Introduction to homotopy type theory

Egbert Rijke

Lecture Notes

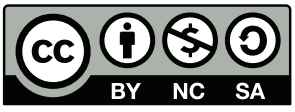
for the

Summer School on Homotopy Type Theory

Carnegie Mellon University
Pittsburgh PA

August 2019

The author gratefully acknowledges the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053.



Contents

Chapter I

Martin-Löf's dependent type theory

1 Dependent type theory

Dependent type theory is a system of inference rules that can be combined to make *derivations*. In these derivations, the goal is often to construct a term of a certain type. Such a term can be a function if the type of the constructed term is a function type; a proof of a property if the type of the constructed term is a proposition; an identification if the type of the constructed term is an identity type, and so on. In some respect, a type is just a collection of mathematical objects and constructing terms of a type is the everyday mathematical task or challenge. The system of inference rules that we call type theory offers a principled way of engaging in mathematical activity.

1.1 Judgments and contexts in type theory

An **inference rule** is an expression of the form

$$\frac{\mathcal{H}_1 \quad \mathcal{H}_2 \quad \dots \quad \mathcal{H}_n}{\mathcal{C}}$$

containing above the horizontal line a finite list $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_n$ of *judgments* for the hypotheses, and below the horizontal line a single judgment $\mathcal{C}$ for the conclusion. A very simple example that we will encounter in §2 when we introduce function types, is the inference rule

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash f(a) : B}$$

This rule asserts that in any context Γ we may use a term $a : A$ and a function $f : A \to B$ to obtain a term $f(a) : B$. Each of the expressions

$$\begin{aligned} &\Gamma \vdash a : A \\ &\Gamma \vdash f : A \to B \\ &\Gamma \vdash f(a) : B \end{aligned}$$

are examples of judgments. There are four kinds of judgments in type theory:

(i) *A is a (well-formed)* ***type*** *in context* Γ. The symbolic expression for this judgment is

$$\Gamma \vdash A \text{ type}$$

(ii) *A and B are* ***judgmentally equal types*** *in context* Γ. The symbolic expression for this judgment is

$$\Gamma \vdash A \equiv B \text{ type}$$

(iii) *a is a (well-formed)* ***term*** *of type A in context* Γ. The symbolic expression for this judgment is

$$\Gamma \vdash a : A$$

(iv) *a and b are* ***judgmentally equal terms*** *of type A in context* Γ. The symbolic expression for this judgment is

$$\Gamma \vdash a \equiv b : A$$

Thus we see that any judgment is of the form $\Gamma \vdash \mathcal{J}$, consisting of a context Γ and an expression $\mathcal{J}$ asserting that A is a type, that A and B are equal types, that a is a term of type A, or that a and b are equal terms of type A. The role of a context is to declare what hypothetical terms are assumed, along with their types. More formally, a **context** is an expression of the form

$$x_1 : A_1,\ x_2 : A_2(x_1),\ \ldots,\ x_n : A_n(x_1, \ldots, x_{n-1}) \tag{1.1}$$

satisfying the condition that for each $1 \leq k \leq n$ we can derive, using the inference rules of type theory, that

$$x_1 : A_1,\ x_2 : A_2(x_1),\ \ldots,\ x_{k-1} : A_{k-1}(x_1, \ldots, x_{k-2}) \vdash A_k(x_1, \ldots, x_{k-1}) \text{ type}. \tag{1.2}$$

In other words, to check that an expression of the form Eq. (1.1) is a context, one starts on the left and works their way to the right verifying that each hypothetical term x_k is assigned a well-formed type. Hypothetical terms are commonly called **variables**, and we say that a context as in Eq. (1.1) **declares the variables** $x_1, \ldots, x_n$. We may use variable names other than $x_1, \ldots, x_n$, as long as no variable is declared more than once.

The condition in Eq. (1.2) that each of the hypothetical terms is assigned a well-formed type, is checked recursively. Note that the context of length 0 satisfies the requirement in Eq. (1.2) vacuously. This context is called the **empty context**. An expression of the form $x_1 : A_1$ is a context if and only if A_1 is a well-formed type in the empty context. Such types are called **closed types**. We will soon encounter the type $\mathbb{N}$ of natural numbers, which is an example of a closed type. There is also the notion of **closed term**, which is simply a term in the empty context. The next case is that an expression of the form $x_1 : A_1,\ x_2 : A_2(x_1)$ is a context if and only if A_1 is a well-formed type in the empty context, and $A_2(x_1)$ is a well-formed type, given a hypothetical term $x_1 : A_1$. This process repeats itself for longer contexts.

It is a feature of *dependent* type theory that all judgments are context-dependent, and indeed that even the types of the variables may depend on any previously declared variables. For example, when we introduce the *identity type* in §5, we make full use of the machinery of type dependency, as is clear from how they are introduced:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma, x : A, y : A \vdash x = y \text{ type}}$$

This rule asserts that given a type A in context Γ, we may form a type $x = y$ in context $\Gamma, x : A, y : A$. Note that in order to know that the expression $\Gamma, x : A, y : A$ is indeed a well-formed context, we need to know that A is a well-formed type in context $\Gamma, x : A$. This is an instance of *weakening*, which we will describe shortly.

In the situation where we have

$$\Gamma, x : A \vdash B(x) \text{ type},$$

we say that B is a **family** of types over A in context Γ. Alternatively, we say that $B(x)$ is a type **indexed** by $x : A$, in context Γ. Similarly, in the situation where we have

$$\Gamma, x : A \vdash b(x) : B(x),$$

we say that b is a **section** of the family B over A in context Γ. Alternatively, we say that $b(x)$ is a term of type $B(x)$, **indexed** by $x : A$ in context Γ. Note that in the above situations A, B, and b also depend on the variables declared in the context Γ, even though we have not explicitly mentioned them. It is common practice to not mention every variable in the context Γ in such situations.

1.2 Inference rules

In this section we present the basic inference rules of dependent type theory. Those rules are valid to be used in any type theoretic derivation. There are only four sets of inference rules:

(i) Rules for judgmental equality

(ii) Rules for substitution

(iii) Rules for weakening

(iv) The "variable rule"

Judgmental equality

In this set of inference rules we ensure that judgmental equality (both on types and on terms) are equivalence relations, and we make sure that in any context Γ, we can change the type of any variable to a judgmentally equal type.

The rules postulating that judgmental equality on types and on terms is an equivalence relation are as follows:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash A \equiv A \text{ type}} \qquad \frac{\Gamma \vdash A \equiv A' \text{ type}}{\Gamma \vdash A' \equiv A \text{ type}} \qquad \frac{\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma \vdash A' \equiv A'' \text{ type}}{\Gamma \vdash A \equiv A'' \text{ type}}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv a' : A}{\Gamma \vdash a' \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv a' : A \qquad \Gamma \vdash a' \equiv a'' : A}{\Gamma \vdash a \equiv a'' : A}$$

Apart from the rules postulating that judgmental equality is an equivalence relation, there are also **variable conversion rules**. Informally, these are rules stating that if A and

A' are judgmentally equal types in context Γ, then any valid judgment in context $\Gamma, x : A$ is also a valid judgment in context $\Gamma, x : A'$. In other words: we can convert the type of a variable to a judgmentally equal type.

The first variable conversion rule states that

$$\frac{\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma, x : A, \Delta \vdash B(x) \text{ type}}{\Gamma, x : A', \Delta \vdash B(x) \text{ type}}$$

In this conversion rule, the context of the form $\Gamma, x : A, \Delta$ is just any extension of the context $\Gamma, x : A$.

Similarly, there are variable conversion rules for judgmental equality of types, for terms, and for judgmental equality of terms. To avoid having to state essentially the same rule four times, we state all four variable conversion rules at once using a *generic judgment* $\mathcal{J}$, which can be any of the four kinds of judgments.

$$\frac{\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, x : A', \Delta \vdash \mathcal{J}}$$

An analogous *term conversion rule*, stated in Exercise 1.1, converting the type of a term to a judgmentally equal type, is derivable using the rules for substitution and weakening, and the variable rule.

Substitution

If we are given a term $a : A$ in context Γ, then for any type B in context $\Gamma, x : A, \Delta$ we can form the type $B[a/x]$ in context $\Gamma, \Delta[a/x]$, where $B[a/x]$ is an abbreviation for

$$B(x_1, \ldots, x_{n-1}, a(x_1, \ldots, x_{n-1}), x_{n+1}, \ldots, x_{n+m-1}).$$

This syntactic operation of substituting a for x is understood to be defined recursively over the length of Δ. Similarly we obtain for any term $b : B$ in context $\Gamma, x : A, \Delta$ a term $b[a/x] : B[a/x]$. The **substitution rule** asserts that substitution preserves well-formedness and judgmental equality of types and terms:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, \Delta[a/x] \vdash \mathcal{J}[a/x]}\, S_a$$

Furthermore, we postulate that substitution by judgmentally equal terms results in judgmentally equal types

$$\frac{\Gamma \vdash a \equiv a' : A \qquad \Gamma, x : A, \Delta \vdash B \text{ type}}{\Gamma, \Delta[a/x] \vdash B[a/x] \equiv B[a'/x] \text{ type}}$$

and it also results in judgmentally equal terms

$$\frac{\Gamma \vdash a \equiv a' : A \qquad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \equiv b[a'/x] : B[a/x]}$$

When B is a family of types over A and $a : A$, we also say that $B[a/x]$ is the **fiber** of B at a. We will usually write $B(a)$ for $B[a/x]$.

Weakening

If we are given a type A in context Γ, then any judgment made in a longer context Γ, Δ can also be made in the context $\Gamma, x : A, \Delta$, for a fresh variable x. The **weakening rule** asserts that weakening by a type A in context preserves well-formedness and judgmental equality of types and terms.

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, \Delta \vdash \mathcal{J}}{\Gamma, x : A, \Delta \vdash \mathcal{J}}\, W_A$$

This process of expanding the context by a fresh variable of type A is called **weakening** (by A).

In the simplest situation where weakening applies, we have two types A and B in context Γ. Then we can weaken B by A as follows

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma, x : A \vdash B \text{ type}}\, W_A$$

in order to form the type B in context $\Gamma, x : A$. The type B in context $\Gamma, x : A$ is called the **constant family** B, or the **trivial family** B.

The variable rule

If we are given a type A in context Γ, then we can weaken A by itself to obtain that A is a type in context $\Gamma, x : A$. The **variable rule** now asserts that any hypothetical term $x : A$ in context Γ is a well-formed term of type A in context $\Gamma, x : A$.

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma, x : A \vdash x : A}\, \delta_A$$

One of the reasons for including the variable rule is that it provides an *identity function* on the type A in context Γ.

1.3 Derivations

A derivation in type theory is a tree in which each node is a valid rule of inference. We give two examples of derivations: a derivation showing that any variable can be changed to a fresh one, and a derivation showing that any two variables that do not depend on one another can be swapped in order.

Thus, we will see some examples of new inference rules that can be derived using the rules of type theory. Such inference rules are called **admissible**. Since derivations tend to get long and unwieldy, we declare that admissible inference rules are also valid to be used in derivations.

Changing variables

Variables can always be changed to fresh variables. We show that this is the case by showing that the inference rule

$$\frac{\Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, x' : A, \Delta[x'/x] \vdash \mathcal{J}[x'/x]}\, x'/x$$

is admissible, where x' is a variable that does not occur in the context $\Gamma, x : A, \Delta$.

Indeed, we have the following derivation using substitution, weakening, and the variable rule:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A\ \text{type}}{\Gamma, x' : A \vdash x' : A}\,\delta_A \qquad \dfrac{\Gamma \vdash A\ \text{type} \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, x' : A, x : A, \Delta \vdash \mathcal{J}}\,W_A}{\Gamma, x' : A, \Delta[x'/x] \vdash \mathcal{J}[x'/x]}\,S_{x'}$$

In this derivation it is the application of the weakening rule where we have to check that x' does not occur in the context $\Gamma, x : A, \Delta$.

Interchanging variables

The **interchange rule** states that if we have two types A and B in context Γ, and we make a judgment in context $\Gamma, x : A, y : B, \Delta$, then we can make that same judgment in context $\Gamma, y : B, x : A, \Delta$ where the order of $x : A$ and $y : B$ is swapped. More formally, the interchange rule is the following inference rule

$$\frac{\Gamma \vdash B\ \text{type} \qquad \Gamma, x : A, y : B, \Delta \vdash \mathcal{J}}{\Gamma, y : B, x : A, \Delta \vdash \mathcal{J}}$$

Just as the rule for changing variables, we claim that the interchange rule is an admissible rule.

The idea of the derivation for the interchange rule is as follows: If we have a judgment

$$\Gamma, x : A, y : B, \Delta \vdash \mathcal{J},$$

then we can change the variable y to a fresh variable y' and weaken the judgment to obtain the judgment

$$\Gamma, y : B, x : A, y' : B, \Delta[y'/y] \vdash \mathcal{J}[y'/y].$$

Now we can substitute y for y' to obtain the desired judgment $\Gamma, y : B, x : A, \Delta \vdash \mathcal{J}$. The formal derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash B\ \text{type}}{\Gamma, y : B \vdash y : B}\,\delta_B}{\Gamma, y : B, x : A \vdash y : B}\,W_{W_B(A)} \qquad \dfrac{\dfrac{\Gamma, x : A, y : B, \Delta \vdash \mathcal{J}}{\Gamma, x : A, y' : B, \Delta[y'/y] \vdash \mathcal{J}[y'/y]}\,y'/y}{\Gamma, y : B, x : A, y' : B, \Delta[y'/y] \vdash \mathcal{J}[y'/y]}\,W_B}{\Gamma, y : B, x : A, \Delta \vdash \mathcal{J}}\,S_{W_A(y)}$$

Exercises

1.1 Give a derivation for the following **term conversion rule**:

$$\frac{\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma \vdash a : A}{\Gamma \vdash a : A'}$$

1.2 Consider a type A in context Γ. In this exercise we establish a correspondence between types in context $\Gamma, x : A$, and uniform choices of types B_a, where a ranges over terms of A in a uniform way. A similar connection is made for terms.

