdot \text{loop}.$$

Define $\text{wind}(\text{neg}(0_{\mathbb{N}})) \equiv \text{loop}^{-1}$ and define

$$\text{wind}(\text{neg}(\text{succ}_{\mathbb{N}}(n))) \equiv \text{wind}(\text{neg}(n)) \cdot \text{loop}^{-1}.$$

□

Definition 3.11. There is a function code : $\mathbb{S}^1 \to \mathcal{U}$

$$\begin{aligned} \text{code}(\text{base}) &\equiv \mathbb{Z} \\ \text{ap}_{\text{code}}(\text{loop}) &= \text{ua}(\text{succ}_{\mathbb{Z}}) \end{aligned}$$

This function is defined by recursion on $\mathbb{S}^1$. Note that we need the strong recursion principle we explained in remark 2.23 to do this.

Lemma 3.12. *We have*

$$\begin{aligned} (\text{loop})_*^{\text{code}} &= \text{succ}_{\mathbb{Z}} \\ (\text{loop}^{-1})_*^{\text{code}} &= \text{succ}_{\mathbb{Z}}^{-1} \end{aligned}$$

Proof. By lemma 2.20 we have

$$\begin{aligned} \text{loop}_*^{\text{code}} &= \text{idtoeqv} \circ \text{ap}_{\text{code}} \circ \text{loop} \\ &= \text{idtoeqv} \circ \text{ua} \circ \text{succ} \\ &= \text{succ} \end{aligned}$$

where we used that idtoeqv and ua are quasi-inverse. Since transportation respects inverses by lemma 2.19, we have

$$\begin{aligned} (\text{loop}^{-1})_*^{\text{code}} &= (\text{loop}_*^{\text{code}})^{-1} \\ &= \text{succ}^{-1} \end{aligned}$$

□

Lemma 3.13. *For all $z : \mathbb{Z}$ we have*

$$\mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(z) = \mathrm{wind}(z) \cdot \mathrm{loop}^{-1}.$$

Proof. We use induction on $\mathbb{Z}$.

1. If $z \equiv 0_{\mathbb{Z}}$, we have to prove $\mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(0_{\mathbb{Z}}) = \mathrm{wind}(0_{\mathbb{Z}}) \cdot \mathrm{loop}^{-1}$. But $\mathrm{wind}(0_{\mathbb{Z}}) \equiv \mathrm{refl}_{\mathrm{base}}$ and $\mathrm{wind}(\mathrm{neg}(0_{\mathbb{N}})) \equiv \mathrm{loop}^{-1}$ so this is clear.
2. If $z \equiv \mathrm{pos}(n)$ for some $n : \mathbb{N}$, we use induction on $\mathbb{N}$.

   If $z \equiv \mathrm{pos}(0_{\mathbb{N}})$, we have to prove $\mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(\mathrm{pos}(0_{\mathbb{N}})) = \mathrm{wind}(\mathrm{pos}(0_{\mathbb{N}})) \cdot \mathrm{loop}^{-1}$. By definition we have $\mathrm{wind}(\mathrm{pos}(0_{\mathbb{N}}) \equiv \mathrm{loop}$ and we know that $\mathrm{succ}_{\mathbb{Z}}^{-1}(\mathrm{pos}(0_{\mathbb{N}})) \equiv 0_{\mathbb{Z}}$ so this is just proving

   $$\mathrm{wind}(0_{\mathbb{Z}}) = \mathrm{loop} \cdot \mathrm{loop}^{-1}.$$

   This, however, is true since both sides are equal to $\mathrm{refl}_{\mathrm{base}}$.

   If $z \equiv \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))$ for some $n : \mathbb{N}$, we calculate:

   $$\begin{aligned} \mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))) &= \mathrm{wind}(\mathrm{pos}(n)) \\ &= \mathrm{wind}(\mathrm{pos}(n)) \cdot \mathrm{loop} \cdot \mathrm{loop}^{-1} \\ &= \mathrm{wind}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))) \cdot \mathrm{loop}^{-1}, \end{aligned}$$

   where we used the definition of wind for the last equation.
3. If $c \equiv \mathrm{neg}(n)$ for some $n : \mathbb{N}$, we use induction on $\mathbb{N}$ and the proof works similar to the one we just did.

□

Definition 3.14. We define $\mathrm{encode} : \prod_{x:\mathbb{S}^1} \prod_{p:(\mathrm{base} =_{\mathbb{S}^1} x)} \mathrm{code}(x)$ by

$$\mathrm{encode}(x, p) \equiv \mathrm{transport}^{\mathrm{code}}(p, 0_{\mathbb{Z}}).$$

This is well typed since $\mathrm{transport}^{\mathrm{code}}(p, -)$ is a function from $\mathbb{Z}$ into $\mathrm{code}(x)$.