(a) We define a **uniform family** over A to consist of a type

$$\Delta, \Gamma \vdash B_a \text{ type}$$

for every context Δ, and every term $\Delta, \Gamma \vdash a : A$, subject to the condition that one can derive

$$\frac{\Delta \vdash d : D \qquad \Delta, y : D, \Gamma \vdash a : A}{\Delta, \Gamma \vdash B_a[d/y] \equiv B_{a[d/y]} \text{ type}}$$

Define a bijection between the set of types in context $\Gamma, x : A$ modulo judgmental equality, and the set of uniform families over A modulo judgmental equality.

(b) Consider a type $\Gamma, x : A \vdash B$. We define a **uniform term** of B over A to consist of a type

$$\Delta, \Gamma \vdash b_a : B[a/x] \text{ type}$$

for every context Δ, and every term $\Delta, \Gamma \vdash a : A$, subject to the condition that one can derive

$$\frac{\Delta \vdash d : D \qquad \Delta, y : D, \Gamma \vdash a : A}{\Delta, \Gamma \vdash b_a[d/y] \equiv b_{a[d/y]} : B[a/x][d/y]}$$

Define a bijection between the set of terms of B in context $\Gamma, x : A$ modulo judgmental equality, and the set of uniform terms of B over A modulo judgmental equality.

2 Dependent function types

A fundamental concept in dependent type theory is that of a dependent function. A dependent function is a function of which the type of the output may depend on the input. They are a generalization of ordinary functions, because an ordinary function $f : A \to B$ is a function of which the output $f(x)$ has type B regardless of the value of x.

2.1 Dependent function types

Consider a section b of a family B over A in context Γ, i.e.,

$$\Gamma, x : A \vdash b(x) : B(x).$$

From one point of view, such a section b is an operation, or a program, that takes as input $x : A$ and produces a term $b(x) : B(x)$. From a more mathematical point of view we see b as a choice of an element of each $B(x)$. In other words, we may see b as a function that takes $x : A$ to $b(x) : B(x)$. Note that the type $B(x)$ of the output is dependent on $x : A$. In this section we postulate rules for the *type* of all such dependent functions: whenever B is a family over A in context Γ, there is a type

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

in context Γ, consisting of all the dependent functions of which the output at $x : A$ has type $B(x)$. There are four principal rules for Π-types:

(i) The formation rule, which tells us how we may form dependent function types.

(ii) The introduction rule, which tells us how to introduce new terms of dependent function types.

(iii) The elimination rule, which tells us how to use arbitrary terms of dependent function types.

(iv) The computation rules, which tell us how the introduction and elimination rules interact. These computation rules guarantee that every term of a dependent function type behaves as expected: as a dependent function.

In the cases of the formation rule, the introduction rule, and the elimination rule, we will also provide conversion rules that ensure that all the constructions respect judgmental equality.

The Π-formation rule

Dependent function types are formed by the following Π**-formation rule**:

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \prod_{(x:A)} B(x) \text{ type}}\ \Pi.$$

With the following conversion rule we postulate that formation of dependent function types respects judgmental equality of types:

$$\frac{\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma, x : A \vdash B(x) \equiv B'(x) \text{ type}}{\Gamma \vdash \prod_{(x:A)} B(x) \equiv \prod_{(x:A')} B'(x) \text{ type}}\ \Pi\text{-eq}.$$

Furthermore, when x' is a fresh variable, i.e., which does not occur in the context $\Gamma, x : A$, we also postulate that

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \prod_{(x:A)} B(x) \equiv \prod_{(x':A)} B(x') \text{ type}}\ \Pi\text{-}x'/x.$$

The Π-introduction rule

The introduction rule for dependent function types is also called the λ-abstraction rule. Recall that dependent functions are formed from terms $b(x)$ of type $B(x)$ in context $\Gamma, x : A$. Therefore the **λ-abstraction rule** is as follows:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.\, b(x) : \prod_{(x:A)} B(x)}\ \lambda$$

Just like ordinary mathematicians, we will sometimes write $x \mapsto f(x)$ for a function f. The map $n \mapsto n^2$ is an example. The λ-abstraction is also required to respect judgmental equality. Therefore we postulate the **λ-conversion rule**, which asserts that

$$\frac{\Gamma, x : A \vdash b(x) \equiv b'(x) : B(x)}{\Gamma \vdash \lambda x.\, b(x) \equiv \lambda x.\, b'(x) : \prod_{(x:A)} B(x)}\ \lambda\text{-eq.}$$

The Π-elimination rule

The elimination rule for dependent function types provides us with a way to *use* dependent functions. The way to use a dependent function is to apply it to an argument of the domain type. The Π-elimination rule is therefore also called the **evaluation rule**. It asserts that given a dependent function $f : \prod_{(x:A)} B(x)$ in context Γ we obtain a term $f(x)$ of type $B(x)$ in context $\Gamma, x : A$. More formally:

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma, x : A \vdash f(x) : B(x)}\ ev$$

Again we require that evaluation respects judgmental equality:

$$\frac{\Gamma \vdash f \equiv f' : \prod_{(x:A)} B(x)}{\Gamma, x : A \vdash f(x) \equiv f'(x) : B(x)}$$

The Π-computation rules

The computation rules for dependent function types postulate that λ-abstraction rule and the evaluation rule are mutual inverses. Thus we have two computation rules.

First we postulate the **β-rule**

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma, x : A \vdash (\lambda y.b(y))(x) \equiv b(x) : B(x)}\ \beta.$$

Second, we postulate the **η-rule**

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma \vdash \lambda x.\, f(x) \equiv f : \prod_{(x:A)} B(x)}\ \eta.$$

This completes the specification of dependent function types.

2.2 Ordinary function types

In the case where both A and B are types in context Γ, we may first weaken B by A, and then apply the formation rule for the dependent function type:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\dfrac{\Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \prod_{(x:A)} B \text{ type}}}$$

The result is the type of functions that take an argument of type A, and return a term of type B. In other words, terms of the type $\prod_{(x:A)} B$ are *ordinary* functions from A to B. We write $A \to B$ for the **type of functions** from A to B. Sometimes we will also write B^A for the type $A \to B$.

We give a brief summary of the rules specifying ordinary function types, omitting the conversion rules. All of these rules can be derived easily from the corresponding rules for Π-types.

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B \text{ type}} \to$$

$$\frac{\Gamma \vdash B \text{ type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma \vdash \lambda x.\, b(x) : A \to B}\, \lambda \qquad \frac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B}\, ev$$

$$\frac{\Gamma \vdash B \text{ type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma, x : A \vdash (\lambda y.\, b(y))(x) \equiv b(x) : B}\, \beta \qquad \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x.\, f(x) \equiv f : A \to B}\, \eta$$

2.3 The identity function, composition, and their laws

Definition 2.3.1. For any type A in context Γ, we define the **identity function** $\mathsf{id}_A : A \to A$ using the variable rule:

$$\frac{\dfrac{\Gamma \vdash A \text{ type}}{\Gamma, x : A \vdash x : A}}{\Gamma \vdash \mathsf{id}_A :\equiv \lambda x.\, x : A \to A}$$

Note that we have used the symbol $\equiv$ in the conclusion to define the identity function. A judgment of the form $\Gamma \vdash a :\equiv b : A$ should be read as "b is a well-defined term of type A in context Γ, and we will refer to it as a".

Definition 2.3.2. For any three types A, B, and C in context Γ, there is a **composition** operation

$$\mathsf{comp} : (B \to C) \to ((A \to B) \to (A \to C)),$$

i.e., we can derive

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type} \qquad \Gamma \vdash C \text{ type}}{\Gamma \vdash \mathsf{comp} : (B \to C) \to ((A \to B) \to (A \to C))}$$

We will write $g \circ f$ for $\mathsf{comp}(g, f)$.

Construction. The idea of the definition is to define $\mathsf{comp}(g, f)$ to be the function $\lambda x.\, g(f(x))$. The derivation we use to construct comp is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash B\ \mathsf{type}}{\Gamma, f : B^A, x : A \vdash f(x) : B}}{\Gamma, g : C^B, f : B^A, x : A \vdash f(x) : B} \qquad \dfrac{\dfrac{\Gamma \vdash B\ \mathsf{type} \qquad \Gamma \vdash C\ \mathsf{type}}{\Gamma, g : C^B, y : B \vdash g(y) : C}}{\Gamma, g : C^B, f : B^A, y : B \vdash g(y) : C}}{\Gamma, g : C^B, f : B^A, x : A, y : B \vdash g(y) : C}}{\Gamma, g : C^B, f : B^A, x : A \vdash g(f(x)) : C}}{\Gamma, g : C^B, f : B^A \vdash \lambda x.\, g(f(x)) : C^A}}{\Gamma, g : B \to C \vdash \lambda f.\, \lambda x.\, g(f(x)) : B^A \to C^A}}{\Gamma \vdash \mathsf{comp} :\equiv \lambda g.\, \lambda f.\, \lambda x.\, g(f(x)) : C^B \to (B^A \to C^A)}$$

□

The rules of function types can be used to derive the laws of a category for functions, i.e., we can derive that function composition is associative and that the identity function satisfies the unit laws. In the remainder of this section we will give these derivations.

Lemma 2.3.3. *Composition of functions is associative, i.e., we can derive*

$$\dfrac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to C \qquad \Gamma \vdash h : C \to D}{\Gamma \vdash (h \circ g) \circ f \equiv h \circ (g \circ f) : A \to D}$$

Proof. The main idea of the proof is that both $((h \circ g) \circ f)(x)$ and $(h \circ (g \circ f))(x)$ evaluate to $h(g(f(x))$, and therefore $(h \circ g) \circ f$ and $h \circ (g \circ f)$ must be judgmentally equal. This idea is made formal in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B} \qquad \dfrac{\dfrac{\Gamma \vdash g : B \to C}{\Gamma, y : B \vdash g(y) : C}}{\Gamma, x : A, y : B \vdash g(y) : C}}{\Gamma, x : A \vdash g(f(x)) : C} \qquad \dfrac{\dfrac{\Gamma \vdash h : C \to D}{\Gamma, z : C \vdash h(z) : D}}{\Gamma, x : A, z : C \vdash h(z) : D}}{\Gamma, x : A \vdash h(g(f(x))) : D}}{\Gamma, x : A \vdash h(g(f(x))) \equiv h(g(f(x))) : D}}{\Gamma, x : A \vdash (h \circ g)(f(x)) \equiv h((g \circ f)(x)) : D}}{\Gamma, x : A \vdash ((h \circ g) \circ f)(x) \equiv (h \circ (g \circ f))(x) : D}}{\Gamma \vdash (h \circ g) \circ f \equiv h \circ (g \circ f) : A \to D.}$$

□

Lemma 2.3.4. *Composition of functions satisfies the left and right unit laws, i.e., we can derive*

$$\dfrac{\Gamma \vdash f : A \to B}{\Gamma \vdash \mathsf{id}_B \circ f \equiv f : A \to B}$$

and

$$\dfrac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \circ \mathsf{id}_A \equiv f : A \to B}$$

Proof. The derivation for the left unit law is

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B} \qquad \dfrac{\dfrac{\Gamma \vdash B \text{ type}}{\Gamma, y : B \vdash \mathsf{id}_B(y) \equiv y : B}}{\Gamma, x : A, y : B \vdash \mathsf{id}_B(y) \equiv y : B}}{\Gamma, x : A \vdash \mathsf{id}_B(f(x)) \equiv f(x) : B}}{\dfrac{\Gamma, x : A \vdash (\mathsf{id}_B \circ f)(x) \equiv f(x) : B}{\Gamma \vdash \mathsf{id}_B \circ f \equiv f : A \to B}}$$

The right unit law is left as Exercise 2.1. □

Exercises

2.1 Give a derivation for the right unit law of Lemma 2.3.4.

2.2 Show that the rule

$$\dfrac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.\, b(x) \equiv \lambda x'.\, b(x') : \prod_{(x:A)} B(x)}\ \lambda\text{-}x'/x$$

is admissible for any variable x' that does not occur in the context $\Gamma, x : A$.

2.3 (a) Construct the **constant function**

$$\dfrac{\Gamma \vdash A \text{ type}}{\Gamma, y : B \vdash \mathsf{const}_y : A \to B}$$

(b) Show that

$$\dfrac{\Gamma \vdash f : A \to B}{\Gamma, z : C \vdash \mathsf{const}_z \circ f \equiv \mathsf{const}_z : A \to C}$$

(c) Show that

$$\dfrac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash g : B \to C}{\Gamma, y : B \vdash g \circ \mathsf{const}_y \equiv \mathsf{const}_{g(y)} : A \to C}$$

2.4 In this exercise we generalize the composition operation of non-dependent function types:

(a) Define a composition operation for dependent function types

$$\dfrac{\Gamma \vdash f : \prod_{(x:A)} B(x) \qquad \Gamma \vdash g : \prod_{(x:A)} \prod_{(y:B(x))} C(x, y)}{\Gamma \vdash g \circ' f : \prod_{(x:A)} C(x, f(x))}$$

and show that this operation agrees with ordinary composition when it is specialized to non-dependent function types.

(b) Show that composition of dependent functions agrees with ordinary composition of functions:

$$\dfrac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash (\lambda x.\, g) \circ' f \equiv g \circ f : A \to C}$$

(c) Show that composition of dependent functions is associative.

(d) Show that composition of dependent functions satisfies the right unit law:

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma \vdash (\lambda x.\, f) \circ' \mathsf{id}_A \equiv f : \prod_{(x:A)} B(x)}$$

(e) Show that composition of dependent functions satisfies the left unit law:

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma \vdash (\lambda x.\, \mathsf{id}_{B(x)}) \circ' f \equiv f : \prod_{(x:A)} B(x)}$$

2.5 (a) Given two types A and B in context Γ, and a type C in context $\Gamma, x : A, y : B$, define the **swap function**

$$\Gamma \vdash \sigma : \left(\prod_{(x:A)} \prod_{(y:B)} C(x,y)\right) \to \left(\prod_{(y:B)} \prod_{(x:A)} C(x,y)\right)$$

that swaps the order of the arguments.

(b) Show that

$$\Gamma \vdash \sigma \circ \sigma \equiv \mathsf{id} : \left(\prod_{(x:A)} \prod_{(y:B)} C(x,y)\right) \to \left(\prod_{(x:A)} \prod_{(y:B)} C(x,y)\right).$$

3 The natural numbers

The set of natural numbers is the most important object in mathematics. We quote Bishop, from his Constructivist Manifesto, the first chapter in Foundations of Constructive Analysis [1], where he gives a colorful illustration of its importance to mathematics.

> "The primary concern of mathematics is number, and this means the positive integers. We feel about number the way Kant felt about space. The positive integers and their arithmetic are presupposed by the very nature of our intelligence and, we are tempted to believe, by the very nature of intelligence in general. The development of the theory of the positive integers from the primitive concept of the unit, the concept of adjoining a unit, and the process of mathematical induction carries complete conviction. In the words of Kronecker, the positive integers were created by God. Kronecker would have expressed it even better if he had said that the positive integers were created by God for the benefit of man (and other finite beings). Mathematics belongs to man, not to God. We are not interested in properties of the positive integers that have no descriptive meaning for finite man. When a man proves a positive integer to exist, he should show how to find it. If God has mathematics of his own that needs to be done, let him do it himself."

A bit later in the same chapter, he continues:

> "Building on the positive integers, weaving a web of ever more sets and ever more functions, we get the basic structures of mathematics: the rational number system, the real number system, the euclidean spaces, the complex number system, the algebraic number fields, Hilbert space, the classical groups, and so forth. Within the framework of these structures, most mathematics is done. Everything attaches itself to number, and every mathematical statement ultimately expresses the fact that if we perform certain computations within the set of positive integers, we shall get certain results."

3.1 The formal specification of the type of natural numbers

The type $\mathbb{N}$ of **natural numbers** is the archetypal example of an inductive type. The rules we postulate for the type of natural numbers come in four sets, just as the rules for Π-types:

(i) The formation rule, which asserts that the type $\mathbb{N}$ can be formed.

(ii) The introduction rules, which provide the zero element and the successor function.

(iii) The elimination rule. This rule is the type theoretic analogue of the induction principle for $\mathbb{N}$.

(iv) The computation rules, which assert that any application of the elimination rule behaves as expected on the constructors $0_{\mathbb{N}}$ and $\mathsf{succ}_{\mathbb{N}}$ of $\mathbb{N}$.

The formation rule of $\mathbb{N}$

The type $\mathbb{N}$ is formed by the $\mathbb{N}$**-formation** rule

$$\frac{}{\vdash \mathbb{N}\ \text{type.}}\ \mathbb{N}\text{-form}$$

In other words, $\mathbb{N}$ is postulated to be a closed type.

The introduction rules of $\mathbb{N}$

Unlike the set of positive integers in Bishop's remarks, Peano's first axiom postulates that 0 is a natural number. The introduction rules for $\mathbb{N}$ equip it with the **zero term** and the **successor function**.

$$\frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}} \qquad\qquad \frac{}{\vdash \mathsf{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

Remark 3.1.1. We annotate the terms $0_{\mathbb{N}}$ and $\mathsf{succ}_{\mathbb{N}}$ of type $\mathbb{N}$ with their type in the subscript, as a reminder that $0_{\mathbb{N}}$ and $\mathsf{succ}_{\mathbb{N}}$ are declared to be terms of type $\mathbb{N}$, and not of any other type. In the next chapter we will introduce the type $\mathbb{Z}$ of the integers, on which we can also define a zero term $0_{\mathbb{Z}}$, and a successor function $\mathsf{succ}_{\mathbb{Z}}$. These should be distinguished from the terms $0_{\mathbb{N}}$ and $\mathsf{succ}_{\mathbb{N}}$. In general, we will make sure that every term is given a unique name. In libraries of mathematics formalized in a computer proof assistant it is also the case that every type must be given a unique name.

The elimination rule of $\mathbb{N}$

To prove properties about the natural numbers, we postulate an *induction principle* for $\mathbb{N}$. For a typical example, it is easy to show by induction that

$$1 + \cdots + n = \frac{n(n+1)}{2}.$$

Similarly, we can define operations by recursion on the natural numbers: the Fibonacci sequence is defined by $F(0) = 0$, $F(1) = 1$, and

$$F(n+2) = F(n) + F(n+1).$$

Needless to say, we want an induction principle to hold for the natural numbers in type theory and we also want it to be possible to construct operations on the natural numbers by recursion.

In dependent type theory we may think of a type family P over $\mathbb{N}$ as a *predicate* over $\mathbb{N}$. Especially after we introduce a few more type-forming operations, such as Σ-types and identity types, it will become clear that the language of dependent type theory expressive enough to find definitions of all of the standard concepts and operations of elementary number theory in type theory. Many of those definitions, the ordering relations $\leq$ and $<$ for example, will make use of type dependency. Then, to prove that $P(n)$ 'holds' for all n we just have to construct a dependent function

$$\prod_{(n:\mathbb{N})} P(n).$$

The induction principle for the natural numbers in type theory exactly states what one has to do in order to construct such a dependent function, via the following inference rule:

$$\frac{\begin{array}{l}\Gamma, n : \mathbb{N} \vdash P(n) \text{ type} \\ \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \\ \Gamma \vdash p_S : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n))\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(p_0, p_S) : \prod_{(n:\mathbb{N})} P(n)}\ \mathbb{N}\text{-ind}$$

Just like for the usual induction principle of the natural numbers, there are two things to be constructed given a type family P over $\mathbb{N}$: in the **base case** we need to construct a term $p_0 : P(0_{\mathbb{N}})$, and for the **inductive step** we need to construct a function of type $P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n))$ for all $n : \mathbb{N}$. And this comes at one immediate advantage: induction and recursion in type theory are one and the same thing!

Remark 3.1.2. We might alternatively present the induction principle of $\mathbb{N}$ as the following inference rule

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) \text{ type}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}} : P(0_{\mathbb{N}}) \to \left(\left(\prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n))\right) \to \prod_{(n:\mathbb{N})} P(n)\right)}$$

In other words, for any type family P over $\mathbb{N}$ there is a *function* $\mathsf{ind}_{\mathbb{N}}$ that takes two arguments, one for the base case and one for the inductive step, and returns a section of P. Now it is justified to wonder: is this slightly different presentation of induction equivalent to the previous presentation?

To see that indeed we get such a function from the induction principle (rule $\mathbb{N}$-ind above), we note that the induction principle is stated to hold in an *arbitrary* context Γ. So let us wield the power of type dependency: by weakening and the variable rule we have the following well-formed terms:

$$\Gamma,\ p_0 : P(0_{\mathbb{N}}),\ p_S : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n)) \vdash p_0 : P(0_{\mathbb{N}})$$
$$\Gamma,\ p_0 : P(0_{\mathbb{N}}),\ p_S : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n)) \vdash p_S : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n)).$$

Therefore, the induction principle of $\mathbb{N}$ provides us with a term

$$\Gamma,\ p_0 : P(0_{\mathbb{N}}),\ p_S : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n)) \vdash \mathsf{ind}_{\mathbb{N}}(p_0, p_S) : \prod_{(n:\mathbb{N})} P(n).$$

By λ-abstraction we now obtain a function

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

in context Γ. Therefore we see that it does not really matter whether we present the induction principle of $\mathbb{N}$ in a more verbose way as an inference rule with the base case and the inductive step as hypotheses, or as a function taking variables for the base case and the inductive step as arguments.

The computation rules of $\mathbb{N}$

The **computation rules** for $\mathbb{N}$ postulate that the dependent function $\mathsf{ind}_{\mathbb{N}}(P, p_0, p_S)$ behaves as expected when it is applied to $0_{\mathbb{N}}$ or a successor. There is one computation rule for each step in the induction principle, covering the base case and the inductive step.

The computation rule for the base case is

$$\frac{\begin{array}{l}\Gamma, n : \mathbb{N} \vdash P(n) \text{ type} \\ \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \\ \Gamma \vdash p_S : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n))\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}(p_0, p_S, 0_{\mathbb{N}}) \equiv p_0 : P(0_{\mathbb{N}})}$$

Similarly, with the same hypotheses as for the computation rule for the base case, the computation rule for the inductive step is

$$\frac{\ldots}{\Gamma, n : \mathbb{N} \vdash \mathsf{ind}_{\mathbb{N}}(p_0, p_S, \mathsf{succ}_{\mathbb{N}}(n)) \equiv p_S(n, \mathsf{ind}_{\mathbb{N}}(p_0, p_S, n)) : P(\mathsf{succ}_{\mathbb{N}}(n))}$$

This completes the formal specification of $\mathbb{N}$.

3.2 Addition on the natural numbers

Using the induction principle of $\mathbb{N}$ we can perform many familiar constructions. For instance, we can define the **addition operation** by induction on $\mathbb{N}$.

Definition 3.2.1. We define a function

$$\mathsf{add}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

satisfying $\mathsf{add}_{\mathbb{N}}(0_{\mathbb{N}}, n) \equiv n$ and $\mathsf{add}_{\mathbb{N}}(\mathsf{succ}_{\mathbb{N}}(m), n) \equiv \mathsf{succ}_{\mathbb{N}}(\mathsf{add}_{\mathbb{N}}(m, n))$. Usually we will write $n + m$ for $\mathsf{add}_{\mathbb{N}}(n, m)$.

Informal construction. Informally, the definition of addition is as follows. By induction it suffices to construct a function $\mathsf{add\text{-}0}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$, and a function

$$\mathsf{add\text{-}succ}_{\mathbb{N}}(n, f) : \mathbb{N} \to \mathbb{N},$$

for every $n : \mathbb{N}$ and every $f : \mathbb{N} \to \mathbb{N}$.

The function $\mathsf{add\text{-}0}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ is of course taken to be $\mathsf{id}_{\mathbb{N}}$, since the result of adding 0 to n should be n.

Given $n : \mathbb{N}$ and a function $f : \mathbb{N} \to \mathbb{N}$ we define $\mathsf{add\text{-}succ}_{\mathbb{N}}(n, f) :\equiv \mathsf{succ}_{\mathbb{N}} \circ f$. The idea is that if f represents adding m, then $\mathsf{add\text{-}succ}_{\mathbb{N}}(n, f)$ should be adding one more than f did. □

Formal derivation. The derivation for the construction of $\mathsf{add\text{-}succ}_{\mathbb{N}}$ looks as follows:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{succ}_{\mathbb{N}} : \mathbb{N}^{\mathbb{N}}} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \mathbb{N}\ \mathsf{type}} \quad \dfrac{}{\vdash \mathbb{N}\ \mathsf{type}} \quad \dfrac{}{\vdash \mathbb{N}\ \mathsf{type}}}{\vdash \mathsf{comp} : \mathbb{N}^{\mathbb{N}} \to (\mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}})}}{g : \mathbb{N} \to \mathbb{N} \vdash \mathsf{comp}(g) : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}}}{\vdash \mathsf{comp}(\mathsf{succ}_{\mathbb{N}}) : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}}}{n : \mathbb{N} \vdash \mathsf{comp}(\mathsf{succ}_{\mathbb{N}}) : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}}}{\vdash \mathsf{add\text{-}succ}_{\mathbb{N}} :\equiv \lambda n.\, \mathsf{comp}(\mathsf{succ}_{\mathbb{N}}) : \mathbb{N} \to (\mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}})}$$

We combine this derivation with the induction principle of $\mathbb{N}$ to complete the construction of addition:

$$\dfrac{\dfrac{\vdots}{n : \mathbb{N} \vdash \mathbb{N}^{\mathbb{N}}\ \mathsf{type}} \qquad \dfrac{\vdots}{\vdash \mathsf{add\text{-}0}_{\mathbb{N}} :\equiv \mathsf{id}_{\mathbb{N}} : \mathbb{N}^{\mathbb{N}}} \qquad \dfrac{\vdots}{\vdash \mathsf{add\text{-}succ}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}})}}{\vdash \mathsf{add}_{\mathbb{N}} \equiv \mathsf{ind}_{\mathbb{N}}(\mathsf{add\text{-}0}_{\mathbb{N}}, \mathsf{add\text{-}succ}_{\mathbb{N}}) : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}}$$

The asserted judgmental equalities then hold by the computation rules for $\mathbb{N}$. □

Remark 3.2.2. When we define a function $f : \prod_{(n:\mathbb{N})} P(n)$, we will often do so just by indicating its definition on $0_{\mathbb{N}}$ and its definition on $\mathsf{succ}_{\mathbb{N}}(n)$, by writing

$$f(0_{\mathbb{N}}) :\equiv p_0$$
$$f(\mathsf{succ}_{\mathbb{N}}(n)) :\equiv p_S(n, f(n)).$$

For example, the definition of addition on the natural numbers could be given as

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

This way of defining a function is called *pattern matching*. A more formal inductive argument can be obtained from a definition by pattern matching if it is possible to obtain from the expression $p_S(n, f(n))$ a general dependent function

$$p_S : \prod_{(n:\mathbb{N})} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n)).$$

In practice this is usually the case. Computer proof assistants such as Agda have sophisticated algorithms to allow for definitions by pattern matching.