Definition 3.15. We defne $\mathrm{decode} : \prod_{x:\mathbb{S}^1} (\mathrm{code}(x) \to \mathrm{base} =_{\mathbb{S}^1} x)$ by induction on $\mathbb{S}^1$, by

$$\begin{aligned} \mathrm{decode}(\mathrm{base}, -) &\equiv \mathrm{wind} \\ \mathrm{apd}_{\mathrm{decode}}(\mathrm{loop}) &= l, \end{aligned}$$

where l is a path $\mathrm{loop}_*^P(\mathrm{wind}) = \mathrm{wind}$, with $P : \mathbb{S}^1 \to \mathcal{U}$ beeing defined by

$$P(x) \equiv (\mathrm{code}(x) \to \mathrm{base} =_{\mathbb{S}^1} x).$$

To see that this is well typed, first note that $P(\mathrm{base}) \equiv \mathbb{Z} \to (\mathrm{base} =_{\mathbb{S}^1} \mathrm{base})$ so we have $\mathrm{wind} : P(\mathrm{base})$.

Now we need to prove that $\mathrm{loop}_*^P(\mathrm{wind}) = \mathrm{wind}$. By definition we have

$$\mathrm{loop}_*^P(\mathrm{wind}) \equiv \mathrm{loop}_*^{\lambda x.\left(\mathrm{code}(x)\to \mathrm{base}=_{\mathbb{S}^1} x\right)}(\mathrm{wind}).$$

We use lemma 2.13 followed by lemma 3.12,which states that, $(\mathrm{loop}^{-1})_*^{\mathrm{code}} = \mathrm{succ}_{\mathbb{Z}}^{-1}$, to see that

$$\begin{aligned}\mathrm{loop}_*^{\lambda x.\left(\mathrm{code}(x)\to \mathrm{base}=_{\mathbb{S}^1} x\right)}(\mathrm{wind}) &= \mathrm{loop}_*^{\lambda x.(\mathrm{base}=x)} \circ \mathrm{wind} \circ (\mathrm{loop}^{-1})_*^{\mathrm{code}} \\ &= \mathrm{loop}_*^{\lambda x.(\mathrm{base}=x)} \circ \mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}.\end{aligned}$$

Next we prove that for all $z : \mathbb{Z}$ we have

$$\mathrm{loop}_*^{\lambda x.(\mathrm{base}=x)} \circ \mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(z) = \mathrm{wind}(z).$$

Function extensionality then implies that $\mathrm{loop}_*^{\lambda x.(\mathrm{base}=x)} \circ \mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1} = \mathrm{wind}$, completing our proof. Let $z : \mathbb{Z}$ be arbitrary. We can rewrite $\mathrm{loop}_*^{\lambda x.(\mathrm{base}=x)} \circ \mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(z)$ and then apply lemma 2.12 to see that

$$\begin{aligned}\mathrm{loop}_*^{\lambda x.(\mathrm{base}=x)} \circ \mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(z) &= \mathrm{loop}_*^{\lambda x.(\mathrm{base}=x)}(\mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(z)) \\ &= (\mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(z)) \cdot \mathrm{loop}\end{aligned}$$

Now we can use the connection between $\mathrm{succ}_{\mathbb{Z}}$ and wind, given by lemma 3.13, to see that

$$\begin{aligned}(\mathrm{wind} \circ \mathrm{succ}_{\mathbb{Z}}^{-1}(z)) \cdot \mathrm{loop} &= (\mathrm{wind}(z) \cdot \mathrm{loop}^{-1}) \cdot \mathrm{loop} \\ &= \mathrm{wind}(z) \cdot (\mathrm{loop}^{-1} \cdot \mathrm{loop}) \\ &= \mathrm{wind}(z) \cdot \mathrm{refl}_{\mathrm{base}} \\ &= \mathrm{wind},\end{aligned}$$

thereby completing the proof.