Remark 3.2.3. By the computation rules for $\mathbb{N}$ it follows that

$$0_{\mathbb{N}} + n \equiv n, \qquad \text{and} \qquad \mathsf{succ}_{\mathbb{N}}(m) + n \equiv \mathsf{succ}_{\mathbb{N}}(m + n).$$

However, the rules that we provided so far are not sufficient to also conclude that $n + 0_{\mathbb{N}} \equiv n$ and $n + \mathsf{succ}_{\mathbb{N}}(m) \equiv \mathsf{succ}_{\mathbb{N}}(n + m)$. Nevertheless, once we have introduced the *identity type* in §5 we will nevertheless be able to *identify* $n + 0_{\mathbb{N}}$ with n, and $n + \mathsf{succ}_{\mathbb{N}}(m)$ with $\mathsf{succ}_{\mathbb{N}}(n + m)$. See Exercise 5.5.

Exercises

3.1 Define the binary **min** and **max** functions

$$\mathsf{min}_{\mathbb{N}}, \mathsf{max}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}).$$

3.2 Define the **multiplication** operation

$$\mathsf{mul}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}).$$

3.3 Define the **exponentiation function** $n, m \mapsto m^n$ of type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$.
3.4 Define the **factorial** operation $n \mapsto n!$.
3.5 Define the **binomial coefficient** $\binom{n}{k}$ for any $n, k : \mathbb{N}$, making sure that $\binom{n}{k} \equiv 0$ when $n < k$.
3.6 Define the **Fibonacci sequence** $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ as a function $F : \mathbb{N} \to \mathbb{N}$.

4 More inductive types

Analogous to the type of natural numbers, many types can be specified as inductive types. In this lecture we introduce some further examples of inductive types: the unit type, the empty type, the booleans, coproducts, dependent pair types, and cartesian products. We also introduce the type of integers.

4.1 The idea of general inductive types

Just like the type of natural numbers, other inductive types are also specified by their *constructors*, an *induction principle*, and their *computation rules*:

(i) The constructors tell what structure the inductive type comes equipped with. There may any finite number of constructors, even no constructors at all, in the specification of an inductive type.

(ii) The induction principle specifies the data that should be provided in order to construct a section of an arbitrary type family over the inductive type.

(iii) The computation rules assert that the inductively defined section agrees on the constructors with the data that was used to define the section. Thus, there is a computation rule for every constructor.

The induction principle and computation rules can be generated automatically once the constructors are specified, but it goes beyond the scope of our course to describe general inductive types.

4.2 The unit type

A straightforward example of an inductive type is the *unit type*, which has just one constructor. Its induction principle is analogous to just the base case of induction on the natural numbers.

Definition 4.2.1. We define the **unit type** to be a closed type $\mathbf{1}$ equipped with a closed term

$$\star : \mathbf{1},$$

satisfying the induction principle that for any type family of types $P(x)$ indexed by $x : \mathbf{1}$, there is a term

$$\mathsf{ind}_{\mathbf{1}} : P(\star) \to \textstyle\prod_{(x:\mathbf{1})} P(x)$$

for which the computation rule

$$\mathsf{ind}_{\mathbf{1}}(p, \star) \equiv p$$

holds. Sometimes we write $\lambda \star . p$ for $\mathsf{ind}_{\mathbf{1}}(p)$.

The induction principle can also be used to define ordinary functions out of the unit type. Indeed, given a type A we can first weaken it to obtain the constant family over $\mathbf{1}$, with value A. Then the induction principle of the unit type provides a function

$$\mathsf{ind}_{\mathbf{1}} : A \to (\mathbf{1} \to A).$$

In other words, by the induction principle for the unit type we obtain for every $x : A$ a function $\mathsf{pt}_x :\equiv \mathsf{ind}_{\mathbf{1}}(x) : \mathbf{1} \to A$.

4.3 The empty type

The empty type is a degenerate example of an inductive type. It does *not* come equipped with any constructors, and therefore there are also no computation rules. The induction principle merely asserts that any type family has a section. In other words: if we assume the empty type has a term, then we can prove anything.

Definition 4.3.1. We define the **empty type** to be a type $\emptyset$ satisfying the induction principle that for any family of types $P(x)$ indexed by x :, there is a term

$$\mathsf{ind}_{\emptyset} : \textstyle\prod_{(x:\emptyset)} P(x).$$

The induction principle for the empty type can also be used to construct a function

$$\emptyset \to A$$

for any type A. Indeed, to obtain this function one first weakens A to obtain the constant family over $\emptyset$ with value A, and then the induction principle gives the desired function.

Thus we see that from the empty type anything follows. Therefore, we we see that anything follows from A, if we have a function from A to the empty type. This motivates the following definition.

Definition 4.3.2. For any type A we define **negation** of A by

$$\neg A :\equiv A \to \emptyset.$$

Since $\neg A$ is the type of functions from A to $\emptyset$, a proof of $\neg A$ is given by assuming that A holds, and then deriving a contradiction. This proof technique is called **proof of negation**. Proofs of negation are not to be confused with *proofs by contradiction*. In type theory there is no way of obtaining a term of type A from a term of type $(A \to \emptyset) \to \emptyset$.

4.4 The booleans

Definition 4.4.1. We define the **booleans** to be a type $\mathbf{2}$ that comes equipped with

$$0_{\mathbf{2}} : \mathbf{2}$$
$$1_{\mathbf{2}} : \mathbf{2}$$

satisfying the induction principle that for any family of types $P(x)$ indexed by $x : \mathbf{2}$, there is a term

$$\mathsf{ind}_{\mathbf{2}} : P(0_{\mathbf{2}}) \to \left(P(1_{\mathbf{2}}) \to \textstyle\prod_{(x:\mathbf{2})} P(x)\right)$$

for which the computation rules

$$\mathsf{ind}_{\mathbf{2}}(p_0, p_1, 0_{\mathbf{2}}) \equiv p_0$$
$$\mathsf{ind}_{\mathbf{2}}(p_0, p_1, 1_{\mathbf{2}}) \equiv p_1$$

hold.

Just as in the cases for the unit type and the empty type, the induction principle for the booleans can also be used to construct an ordinary function $\mathbf{2} \to A$, provided that we can construct two terms of type A. Indeed, by the induction principle for the booleans there is a function

$$\mathsf{ind}_{\mathbf{2}} : A \to (A \to A^{\mathbf{2}})$$

for any type A.

Example 4.4.2. Using the induction principle of $\mathbf{2}$ we can define all the operations of Boolean algebra. For example, the **boolean negation** operation $\mathsf{neg}_{\mathbf{2}} : \mathbf{2} \to \mathbf{2}$ is defined by

$$\mathsf{neg}_{\mathbf{2}}(1_{\mathbf{2}}) :\equiv 0_{\mathbf{2}} \qquad \mathsf{neg}_{\mathbf{2}}(0_{\mathbf{2}}) :\equiv 1_{\mathbf{2}}.$$

The **boolean conjunction** operation $- \wedge - : \mathbf{2} \to (\mathbf{2} \to \mathbf{2})$ is defined by

$$1_{\mathbf{2}} \wedge 1_{\mathbf{2}} :\equiv 1_{\mathbf{2}} \qquad 0_{\mathbf{2}} \wedge 1_{\mathbf{2}} :\equiv 0_{\mathbf{2}}$$
$$1_{\mathbf{2}} \wedge 0_{\mathbf{2}} :\equiv 0_{\mathbf{2}} \qquad 0_{\mathbf{2}} \wedge 0_{\mathbf{2}} :\equiv 0_{\mathbf{2}}.$$

The **boolean disjunction** operation $- \vee - : \mathbf{2} \to (\mathbf{2} \to \mathbf{2})$ is defined by

$$1_{\mathbf{2}} \vee 1_{\mathbf{2}} :\equiv 1_{\mathbf{2}} \qquad 0_{\mathbf{2}} \vee 1_{\mathbf{2}} :\equiv 1_{\mathbf{2}}$$
$$1_{\mathbf{2}} \vee 0_{\mathbf{2}} :\equiv 1_{\mathbf{2}} \qquad 0_{\mathbf{2}} \vee 0_{\mathbf{2}} :\equiv 0_{\mathbf{2}}.$$

We leave the definitions of some of the other boolean operations as Exercise 4.3. Note that the method of defining the boolean operations by the induction principle of $\mathbf{2}$ is not that different from defining them by truth tables.

Boolean logic is important, but it won't be very prominent in this course. The reason is simple: in type theory it is more natural to use the 'logic' of types that is provided by the inference rules.

4.5 Coproducts and the type of integers

Definition 4.5.1. Let A and B be types. We define the **coproduct** $A + B$ to be a type that comes equipped with

$$\begin{aligned}\mathsf{inl} &: A \to A + B\\ \mathsf{inr} &: B \to A + B\end{aligned}$$

satisfying the induction principle that for any family of types $P(x)$ indexed by $x : A + B$, there is a term

$$\mathsf{ind}_+ : \left(\prod_{(x:A)} P(\mathsf{inl}(x))\right) \to \left(\prod_{(y:B)} P(\mathsf{inr}(y))\right) \to \prod_{(z:A+B)} P(z)$$

for which the computation rules

$$\begin{aligned}\mathsf{ind}_+(f, g, \mathsf{inl}(x)) &\equiv f(x)\\ \mathsf{inr}{+}(f, g, \mathsf{inr}(y)) &\equiv g(y)\end{aligned}$$

hold. Sometimes we write $[f, g]$ for $\mathsf{ind}_+(f, g)$.

The coproduct of two types is sometimes also called the **disjoint sum**. By the induction principle of coproducts it follows that we have a function

$$(A \to X) \to \big((B \to X) \to (A + B \to X)\big)$$

for any type X. Note that this special case of the induction principle of coproducts is very much like the elimination rule of disjunction in first order logic: if P, P', and Q are propositions, then we have

$$(P \to Q) \to \big((P' \to Q) \to (P \vee P' \to Q)\big).$$

Indeed, we can think of *propositions as types* and of terms as their constructive proofs. Under this interpretation of type theory the coproduct is indeed the disjunction.

An important example of a type that can be defined using coproducts is the type $\mathbb{Z}$ of integers.

Definition 4.5.2. We define the **integers** to be the type $\mathbb{Z} :\equiv \mathbb{N} + (\mathbf{1} + \mathbb{N})$. The type of integers comes equipped with inclusion functions of the positive and negative integers

$$\begin{aligned}\mathsf{in\text{-}pos} &:\equiv \mathsf{inr} \circ \mathsf{inr}\\ \mathsf{in\text{-}neg} &:\equiv \mathsf{inl},\end{aligned}$$

which are both of type $\mathbb{N} \to \mathbb{Z}$, and the constants

$$\begin{aligned}-1_{\mathbb{Z}} &:\equiv \mathsf{in\text{-}neg}(0)\\ 0_{\mathbb{Z}} &:\equiv \mathsf{inr}(\mathsf{inl}(\star))\\ 1_{\mathbb{Z}} &:\equiv \mathsf{in\text{-}pos}(0).\end{aligned}$$

In the following lemma we derive an induction principle for $\mathbb{Z}$, which can be used in many familiar constructions on $\mathbb{Z}$, such as in the definitions of addition and multiplication.

Lemma 4.5.3. *Consider a type family P over $\mathbb{Z}$. If we are given*

$$\begin{aligned} p_{-1} &: P(-1_{\mathbb{Z}}) \\ p_{-S} &: \textstyle\prod_{(n:\mathbb{N})} P(\mathsf{in\text{-}neg}(n)) \to P(\mathsf{in\text{-}neg}(\mathsf{succ}_{\mathbb{N}}(n))) \\ p_0 &: P(0_{\mathbb{Z}}) \\ P_1 &: P(1_{\mathbb{Z}}) \\ P_S &: \textstyle\prod_{(n:\mathbb{N})} P(\mathsf{in\text{-}pos}(n)) \to P(\mathsf{in\text{-}pos}(\mathsf{succ}_{\mathbb{N}}(n))), \end{aligned}$$

then we can construct a dependent function $f : \prod_{(k:\mathbb{Z})} P(k)$ for which the following judgmental equalities hold:

$$\begin{aligned} f(-1_{\mathbb{Z}}) &\equiv p_{-1} \\ f(\mathsf{in\text{-}neg}(\mathsf{succ}_{\mathbb{N}}(n))) &\equiv p_{-S}(n, f(\mathsf{in\text{-}neg}(n))) \\ f(0_{\mathbb{Z}}) &\equiv p_0 \\ f(1_{\mathbb{Z}}) &\equiv p_1 \\ f(\mathsf{in\text{-}pos}(\mathsf{succ}_{\mathbb{N}}(n))) &\equiv p_S(n, f(\mathsf{in\text{-}pos}(n))). \end{aligned}$$

Proof. Since $\mathbb{Z}$ is the coproduct of $\mathbb{N}$ and $\mathbf{1} + \mathbb{N}$, it suffices to define

$$\begin{aligned} p_{inl} &: \textstyle\prod_{(n:\mathbb{N})} P(\mathsf{inl}(n)) \\ p_{inr} &: \textstyle\prod_{(t:\mathbf{1}+\mathbb{N})} P(\mathsf{inr}(t)). \end{aligned}$$

Note that $\mathsf{in\text{-}neg} \equiv \mathsf{inl}$ and $-1_{\mathbb{Z}} \equiv \mathsf{in\text{-}neg}(0_{\mathbb{N}})$. In order to define p_{inl} we use induction on the natural numbers, so it suffices to define

$$\begin{aligned} p_{-1} &: P(-1) \\ p_{-S} &: \textstyle\prod_{(n:\mathbb{N})} P(\mathsf{in\text{-}neg}(n)) \to P(\mathsf{in\text{-}neg}(\mathsf{succ}_{\mathbb{N}}(n))). \end{aligned}$$

Similarly, we proceed by coproduct induction, followed by induction on $\mathbf{1}$ in the left case and induction on $\mathbb{N}$ on the right case, in order to define p_{inr}. □

As an application we define the successor function on the integers.

Definition 4.5.4. We define the **successor function** on the integers $\mathsf{succ}_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z}$ using the induction principle of Lemma 4.5.3, taking

$$\begin{aligned} \mathsf{succ}_{\mathbb{Z}}(-1_{\mathbb{Z}}) &:\equiv 0_{\mathbb{N}} \\ \mathsf{succ}_{\mathbb{Z}}(\mathsf{in\text{-}neg}(\mathsf{succ}_{\mathbb{N}}(n))) &:\equiv \mathsf{in\text{-}neg}(n) \\ \mathsf{succ}_{\mathbb{Z}}(0_{\mathbb{Z}}) &:\equiv 1_{\mathbb{N}} \\ \mathsf{succ}_{\mathbb{Z}}(1_{\mathbb{Z}}) &:\equiv \mathsf{in\text{-}pos}(1_{\mathbb{N}}) \\ \mathsf{succ}_{\mathbb{Z}}(\mathsf{in\text{-}pos}(\mathsf{succ}_{\mathbb{N}}(n))) &:\equiv \mathsf{in\text{-}pos}(\mathsf{succ}_{\mathbb{N}}(\mathsf{succ}_{\mathbb{N}}(n))). \end{aligned}$$

4.6 Dependent pair types

Given a type family B over A, we may consider pairs (a, b) of terms, where $a : A$ and $b : B(a)$. Note that the type of b depends on the first term in the pair, so we call such a pair a **dependent pair**.

The *dependent pair type* is an inductive type that is generated by the dependent pairs.

Definition 4.6.1. Consider a type family B over A. The **dependent pair type** (or Σ-type) is defined to be the inductive type $\sum_{(x:A)} B(x)$ equipped with a **pairing function**

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

The induction principle for $\sum_{(x:A)} B(x)$ asserts that for any family of types $P(p)$ indexed by $p : \sum_{(x:A)} B(x)$, there is a function

$$\mathsf{ind}_\Sigma : \Big(\prod_{(x:A)} \prod_{(y:B(x))} P(x,y) \Big) \to \Big(\prod_{(p:\sum_{(x:A)} B(x))} P(p) \Big).$$

satisfying the computation rule

$$\mathsf{ind}_\Sigma(f, (x,y)) \equiv f(x,y).$$

Sometimes we write $\lambda(x,y).\, f(x,y)$ for $\mathsf{ind}_\Sigma(\lambda x.\, \lambda y.\, f(x,y))$.

Definition 4.6.2. Given a type A and a type family B over A, the **first projection map**

$$\mathsf{pr}_1 : \Big(\sum_{(x:A)} B(x) \Big) \to A$$

is defined by induction as

$$\mathsf{pr}_1 :\equiv \lambda(x,y).\, x.$$

The **second projection map** is a dependent function

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

defined by induction as

$$\mathsf{pr}_2 :\equiv \lambda(x,y).\, y.$$

By the computation rule we have

$$\mathsf{pr}_1(x,y) \equiv x$$
$$\mathsf{pr}_2(x,y) \equiv y.$$

4.7 Cartesian products

A special case of the Σ-type occurs when the B is a constant family over A, i.e., when B is just a type. In this case, the inductive type $\sum_{(x:A)} B(x)$ is generated by *ordinary* pairs (x,y) where $x : A$ and $y : B$. In other words, if B does not depend on A, then the type $\sum_{(x:A)} B$ is the *(cartesian) product* $A \times B$. The cartesian product is a very common special case of the dependent pair type, just as the type $A \to B$ of ordinary functions from $A \to B$ is a common special case of the dependent product. Therefore we provide its specification along with the induction principle for cartesian products.

Definition 4.7.1. Consider two types A and B. The **(cartesian) product** of A and B is defined as the inductive type $A \times B$ with constructor

$$(-,-) : A \to (B \to A \times B).$$

The induction principle for $A \times B$ asserts that for any type family P over $A \times B$, one has

$$\mathsf{ind}_\times : \left(\prod_{(x:A)} \prod_{(y:B)} P(a,b)\right) \to \left(\prod_{(p:A\times B)} P(p)\right)$$

satisfying the computation rule that

$$\mathsf{ind}_\times(f,(x,y)) \equiv f(x,y).$$

The projection maps are defined similarly to the projection maps of Σ-types. When one thinks of types as propositions, then $A \times B$ is interpreted as the conjunction of A and B.

Exercises

4.1 Write the rules for $\mathbf{1}$, $\emptyset$, $\mathbf{2}$, $A + B$, $\sum_{(x:A)} B(x)$, and $A \times B$. As usual, present the rules in four sets:

(i) A formation rule.

(ii) Introduction rules.

(iii) An elimination rule.

(iv) Computation rules.

4.2 Let A be a type.

(a) Show that $(A + \neg A) \to (\neg\neg A \to A)$.
(b) Show that $\neg\neg\neg A \to \neg A$.

4.3 Define the following operations of Boolean algebra:

exclusive disjunction	$p \oplus q$
implication	$p \Rightarrow q$
if and only if	$p \Leftrightarrow q$
Peirce's arrow (neither ... nor)	$p \downarrow q$
Sheffer stroke (not both)	$p \mid q$.

Here p and q range over $\mathbf{2}$.

4.4 Define the predecessor function $\mathsf{pred}_\mathbb{Z} : \mathbb{Z} \to \mathbb{Z}$.

4.5 Define the group operations

$$\mathsf{add}_\mathbb{Z} : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$$
$$\mathsf{neg}_\mathbb{Z} : \mathbb{Z} \to \mathbb{Z},$$

and define the multiplication

$$\mathsf{mul}_\mathbb{Z} : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z}).$$

4.6 Construct a function $F : \mathbb{Z} \to \mathbb{Z}$ that extends the Fibonacci sequence to the negative integers

$$\ldots, 5, -3, 2, -1, 1, 0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

in the expected way.

4.7 Show that $\mathbf{1} + \mathbf{1}$ satisfies the same induction principle as $\mathbf{2}$, i.e., define

$$t_0 : \mathbf{1} + \mathbf{1}$$
$$t_1 : \mathbf{1} + \mathbf{1},$$

and show that for any type family P over $\mathbf{1} + \mathbf{1}$ there is a function

$$\mathsf{ind}_{\mathbf{1}+\mathbf{1}} : P(t_0) \to \left(P(t_1) \to \textstyle\prod_{(t:\mathbf{1}+\mathbf{1})} P(t)\right)$$

satisfying

$$\mathsf{ind}_{\mathbf{1}+\mathbf{1}}(p_0, p_1, t_0) \equiv p_0$$
$$\mathsf{ind}_{\mathbf{1}+\mathbf{1}}(p_0, p_1, t_1) \equiv p_1.$$

In other words, *type theory cannot distinguish between the types* $\mathbf{2}$ *and* $\mathbf{1} + \mathbf{1}$.

4.8 For any type A we can define the type $\mathsf{list}(A)$ of **lists** elements of A as the inductive type with constructors

$$\mathsf{nil} : \mathsf{list}(A)$$
$$\mathsf{cons} : A \to (\mathsf{list}(A) \to \mathsf{list}(A)).$$

(a) Write down the induction principle and the computation rules for $\mathsf{list}(A)$.

(b) Let A and B be types, suppose that $b : B$, and consider a binary operation $\mu : A \to (B \to B)$. Define a function

$$\mathsf{fold\text{-}list}(\mu) : \mathsf{list}(A) \to B$$

that iterates the operation μ, starting with $\mathsf{fold\text{-}list}(\mu, \mathsf{nil}) :\equiv b$.

(c) Define a function $\mathsf{length\text{-}list} : \mathsf{list}(A) \to \mathbb{N}$.

(d) Define a function

$$\mathsf{sum\text{-}list} : \mathsf{list}(\mathbb{N}) \to \mathbb{N}$$

that adds all the elements in a list of natural numbers.

(e) Define a function

$$\mathsf{concat\text{-}list} : \mathsf{list}(A) \to (\mathsf{list}(A) \to \mathsf{list}(A))$$

that concatenates any two lists of elements in A.

(f) Define a function

$$\mathsf{flatten\text{-}list} : \mathsf{list}(\mathsf{list}(A)) \to \mathsf{list}(A)$$

that concatenates all the lists in a lists of lists in A.

(g) Define a function $\mathsf{reverse\text{-}list} : \mathsf{list}(A) \to \mathsf{list}(A)$ that reverses the order of the elements in any list.

Table I.1: The homotopy interpretation

Type theory	*Homotopy theory*
Types	Spaces
Dependent types	Fibrations
Terms	Points
Dependent pair type	Total space
Identity type	Path fibration

5 Identity types

From the perspective of types as proof-relevant propositions, how should we think of *equality* in type theory? Given a type A, and two terms $x, y : A$, the equality $x = y$ should again be a type. Indeed, we want to *use* type theory to prove equalities. *Dependent* type theory provides us with a convenient setting for this: the equality type $x = y$ is dependent on $x, y : A$.

Then, if $x = y$ is to be a type, how should we think of the terms of $x = y$. A term $p : x = y$ witnesses that x and y are equal terms of type A. In other words $p : x = y$ is an *identification* of x and y. In a proof-relevant world, there might be many terms of type $x = y$. I.e., there might be many identifications of x and y. And, since $x = y$ is itself a type, we can form the type $p = q$ for any two identifications $p, q : x = y$. That is, since $x = y$ is a type, we may also use the type theory to prove things *about* identifications (for instance, that two given such identifications can themselves be identified), and we may use the type theory to perform constructions with them. As we will see shortly, we can give every type a groupoidal structure.

Clearly, the equality type should not just be any type dependent on $x, y : A$. Then how do we form the equality type, and what ways are there to use identifications in constructions in type theory? The answer to both these questions is that we will form the identity type as an *inductive* type, generated by just a reflexivity term providing an identification of x to itself. The induction principle then provides us with a way of performing constructions with identifications, such as concatenating them, inverting them, and so on. Thus, the identity type is equipped with a reflexivity term, and further possesses the structure that are generated by its induction principle and by the type theory. This inductive construction of the identity type is elegant, beautifully simple, but far from trivial!

The situation where two terms can be identified in possibly more than one way is analogous to the situation in *homotopy theory*, where two points of a space can be connected by possibly more than one *path*. Indeed, for any two points x, y in a space, there is a *space of paths* from x to y. Moreover, between any two paths from x to y there is a space of *homotopies* between them, and so on. This leads to the homotopy interpretation of type theory, outlined in Table I.1. The connection between homotopy theory and type theory been made precise by the construction of homotopical models of type theory, and it has led to the fruitful research area of *synthetic homotopy theory*, the subfield of *homotopy type theory* that is the topic of this course.

5.1 The inductive definition of identity types

Definition 5.1.1. Consider a type A and let $a : A$. Then we define the **identity type** of A at a as an inductive family of types $a =_A x$ indexed by $x : A$, of which the constructor is

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

The induction principle of the identity type postulates that for any family of types $P(x, p)$ indexed by $x : A$ and $p : a =_A x$, there is a function

$$\mathsf{path\text{-}ind}_a : P(a, \mathsf{refl}_a) \to \prod_{(x:A)} \prod_{(p:a=_A x)} P(x, p)$$

that satisfies $\mathsf{path\text{-}ind}_a(p, a, \mathsf{refl}_a) \equiv p$.

A term of type $a =_A x$ is also called an **identification** of a with x, and sometimes it is called a **path** from a to x. The induction principle for identity types is sometimes called **identification elimination** or **path induction**. We also write Id_A for the identity type on A, and often we write $a = x$ for the type of identifications of a with x, omitting reference to the ambient type A.

Remark 5.1.2. We see that the identity type is not just an inductive type, like the inductive types $\mathbb{N}$, $\emptyset$, and $\mathbf{1}$ for example, but it is and inductive *family* of types. Even though we have a type $a =_A x$ for any $x : A$, the constructor only provides a term $\mathsf{refl}_a : a =_A a$, identifying a with itself. The induction principle then asserts that in order to prove something about all identifications of a with some $x : A$, it suffices to prove this assertion about refl_a only. We will see in the next sections that this induction principle is strong enough to derive many familiar facts about equality, namely that it is a symmetric and transitive relation, and that all functions preserve equality.