Lemma 3.16. *Let* $x : \mathbb{S}^1, p : \mathrm{base} =_{\mathbb{S}^1} x$ *then*

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

Proof. Since $\mathrm{apd}_{\mathrm{decode}}(p) : p_*^{(\mathrm{code}(x)\to \mathrm{base}=_{\mathbb{S}^1} x)}(\mathrm{decode}(\mathrm{base})) = \mathrm{decode}(x)$ we have

$$\begin{aligned}\mathrm{decode}(x, \mathrm{encode}(x, p)) &\equiv \mathrm{decode}(x)(p_*^{\mathrm{code}}(0_{\mathbb{Z}})) \\ &= p_*^{x\mapsto(\mathrm{code}(x)\to \mathrm{base}=x)}(\mathrm{decode}(\mathrm{base}))(p_*^{\mathrm{code}}(0_{\mathbb{Z}})) \\ &\equiv p_*^{x\mapsto(\mathrm{code}(x)\to \mathrm{base}=x)}(\mathrm{wind})(p_*^{\mathrm{code}}(0_{\mathbb{Z}})).\end{aligned}$$

With lemma 2.13 and lemma 2.14 we can rewrite this

$$\begin{aligned}
p_*^{x\mapsto(\mathrm{code}(x)\to \mathrm{base}=x)}(\mathrm{wind})(p_*^{\mathrm{code}}(0_{\mathbb{Z}})) &= p_*^{x\mapsto \mathrm{base}=x}\circ \mathrm{wind}\circ (p^{-1})_*^{\mathrm{code}}(p_*^{\mathrm{code}}(0_{\mathbb{Z}}))\\
&= p_*^{x\mapsto \mathrm{base}=x}\circ \mathrm{wind}\circ (p \cdot p^{-1})_*^{\mathrm{code}}(0_{\mathbb{Z}})\\
&= p_*^{x\mapsto \mathrm{base}=x}(\mathrm{wind}\circ (\mathrm{refl}_{\mathrm{base}})_*^{\mathrm{code}}(0_{\mathbb{Z}}))\\
&= p_*^{x\mapsto \mathrm{base}=x}(\mathrm{wind}(0_{\mathbb{Z}}))\\
&= p_*^{x\mapsto \mathrm{base}=x}(\mathrm{refl}_{\mathrm{base}})\\
&= \mathrm{refl}_{\mathrm{base}} \cdot p\\
&= p.
\end{aligned}$$

□

Lemma 3.17. *For $x : \mathbb{S}^1$ and $c : \mathrm{code}(x)$ we have*

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

Proof. We define $P : \mathbb{S}^1 \to \mathcal{U}$ by $P(x) \equiv \prod_{c:\mathrm{code}(x)}(\mathrm{encode}(x, \mathrm{decode}(x, c)) = c)$. To prove the lemma we have to prove

$$\prod_{x:\mathbb{S}^1} P(x).$$

For this we use induction on $\mathbb{S}^1$. If $x \equiv \mathrm{base}$ we have to give a proof of

$$\prod_{c:\mathrm{code}(\mathrm{base})}(\mathrm{encode}(\mathrm{base}, \mathrm{decode}(\mathrm{base}, c)) = c).$$

By definition we have that $\mathrm{code}(\mathrm{base}) \equiv \mathbb{Z}$ as well as

$$\begin{aligned}
(\mathrm{encode}(\mathrm{base}, \mathrm{decode}(\mathrm{base}, c)) = c) &\equiv (\mathrm{encode}(\mathrm{base}, \mathrm{wind}(c)) = c)\\
&\equiv (\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(c), 0_{\mathbb{Z}}) = c).
\end{aligned}$$

So we can simply use induction on $\mathbb{Z}$ to prove

$$\prod_{c:\mathbb{Z}}(\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(c), 0_{\mathbb{Z}}) = c).$$

1. If $c \equiv 0_{\mathbb{Z}}$, we have to prove $(\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(0_{\mathbb{Z}}), 0_{\mathbb{Z}}) = 0_{\mathbb{Z}})$. But $\mathrm{wind}(0_{\mathbb{Z}}) \equiv \mathrm{refl}_{\mathrm{base}}$ and by definition $(\mathrm{transport}^{\mathrm{code}}(\mathrm{refl}_{\mathrm{base}}, 0_{\mathbb{Z}}) \equiv \mathrm{id}_{\mathbb{Z}}(0_{\mathbb{Z}})$; so this is clear.
2. If $c \equiv \mathrm{pos}(n)$ for some $n : \mathbb{N}$, we use induction on $\mathbb{N}$.

   If $c \equiv \mathrm{pos}(0_{\mathbb{N}})$, we have to prove $(\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(\mathrm{pos}(0_{\mathbb{N}})), 0_{\mathbb{Z}}) = \mathrm{pos}(0_{\mathbb{N}}))$. By definition we have $\mathrm{wind}(\mathrm{pos}(0_{\mathbb{N}}) \equiv \mathrm{loop}$ and we know that

   $$\mathrm{transport}^{\mathrm{code}}(\mathrm{loop}, 0_{\mathbb{Z}}) = \mathrm{succ}(0_{\mathbb{Z}}),$$

   which again by definition is $\mathrm{pos}(0_{\mathbb{N}})$.