Remark 5.1.3. Since the identity types require getting used to, we provide the formal rules for identity types. The identity type is formed by the formation rule:

$$\frac{\Gamma \vdash a : A}{\Gamma, x : A \vdash a =_A x \text{ type}}$$

The constructor of the identity type is then given by the introduction rule:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a : a =_A a}$$

The induction principle is now given by the elimination rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, p : a =_A x \vdash P(x, p) \text{ type}}{\Gamma \vdash \mathsf{path\text{-}ind}_a : P(a, \mathsf{refl}_a) \to \prod_{(x:A)} \prod_{(p:a=_A x)} P(x, p)}$$

And finally the computation rule is:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, p : a =_A x \vdash P(x, p) \text{ type}}{\Gamma \vdash \mathsf{path\text{-}ind}_a(p, a, \mathsf{refl}_a) \equiv p : P(a, \mathsf{refl}_a)}$$

Remark 5.1.4. One might wonder whether it is also possible to form the identity type at a *variable* of type A, rather than at a term. This is certainly possible: since we can form the identity type in *any* context, we can form the identity type at a variable $x : A$ as follows:

$$\frac{\Gamma, x : A \vdash x : A}{\Gamma, x : A, y : A \vdash x =_A y \text{ type}}$$

In this way we obtain the 'binary' identity type. Its constructor is then also indexed by $x : A$. We have the following introduction rule

$$\frac{\Gamma, x : A \vdash x : A}{\Gamma, x : A \vdash \mathsf{refl}_x : x =_A x}$$

and similarly we have elimination and computation rules.

5.2 The groupoidal structure of types

We show that identifications can be *concatenated* and *inverted*, which corresponds to the transitivity and symmetry of the identity type.

Definition 5.2.1. Let A be a type. We define the **concatenation** operation

$$\mathsf{concat} : \textstyle\prod_{(x,y,z:A)} (x = y) \to (y = z) \to (x = z).$$

We will write $p \cdot q$ for $\mathsf{concat}(p, q)$.

Construction. We construct the concatenation operation by path induction. It suffices to construct

$$\mathsf{concat}(\mathsf{refl}_x) : \textstyle\prod_{(z:A)} (x = z) \to (x = z).$$

Here we take $\mathsf{concat}(\mathsf{refl}_x)_z \equiv \mathsf{id}_{(x=z)}$. Explicitly, the term we have constructed is

$$\lambda x.\, \mathsf{path\text{-}ind}_x(\lambda z.\, \mathsf{id}_{(x=z)}) : \textstyle\prod_{(x,y:A)} (x = y) \to \prod_{(z:A)} (y = z) \to (x = z).$$

To obtain a term of the asserted type we need to swap the order of the arguments $p : x = y$ and $z : A$, using Exercise 2.5. □

Definition 5.2.2. Let A be a type. We define the **inverse operation**

$$\mathsf{inv} : \textstyle\prod_{(x,y:A)} (x = y) \to (y = x).$$

Most of the time we will write p^{-1} for $\mathsf{inv}(p)$.

Construction. We construct the inverse operation by path induction. It suffices to construct

$$\mathsf{inv}(\mathsf{refl}_x) : x = x,$$

for any $x : A$. Here we take $\mathsf{inv}(\mathsf{refl}_x) :\equiv \mathsf{refl}_x$. □

The next question is whether the concatenation and inverting operations on paths behave as expected. More concretely, is path concatenation associative, does it satisfy the unit laws, and is the inverse of a path indeed a two-sided inverse?

For example, in the case of associativity we are asking to compare the paths

$$(p \cdot q) \cdot r \qquad \text{and} \qquad p \cdot (q \cdot r)$$

for any $p : x = y$, $q : y = z$, and $r : z = w$ in a type A. The computation rules of path induction are not strong enough to conclude that $(p \cdot q) \cdot r$ and $p \cdot (q \cdot r)$ are judgmentally equal. However, both $(p \cdot q) \cdot r$ and $p \cdot (q \cdot r)$ are terms of the same type: they are identifications of type $x = w$. Since the identity type is a type like any other, we can ask whether there is an *identification*

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

This is a very useful idea: while it is often impossible to show that two terms of the same type are judgmentally equal, it may be the case that those two terms can be *identified*. Indeed, we identify two terms by constructing a term of the identity type, and we can use all the type theory at our disposal in order to construct such a term. In this way we can show, for example, that addition on the natural numbers or on the integers is associative and satisfies the unit laws. And indeed, here we will show that path concatenation is associative and satisfies the unit laws.

Definition 5.2.3. Let A be a type and consider three consecutive paths

$$x \overset{p}{=\!=} y \overset{q}{=\!=} z \overset{r}{=\!=} w$$

in A. We define the **associator**

$$\mathsf{assoc}(p, q, r) : (p \cdot q) \cdot r = p \cdot (q \cdot r).$$

Construction. By path induction it suffices to show that

$$\textstyle\prod_{(z:A)} \prod_{(q:x=z)} \prod_{(w:A)} \prod_{(r:z=w)} (\mathsf{refl}_x \cdot q) \cdot r = \mathsf{refl}_x \cdot (q \cdot r).$$

Let $q : x = z$ and $r : z = w$. Note that by the computation rule of the path induction principle we have a judgmental equality $\mathsf{refl}_x \cdot q \equiv q$. Therefore we conclude that

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

Similarly we have a judgmental equality $\mathsf{refl}_x \cdot (q \cdot r) \equiv q \cdot r$. Thus we see that the left-hand side and the right-hand side in

$$(\mathsf{refl}_x \cdot q) \cdot r = \mathsf{refl}_x \cdot (q \cdot r)$$

are judgmentally equal, so we can simply define $\mathsf{assoc}(\mathsf{refl}_x, q, r) :\equiv \mathsf{refl}_{q \cdot r}$. □

Definition 5.2.4. Let A be a type. We define the left and right **unit law operations**, which assigns to each $p : x = y$ the terms

$$\mathsf{left\text{-}unit}(p) : \mathsf{refl}_x \cdot p = p$$
$$\mathsf{right\text{-}unit}(p) : p \cdot \mathsf{refl}_y = p,$$

respectively.

Construction. By identification elimination it suffices to construct

$$\mathsf{left\text{-}unit}(\mathsf{refl}_x) : \mathsf{refl}_x \cdot \mathsf{refl}_x = \mathsf{refl}_x$$
$$\mathsf{right\text{-}unit}(\mathsf{refl}_x) : \mathsf{refl}_x \cdot \mathsf{refl}_x = \mathsf{refl}_x.$$

In both cases we take $\mathsf{refl}_{\mathsf{refl}_x}$. □

Definition 5.2.5. Let A be a type. We define left and right **inverse law operations**

$$\mathsf{left\text{-}inv}(p) : p^{-1} \cdot p = \mathsf{refl}_y$$
$$\mathsf{right\text{-}inv}(p) : p \cdot p^{-1} = \mathsf{refl}_x.$$

Construction. By identification elimination it suffices to construct

$$\mathsf{left\text{-}inv}(\mathsf{refl}_x) : \mathsf{refl}_x^{-1} \cdot \mathsf{refl}_x = \mathsf{refl}_x$$
$$\mathsf{right\text{-}inv}(\mathsf{refl}_x) : \mathsf{refl}_x \cdot \mathsf{refl}_x^{-1} = \mathsf{refl}_x.$$

Using the computation rules we see that

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

so we define $\mathsf{left\text{-}inv}(\mathsf{refl}_x) :\equiv \mathsf{refl}_{\mathsf{refl}_x}$. Similarly it follows from the computation rules that

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

so we again define $\mathsf{right\text{-}inv}(\mathsf{refl}_x) :\equiv \mathsf{refl}_{\mathsf{refl}_x}$. □

Remark 5.2.6. We have seen that the associator, the unit laws, and the inverse laws, are all proven by constructing an identification of identifications. And indeed, there is nothing that would stop us from considering identifications of those identifications of identifications. We can go up as far as we like in the *tower of identity types*, which is obtained by iteratively taking identity types.

The iterated identity types give types in homotopy type theory a very intricate structure. One important way of studying this structure is via the homotopy groups of types, a subject that we will gradually be working towards.

5.3 The action on paths of functions

Using the induction principle of the identity type we can show that every function preserves identifications. In other words, every function sends identified terms to identified terms. Note that this is a form of continuity for functions in type theory: if there is a path that identifies two points x and y of a type A, then there also is a path that identifies the values $f(x)$ and $f(y)$ in the codomain of f.

Definition 5.3.1. Let $f : A \to B$ be a map. We define the **action on paths** of f as an operation

$$\mathsf{ap}_f : \textstyle\prod_{(x,y:A)}(x = y) \to (f(x) = f(y)).$$

Moreover, there are operations

$$\mathsf{ap\text{-}id}_A : \textstyle\prod_{(x,y:A)}\prod_{(p:x=y)} p = \mathsf{ap}_{\mathsf{id}_A}(p)$$
$$\mathsf{ap\text{-}comp}(f,g) : \textstyle\prod_{(x,y:A)}\prod_{(p:x=y)} \mathsf{ap}_g(\mathsf{ap}_f(p)) = \mathsf{ap}_{g\circ f}(p).$$

Construction. First we define ap_f by identity elimination, taking

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

Next, we construct $\mathsf{ap\text{-}id}_A$ by identity elimination, taking

$$\mathsf{ap\text{-}id}_A(\mathsf{refl}_x) :\equiv \mathsf{refl}_{\mathsf{refl}_x}.$$

Finally, we construct $\mathsf{ap\text{-}comp}(f, g)$ by identity elimination, taking

$$\mathsf{ap\text{-}comp}(f, g, \mathsf{refl}_x) :\equiv \mathsf{refl}_{g(f(x))}.$$ □

Definition 5.3.2. Let $f : A \to B$ be a map. Then there are identifications

$$\mathsf{ap\text{-}refl}(f, x) : \mathsf{ap}_f(\mathsf{refl}_x) = \mathsf{refl}_f(x)$$
$$\mathsf{ap\text{-}inv}(f, p) : \mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$$
$$\mathsf{ap\text{-}concat}(f, p, q) : \mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$$

for every $p : x = y$ and $q : x = y$.

Construction. To construct $\mathsf{ap\text{-}refl}(f, x)$ we simply observe that $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_f(x)$, so we take

$$\mathsf{ap\text{-}refl}(f, x) :\equiv \mathsf{refl}_{\mathsf{refl}_{f(x)}}.$$

We construct $\mathsf{ap\text{-}inv}(f, p)$ by identification elimination on p, taking

$$\mathsf{ap\text{-}inv}(f, \mathsf{refl}_x) :\equiv \mathsf{refl}_{\mathsf{ap}_f(\mathsf{refl}_x)}.$$

Finally we construct $\mathsf{ap\text{-}concat}(f, p, q)$ by identification elimination on p, taking

$$\mathsf{ap\text{-}concat}(f, \mathsf{refl}_x, q) :\equiv \mathsf{refl}_{\mathsf{ap}_f(q)}.$$ □

5.4 Transport

Dependent types also come with an action on paths: the *transport* functions. Given an identification $p : x = y$ in the base type A, we can transport any term $b : B(x)$ to the fiber $B(y)$. The transport functions have many applications, which we will encounter throughout this course.

Definition 5.4.1. Let A be a type, and let B be a type family over A. We will construct a **transport** operation

$$\mathsf{tr}_B : \textstyle\prod_{(x,y:A)}(x = y) \to (B(x) \to B(y)).$$

Construction. We construct $\mathsf{tr}_B(p)$ by induction on $p : x =_A y$, taking

$$\mathsf{tr}_B(\mathsf{refl}_x) :\equiv \mathsf{id}_{B(x)}.$$ □

Thus we see that type theory cannot distinguish between identified terms x and y, because for any type family B over A one gets a term of $B(y)$ as soon as $B(x)$ has a term.

As an application of the transport function we construct the *dependent* action on paths of a dependent function $f : \prod_{(x:A)} B(x)$. Note that for such a dependent function f, and an identification $p : x =_A y$, it does not make sense to directly compare $f(x)$ and $f(y)$, since the type of $f(x)$ is $B(x)$ whereas the type of $f(y)$ is $B(y)$, which might not be exactly the same type. However, we can first *transport* $f(x)$ along p, so that we obtain the term $\mathsf{tr}_B(p, f(x))$ which is of type $B(y)$. Now we can ask whether it is the case that $\mathsf{tr}_B(p, f(x)) = f(y)$. The dependent action on paths of f establishes this identification.

Definition 5.4.2. Given a dependent function $f : \prod_{(a:A)} B(a)$ and a path $p : x = y$ in A, we construct a path

$$\mathsf{apd}_f(p) : \mathsf{tr}_B(p, f(x)) = f(y).$$

Construction. The path $\mathsf{apd}_f(p)$ is constructed by path induction on p. Thus, it suffices to construct a path

$$\mathsf{apd}_f(\mathsf{refl}_x) : \mathsf{tr}_B(\mathsf{refl}_x, f(x)) = f(x).$$

Since transporting along refl_x is the identity function on $B(x)$, we simply take $\mathsf{apd}_f(\mathsf{refl}_x) :\equiv \mathsf{refl}_{f(x)}$. □

Exercises

5.1 (a) State Goldbach's Conjecture in type theory.
(b) State the Twin Prime Conjecture in type theory.

5.2 Show that the operation inverting paths distributes over the concatenation operation, i.e., construct an identification

$$\mathsf{distributive\text{-}inv\text{-}concat}(p, q) : (p \cdot q)^{-1} = q^{-1} \cdot p^{-1}.$$

for any $p : x = y$ and $q : y = z$.

5.3 For any $p : x = y$, $q : y = z$, and $r : x = z$, construct maps

$$\mathsf{inv\text{-}con}(p, q, r) : (p \cdot q = r) \to (q = p^{-1} \cdot r)$$
$$\mathsf{con\text{-}inv}(p, q, r) : (p \cdot q = r) \to (p = r \cdot q^{-1}).$$

5.4 Let B be a type family over A, and consider a path $p : x = x'$ in A. Construct for any $y : B(x)$ a path

$$\mathsf{lift}_B(p, y) : (x, y) = (x', \mathsf{tr}_B(p, y)).$$

In other words, a path in the *base type A lifts* to a path in the total space $\sum_{(x:A)} B(x)$ for every term over the domain, analogous to the path lifting property for fibrations in homotopy theory.

5.5 Show that the operations of addition and multiplication on the natural numbers satisfy the laws of a commutative **semi-ring**:

$$\begin{aligned} m + (n + k) &= (m + n) + k & \qquad m \cdot (n \cdot k) &= (m \cdot n) \cdot k \\ m + 0 &= m & \qquad m \cdot 1 &= m \\ 0 + m &= m & \qquad 1 \cdot m &= m \end{aligned}$$

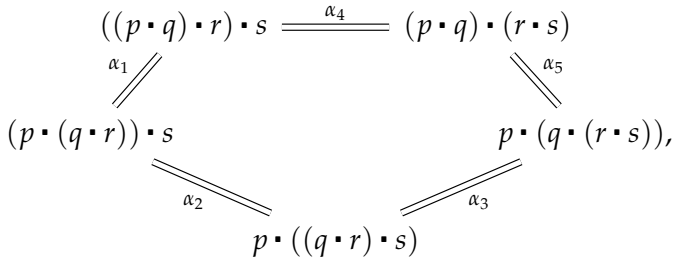
$$m + n = n + m \qquad\qquad \begin{aligned} m \cdot 0 &= 0 \\ 0 \cdot m &= 0 \\ m \cdot n &= n \cdot m \\ m \cdot (n + k) &= m \cdot n + m \cdot k. \end{aligned}$$

5.6 Consider four consecutive identifications

$$a \overset{p}{=\!=} b \overset{q}{=\!=} c \overset{r}{=\!=} d \overset{s}{=\!=} e$$

in a type A. In this exercise we will show that the **Mac Lane pentagon** for identifications commutes.

(a) Construct the five identifications $\alpha_1, \ldots, \alpha_5$ in the pentagon



where α_1, α_2, and α_3 run counter-clockwise, and α_4 and α_5 run clockwise.

(b) Show that

$$(\alpha_1 \blacksquare \alpha_2) \blacksquare \alpha_3 = \alpha_4 \blacksquare \alpha_5.$$

6 Type theoretic universes

To complete our specification of dependent type theory, we introduce type theoretic *universes*. Universes are types that consist of types. In other words, a universe is a type $\mathcal{U}$ that comes equipped with a type family $\mathcal{T}$ over $\mathcal{U}$, and for any $X : \mathcal{U}$ we think of X as an *encoding* of the type $\mathcal{T}(X)$. We call this type family the *universal type family*.

There are several reasons to equip type theory with universes. One reason is that it enables us to define new type families over inductive types, using their induction principle. For example, since the universe is itself a type, we can use the induction principle of $\mathbf{2}$ to obtain a map $P : \mathbf{2} \to \mathcal{U}$ from any two terms $X_0, X_1 : \mathcal{U}$. Then we obtain a type family over $\mathbf{2}$ by substituting P into the universal type family:

$$x : \mathbf{2} \vdash \mathcal{T}(P(x)) \text{ type}$$

satisfying $\mathcal{T}(P(0_{\mathbf{2}})) \equiv \mathcal{T}(X_0)$ and $\mathcal{T}(P(1_{\mathbf{2}})) \equiv \mathcal{T}(X_1)$.

We use this way of defining type families to define many familiar relations over $\mathbb{N}$, such as $\leq$ and $<$. We also introduce a relation called *observational equality* $\mathsf{Eq}_{\mathbb{N}}$ on $\mathbb{N}$, which we can think of as equality of $\mathbb{N}$. This relation is reflexive, symmetric, and transitive, and moreover it is the least reflexive relation. Furthermore, one of the most important aspects of observational equality $\mathsf{Eq}_{\mathbb{N}}$ on $\mathbb{N}$ is that $\mathsf{Eq}_{\mathbb{N}}(m, n)$ is a type for

every $m, n : \mathbb{N}$, unlike judgmental equality. Therefore we can use type theory to reason about observational equality on $\mathbb{N}$. Indeed, in the exercises we show that some very elementary mathematics can already be done at this early stage in our development of type theory.

A second reason to introduce universes is that it allows us to define many types of types equipped with structure. One of the most important examples is the type of groups, which is the type of types equipped with the group operations satisfying the group laws, and for which the underlying type is a set. We won't discuss the condition for a type to be a set until §10, so the definition of groups in type theory will be given much later. Therefore we illustrate this use of the universe by giving simpler examples: pointed types, graphs, and reflexive graphs.

One of the aspects that make universes useful is that they are postulated to be closed under all the type constructors. For example, if we are given $X : \mathcal{U}$ and $P : \mathcal{T}(X) \to \mathcal{U}$, then the universe is equipped with a term

$$\check{\Sigma}(X, P) : \mathcal{U}$$

satisfying the judgmental equality $\mathcal{T}(\check{\Sigma}(X, P) \equiv \sum_{(x:\mathcal{T}(X))} \mathcal{T}(P(x))$. We will similarly assume that any universe is closed under Π-types and the other ways of forming types. However, there is an important restriction: it would be inconsistent to assume that the universe is contained in itself. One way of thinking about this is that universes are types of *small* types, and it cannot be the case that the universe is small with respect to itself. We address this problem by assuming that there are many universes: enough universes so that any type family can be obtained by substituting into the universal type family of some universe.

6.1 Specification of type theoretic universes

In the following definition we already state that universes are closed under identity types. Identity types will be introduced in §5.

Definition 6.1.1. A **universe** in type theory is a closed type $\mathcal{U}$ equipped with a type family $\mathcal{T}$ over $\mathcal{U}$ called the **universal family**, equipped with the following structure:

(i) $\mathcal{U}$ is closed under Π, in the sense that it comes equipped with a function

$$\check{\Pi} : \prod_{(X:\mathcal{U})} (\mathcal{T}(X) \to \mathcal{U}) \to \mathcal{U}$$

for which the judgmental equality

$$\mathcal{T}\big(\check{\Pi}(X, P)\big) \equiv \prod_{(x:\mathcal{T}(X))} \mathcal{T}(P(x)).$$

holds, for every $X : \mathcal{U}$ and $P : \mathcal{T}(X) \to \mathcal{U}$.

(ii) $\mathcal{U}$ is closed under Σ in the sense that it comes equipped with a function

$$\check{\Sigma} : \prod_{(X:\mathcal{U})} (\mathcal{T}(X) \to \mathcal{U}) \to \mathcal{U}$$

for which the judgmental equality

$$\mathcal{T}\big(\check{\Sigma}(X, P)\big) \equiv \sum_{(x:\mathcal{T}(X))} \mathcal{T}(P(x))$$

holds, for every $X : \mathcal{U}$ and $P : \mathcal{T}(X) \to \mathcal{U}$.

(iii) $\mathcal{U}$ is closed under identity types, in the sense that it comes equipped with a function

$$\check{\mathrm{I}} : \textstyle\prod_{(X:\mathcal{U})} \mathcal{T}(X) \to (\mathcal{T}(X) \to \mathcal{U})$$

for which the judgmental equality

$$\mathcal{T}\big(\check{\mathrm{I}}(X, x, y)\big) \equiv (x = y)$$

holds, for every $X : \mathcal{U}$ and $x, y : \mathcal{T}(X)$.

(iv) $\mathcal{U}$ is closed under coproducts, in the sense that it comes equipped with a function

$$\check{+} : \mathcal{U} \to (\mathcal{U} \to \mathcal{U})$$

that satisfies $\mathcal{T}\big(X \check{+} Y\big) \equiv \mathcal{T}(X) + \mathcal{T}(Y)$.

(v) $\mathcal{U}$ contains terms $\check{\emptyset}, \check{\mathbf{1}}, \check{\mathbb{N}} : \mathcal{U}$ that satisfy the judgmental equalities

$$\begin{aligned} \mathcal{T}\big(\check{\emptyset}\big) &\equiv \emptyset \\ \mathcal{T}\big(\check{\mathbf{1}}\big) &\equiv \mathbf{1} \\ \mathcal{T}(\check{\mathbb{N}}) &\equiv \mathbb{N}. \end{aligned}$$

Given a universe $\mathcal{U}$, we say that a type A in context Γ is **small** with respect to $\mathcal{U}$ if it occurs in the universe, i.e., if it comes equipped with a term $\check{A} : \mathcal{U}$ in context Γ, for which the judgment

$$\Gamma \vdash \mathcal{T}\big(\check{A}\big) \equiv A \text{ type}$$

holds. If A is small with respect to $\mathcal{U}$, we usually write simply A for $\check{A}$ and also A for $\mathcal{T}(\check{A})$. In other words, by $A : \mathcal{U}$ we mean that A is a small type.

Remark 6.1.2. Since ordinary function types are defined as a special case of dependent function types, we don't have to assume that universes are closed under ordinary function types. Similarly, it follows from the assumption that universes are closed under dependent pair types that universes are closed under cartesian product types.

6.2 Assuming enough universes

Most of the time we will get by with assuming one universe $\mathcal{U}$, and indeed we recommend on a first reading of this text to simply assume that there is one universe $\mathcal{U}$. However, sometimes we might need a second universe $\mathcal{V}$ that contains $\mathcal{U}$ as well as all the types in $\mathcal{U}$. In such situations we cannot get by with a single universe, because the assumption that $\mathcal{U}$ is a term of itself would lead to inconsistencies like the Russel's paradox.

Russel's paradox is the famous argument that there cannot be a set of all sets. If there were such a set S, then we could consider Russel's subset

$$R := \{x \in S \mid x \notin x\}.$$

Russell then observed that $R \in R$ if and only if $R \notin R$, so we reach a contradiction. A variant of this argument reaches a similar contradiction when we assume that $\mathcal{U}$ is

a universe that contains a term $\check{\mathcal{U}} : \mathcal{U}$ such that $\mathcal{T}(\check{\mathcal{U}}) \equiv \mathcal{U}$. In order to avoid such paradoxes, Russell and Whitehead formulated the *ramified theory of types* in their book *Principia Mathematica*. The ramified theory of types is a precursor of Martin Löf's type theory that we are studying in this course.

Even though the universe is not a term of itself, it is still convenient if every type, including any universe, is small with respect to *some* universe. Therefore we will assume that there are sufficiently many universes: we will assume that for every finite list of types

$$\begin{aligned} \Gamma_1 &\vdash A_1 \text{ type} \\ &\vdots \\ \Gamma_n &\vdash A_n \text{ type,} \end{aligned}$$

there is a universe $\mathcal{U}$ that contains each A_i in the sense that $\mathcal{U}$ comes equipped with a term

$$\Gamma_i \vdash \check{A}_i : \mathcal{U}$$

for which the judgment

$$\Gamma_i \vdash \mathcal{T}(\check{A}_i) \equiv A_i \text{ type}$$

holds. With this assumption it will rarely be necessary to work with more than one universe at the same time.

Remark 6.2.1. Using the assumption that for any finite list of types in context there is a universe that contains those types, we obtain many specific universes:

(i) There is a *base universe* $\mathcal{U}_0$ that we obtain using the empty list of types in context. This is a universe, but it isn't specified to contain any further types.

(ii) Given a finite list

$$\begin{aligned} \Gamma_1 &\vdash A_1 \text{ type} \\ &\vdots \\ \Gamma_n &\vdash A_n \text{ type,} \end{aligned}$$

of types in context, and a universe $\mathcal{U}$ that contains them, there is a universe $\mathcal{U}^+$ that contains all the types in $\mathcal{U}$ as well as $\mathcal{U}$. More precisely, it is specified by the finite list

$$\begin{aligned} &\vdash \mathcal{U} \text{ type} \\ X : \mathcal{U} &\vdash \mathcal{T}(X) \text{ type.} \end{aligned}$$

Note that since the universe $\mathcal{U}^+$ contains all the types in $\mathcal{U}$, it also contains the types $A_1, \ldots, A_n$. To see this, we derive that there is a code for A_i in $\mathcal{U}^+$.

$$\frac{\Gamma_i \vdash \check{A}_i : \mathcal{U} \qquad \dfrac{X : \mathcal{U} \vdash \check{\mathcal{T}}(X) : \mathcal{U}^+}{\Gamma_i, X : \mathcal{U} \vdash \check{\mathcal{T}}(X) : \mathcal{U}^+}}{\Gamma_i \vdash \check{\mathcal{T}}(\check{A}_i) : \mathcal{U}^+}$$

We leave it as an exercise to derive the judgmental equality

$$\mathcal{T}^+(\check{\mathcal{T}}(\check{A}_i)) \equiv A_i.$$

(iii) Given two finite lists

$$\begin{array}{cc} \Gamma_1 \vdash A_1 \text{ type} & \Delta_1 \vdash B_1 \text{ type} \\ \vdots & \vdots \\ \Gamma_n \vdash A_n \text{ type} & \Delta_m \vdash B_m \text{ type} \end{array}$$

of types in context, and two universes $\mathcal{U}$ and $\mathcal{V}$ that contain $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ respectively, there is a universe $\mathcal{U} \sqcup \mathcal{V}$ that contains the types of both $\mathcal{U}$ and $\mathcal{V}$. The universe $\mathcal{U} \sqcup \mathcal{V}$ is specified by the finite list

$$\begin{aligned} X : \mathcal{U} &\vdash \mathcal{T}_{\mathcal{U}}(X) \text{ type} \\ Y : \mathcal{V} &\vdash \mathcal{T}_{\mathcal{V}}(Y) \text{ type.} \end{aligned}$$

With an argument similar to the previous construction of a universe, we see that the universe $\mathcal{U} \sqcup \mathcal{V}$ contains the types $A_1, \ldots, A_n$ as well as the types $B_1, \ldots, B_m$.