If $c \equiv \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))$ for some $n : \mathbb{N}$, we calculate:

$$\begin{aligned}\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(\mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n))), 0_{\mathbb{Z}}) &= \mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(\mathrm{pos}(n)) \centerdot \mathrm{loop}, 0_{\mathbb{Z}}) \\ &= \mathrm{loop}_*^{\mathrm{code}}(\mathrm{wind}(\mathrm{pos}(n))_*^{\mathrm{code}}(0_{\mathbb{Z}}))\end{aligned}$$

Now we use the induction assumption and get

$$\begin{aligned}\mathrm{loop}_*^{\mathrm{code}}(\mathrm{wind}(\mathrm{pos}(n))_*^{\mathrm{code}}(0_{\mathbb{Z}})) &= \mathrm{loop}_*^{\mathrm{code}}(\mathrm{pos}(n)) \\ &= \mathrm{succ}_{\mathbb{Z}}(\mathrm{pos}(n)) \\ &= \mathrm{pos}(\mathrm{succ}_{\mathbb{N}}(n)).\end{aligned}$$

3. If $c \equiv \mathrm{neg}(n)$ for some $n : \mathbb{N}$, we use induction on $\mathbb{N}$ and the proof works similar to the one we just did.

This means we have found an element $a_0 : P(\mathrm{base})$. Now to complete the induction on $\mathbb{S}^1$, we have to prove that $\mathrm{loop}_*^P(a_0) = a_0$. a_0 and $\mathrm{loop}_*^P(a_0)$ are both of the type $\prod_{c:\mathbb{Z}}(\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(c), 0_{\mathbb{Z}}) = c)$. That means for every $c : \mathbb{Z}$ a_0 and $\mathrm{loop}_*^P(a_0)$ are both proofs for $\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(c), 0_{\mathbb{Z}}) =_{\mathbb{Z}} c$. By lemma 2.34 $\mathbb{Z}$ is a socalled set, i.e. all the paths $\mathrm{transport}^{\mathrm{code}}(\mathrm{wind}(c), 0_{\mathbb{Z}}) =_{\mathbb{Z}} c$ are equal in the identity type. So the two functions a_0 and $\mathrm{loop}_*^P(a_0)$ have the same value on every element of $\mathbb{Z}$ and therefore are equal as functions by function extentionality. □

Theorem 3.18. $\prod_{x:\mathbb{S}^1}[(\mathrm{base} =_{\mathbb{S}^1} x) \simeq \mathrm{code}(x)]$.

Proof. Just combining the two lemmata above. □

Corollary 3.19. $(\mathrm{base} = \mathrm{base}) \simeq \mathbb{Z}$.

Proof. Immediate from the theorem above with $x \equiv \mathrm{base}$. □

Bibliography

[1] S. Awodey, M.A. Warren. Homotpy theoretic models of identity types, Mathematical Proceedings of the Cambridge Philosopical Society, 2009, 146:45-55.

[2] M. Hofmann, T.Streicher: The groupoid interpretation of type theory, G. Sambin, J. M. Smith (Eds.) *Twenty-five years of constructive type theory* (Venice, 1995), volume 36 of Oxford Logic Guides, Oxford University Press, 1998, 83-111.

[3] P. Martin-Löf: An intuitionistic theory of types: predicative part, in H. E. Rose and J. C. Shepherdso (Eds.), *Logic Colloquium'73*, pp.73-118, North-Holland, 1975.

[4] P. Martin-Löf: *Intuitionistic type theory: Notes by Giovanni Sambi on a series of lectures given in Padua, June 1980*, Napoli: Bibliopolis, 1984.

[5] P. Martin-Löf: An ituitionsistic theory of types, in G. Sambin, J. M. Smith (Eds.) *Twenty five years of constructive type theory* (Venice, 1995), voume 36 of Oxford Logic Guides, Oxford University Press, 1998, 127-172.

[6] E. Rijke: *Homotopy type theory*, Master Thesis, Utrecht University 2012.

[7] The Univalent Foundations Program: *Homotopy Type Theory: Univalent Foundations Of Mathematics*, institute for Advanced Study, Princeton, 2013.

[8] V. Voevodsky: A very short note on the homtopy λ- calculus, in https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2006_09_Hlambda_0.pdf, 2006.

[9] D. R. Licata, M. Shulman: Calculating the Fundamental Group of the Circle in Homotopy Type Theory, *Logic in Computer Science*, pp.223-232, IEEE, 2013.

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie wörtlich oder inhaltlich übernommene Stellen als solche kenntlich gemacht habe.

München, den 17. Juli 2018