Note that we could also directly obtain a universe $\mathcal{W}$ that contains the types $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$. However, this universe might not contain all the types in $\mathcal{U}$ or all the types in $\mathcal{V}$.

Since we don't postulate any relations between the universes, there are indeed very few of them. For example, the base universe $\mathcal{U}_0$ might contain many more types than it is postulated to contain. Nevertheless, there are some relations between the universes. For instance, there is a function $\mathcal{U} \to \mathcal{U}^+$, since we can simply derive

$$\frac{X : \mathcal{U} \vdash \check{\mathcal{T}}(X) : \mathcal{U}^+}{\vdash \lambda X.\, \check{\mathcal{T}}(X) : \mathcal{U} \to \mathcal{U}^+}$$

Similarly, there are functions $\mathcal{U} \to \mathcal{U} \sqcup \mathcal{V}$ and $\mathcal{V} \to \mathcal{U} \sqcup \mathcal{V}$ for any two universes $\mathcal{U}$ and $\mathcal{V}$.

6.3 Pointed types

Definition 6.3.1. A **pointed type** is a pair (A, a) consisting of a type A and a term $a : A$. The type of all pointed types in a universe $\mathcal{U}$ is defined to be

$$\mathcal{U}_* :\equiv \textstyle\sum_{(X:\mathcal{U})} X.$$

Definition 6.3.2. Consider two pointed types (A, a) and (B, b). A **pointed map** from (A, a) to (B, b) is a pair (f, p) consisting of a function $f : A \to B$ and an identification $p : f(a) = b$. We write

$$A \to_* B :\equiv \textstyle\sum_{(f:A\to B)} f(a) = b$$

for the type of all pointed maps from (A, a) to (B, b), leaving the base point implicit.

Since we have a type $\mathcal{U}_*$ of *all* pointed types in a universe $\mathcal{U}$, we can start defining operations on $\mathcal{U}_*$. An important example of such an operation is to take the loop space of a pointed type.

Definition 6.3.3. We define the **loop space** operation $\Omega : \mathcal{U}_* \to \mathcal{U}_*$

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

We can even go further and define the *iterated loop space* of a pointed type. Note that this definition could not be given in type theory if we didn't have universes.

Definition 6.3.4. Given a pointed type (A, a) and a natural number n, we define the n-th loop space $\Omega^n(A, a)$ by induction on $n : \mathbb{N}$, taking

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

6.4 Families and relations on the natural numbers

As we have already seen in the case of the iterated loop space, we can use the universe to define a type family over $\mathbb{N}$ by induction on $\mathbb{N}$. For example, we can define the finite types in this way.

Definition 6.4.1. We define the type family $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$ of finite types by induction on $\mathbb{N}$, taking

$$\mathsf{Fin}(0_{\mathbb{N}}) :\equiv \emptyset$$
$$\mathsf{Fin}(\mathsf{succ}_{\mathbb{N}}(n)) :\equiv \mathsf{Fin}(n) + \mathbf{1}$$

Similarly, we can define many relations on the natural numbers using a universe. We give here the example of *observational equality* on $\mathbb{N}$. This inductively defined equivalence relation is very important, as it can be used to show that equality on the natural numbers is *decidable*, i.e., there is a program that decides for any two natural numbers m and n whether they are equal or not.

Definition 6.4.2. We define the **observational equality** on $\mathbb{N}$ as binary relation $\mathsf{Eq}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathcal{U})$ satisfying

$$\mathsf{Eq}_{\mathbb{N}}(0_{\mathbb{N}}, 0_{\mathbb{N}}) \equiv \mathbf{1} \qquad \mathsf{Eq}_{\mathbb{N}}(\mathsf{succ}_{\mathbb{N}}(n), 0_{\mathbb{N}}) \equiv \emptyset$$
$$\mathsf{Eq}_{\mathbb{N}}(0_{\mathbb{N}}, \mathsf{succ}_{\mathbb{N}}(n)) \equiv \emptyset \qquad \mathsf{Eq}_{\mathbb{N}}(\mathsf{succ}_{\mathbb{N}}(n), \mathsf{succ}_{\mathbb{N}}(m)) \equiv \mathsf{Eq}_{\mathbb{N}}(n, m).$$

Construction. We define $\mathsf{Eq}_{\mathbb{N}}$ by double induction on $\mathbb{N}$. By the first application of induction it suffices to provide

$$E_0 : \mathbb{N} \to \mathcal{U}$$
$$E_S : \mathbb{N} \to (\mathbb{N} \to \mathcal{U}) \to (\mathbb{N} \to \mathcal{U})$$

We define E_0 by induction, taking $E_{00} :\equiv \mathbf{1}$ and $E_{0S}(n, X, m) :\equiv \emptyset$. The resulting family E_0 satisfies

$$E_0(0_{\mathbb{N}}) \equiv \mathbf{1}$$
$$E_0(\mathsf{succ}_{\mathbb{N}}(n)) \equiv \emptyset.$$

We define E_S by induction, taking $E_{S0} :\equiv \emptyset$ and $E_{S0}(n, X, m) :\equiv X(m)$. The resulting family E_S satisfies

$$E_S(n, X, 0_{\mathbb{N}}) \equiv \emptyset$$
$$E_S(n, X, \mathsf{succ}_{\mathbb{N}}(m)) \equiv X(m)$$

Therefore we have by the computation rule for the first induction that the judgmental equality

$$\mathsf{Eq}_{\mathbb{N}}(0_{\mathbb{N}}, m) \equiv E_0(m)$$
$$\mathsf{Eq}_{\mathbb{N}}(\mathsf{succ}_{\mathbb{N}}(n), m) \equiv E_S(n, \mathsf{Eq}_{\mathbb{N}}(n), m)$$

holds, from which the judgmental equalities in the statement of the definition follow. □

Lemma 6.4.3. *Suppose $R : \mathbb{N} \to (\mathbb{N} \to \mathcal{U})$ is a reflexive relation on $\mathbb{N}$, i.e., R comes equipped with*

$$\rho : \prod_{(n:\mathbb{N})} R(n, n).$$

Then there is a family of maps

$$\prod_{(m,n:\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(m, n) \to R(m, n).$$

Proof. We will prove by induction on $m, n : \mathbb{N}$ that there is a term of type

$$f_{m,n} : \prod_{(e:\mathsf{Eq}_{\mathbb{N}}(m,n))} \prod_{(R:\mathbb{N}\to(\mathbb{N}\to\mathcal{U}))} \left(\prod_{(x:\mathbb{N})} R(x, x) \right) \to R(m, n)$$

The dependent function $f_{m,n}$ is defined by

$$f_{0_{\mathbb{N}},0_{\mathbb{N}}} :\equiv \lambda \star . \lambda r. \lambda \rho. \rho(0_{\mathbb{N}})$$
$$f_{0_{\mathbb{N}},\mathsf{succ}_{\mathbb{N}}(n)} :\equiv \mathsf{ind}_{\emptyset}$$
$$f_{\mathsf{succ}_{\mathbb{N}}(m),0_{\mathbb{N}}} :\equiv \mathsf{ind}_{\emptyset}$$
$$f_{\mathsf{succ}_{\mathbb{N}}(m),\mathsf{succ}_{\mathbb{N}}(n)} :\equiv \lambda e. \lambda R. \lambda \rho. f_{m,n}(e, R', \rho'),$$

where R' and ρ' are given by

$$R'(m, n) :\equiv R(\mathsf{succ}_{\mathbb{N}}(m), \mathsf{succ}_{\mathbb{N}}(n))$$
$$\rho'(n) :\equiv \rho(\mathsf{succ}_{\mathbb{N}}(n)).$$

□

We can also define observational equality for many other kinds of types, such as $\mathbf{2}$ or $\mathbb{Z}$. In each of these cases, what sets the observational equality apart from other relations is that it is the *least* reflexive relation.

Exercises

6.1 Show that observational equality on $\mathbb{N}$ is an equivalence relation, i.e., construct terms of the following types:

$$\prod_{(n:\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(n,n)$$
$$\prod_{(n,m:\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(n,m) \to \mathsf{Eq}_{\mathbb{N}}(m,n)$$
$$\prod_{(n,m,l:\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(n,m) \to (\mathsf{Eq}_{\mathbb{N}}(m,l) \to \mathsf{Eq}_{\mathbb{N}}(n,l)).$$

6.2 Let R be a reflexive binary relation on $\mathbb{N}$, i.e., R is of type $\mathbb{N} \to (\mathbb{N} \to \mathcal{U})$ and comes equipped with a term $\rho : \prod_{(n:\mathbb{N})} R(n,n)$. Show that

$$\prod_{(n,m:\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(n,m) \to R(n,m).$$

6.3 Show that every function $f : \mathbb{N} \to \mathbb{N}$ preserves observational equality in the sense that

$$\prod_{(n,m:\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(n,m) \to \mathsf{Eq}_{\mathbb{N}}(f(n),f(m)).$$

Hint: to get the inductive step going the induction hypothesis has to be strong enough. Construct by double induction a term of type

$$\prod_{(n,m:\mathbb{N})} \prod_{(f:\mathbb{N}\to\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(n,m) \to \mathsf{Eq}_{\mathbb{N}}(f(n),f(m)),$$

and pull out the universal quantification over $f : \mathbb{N} \to \mathbb{N}$ by Exercise 2.5.

6.4 (a) Define the **order relations** $\leq$ and $<$ on $\mathbb{N}$.
(b) Show that $\leq$ is reflexive and that $<$ is **anti-reflexive**, i.e., that $\neg(n < n)$.
(c) Show that both $\leq$ and $<$ are transitive, and that $n < S(n)$.
(d) Show that $k \leq \min(m,n)$ holds if and only if both $k \leq m$ and $k \leq n$ hold, and show that $\max(m,n) \leq k$ holds if and only if both $m \leq k$ and $n \leq k$ hold.

6.5 (a) Define observational equality $\mathsf{Eq}_{\mathbf{2}}$ on the booleans.
(b) Show that $\mathsf{Eq}_{\mathbf{2}}$ is reflexive.
(c) Show that for any reflexive relation $R : \mathbf{2} \to (\mathbf{2} \to \mathcal{U})$ one has

$$\prod_{(x,y:\mathbf{2})} \mathsf{Eq}_{\mathbf{2}}(x,y) \to R(x,y).$$

6.6 (a) Define the order relations $\leq$ and $<$ on and $\mathbb{Z}$.
(b) Show that $\leq$ is reflexive, transitive, and anti-symmetric.
(c) Show that $<$ is anti-reflexive and transitive.

6.7 (a) Show that $\mathbb{N}$ satisfies **strong induction**, i.e., construct for any type family P over $\mathbb{N}$ a function of type

$$P(0_{\mathbb{N}}) \to \left(\prod_{(k:\mathbb{N})} \left(\prod_{(m:\mathbb{N})} (m \leq k) \to P(m)\right) \to P(\mathsf{succ}_{\mathbb{N}}(k))\right) \to \prod_{(n:\mathbb{N})} P(n).$$

(b) Show that $\mathbb{N}$ satisfies **ordinal induction**, i.e., construct for any type family P over $\mathbb{N}$ a function of type

$$\left(\prod_{(k:\mathbb{N})} \left(\prod_{(m:\mathbb{N})} (m < k) \to P(m)\right) \to P(k)\right) \to \prod_{(n:\mathbb{N})} P(n).$$

Chapter II

Basic concepts of type theory

7 Equivalences

7.1 Homotopies

In homotopy type theory, a homotopy is just a pointwise equality between two functions f and g. We view the type of homotopies as the observational equality for Π-types.

Definition 7.1.1. Let $f, g : \prod_{(x:A)} P(x)$ be two dependent functions. The type of **homotopies** from f to g is defined as

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

Note that the type of homotopies $f \sim g$ is a special case of a dependent function type. Therefore the definition of homotopies is set up in such a way that we may also consider homotopies *between* homotopies, and even further homotopies between those higher homotopies. More concretely, if $H, K : f \sim g$ are two homotopies, then the type of homotopies $H \sim K$ between them is just the type

$$\prod_{(x:A)} H(x) = K(x).$$

In the following definition we define the groupoidal structure of homotopies. Note that we implement the groupoid laws as *homotopies* rather than as identifications.

Definition 7.1.2. For any type family B over A there are operations

$$\begin{aligned}
&\mathsf{htpy\text{-}refl} && : \textstyle\prod_{(f:\prod_{(x:A)} B(x))} f \sim f \\
&\mathsf{htpy\text{-}inv} && : \textstyle\prod_{(f,g:\prod_{(x:A)} B(x))} (f \sim g) \to (g \sim f) \\
&\mathsf{htpy\text{-}concat} && : \textstyle\prod_{(f,g,h:\prod_{(x:A)} B(x))} (f \sim g) \to ((g \sim h) \to (f \sim h)).
\end{aligned}$$

We will write H^{-1} for $\mathsf{htpy\text{-}inv}(H)$, and $H \cdot K$ for $\mathsf{htpy\text{-}concat}(H, K)$.

Furthermore, we define

$$\begin{aligned}
&\mathsf{htpy\text{-}assoc}(H, K, L) && : (H \cdot K) \cdot L \sim H \cdot (K \cdot L) \\
&\mathsf{htpy\text{-}left\text{-}unit}(H) && : \mathsf{htpy\text{-}refl}_f \cdot H \sim H
\end{aligned}$$

$$\begin{aligned}
&\mathsf{htpy\text{-}right\text{-}unit}(H) && : H \cdot \mathsf{htpy\text{-}refl}_g \sim H \\
&\mathsf{htpy\text{-}left\text{-}inv}(H) && : H^{-1} \cdot H \sim \mathsf{htpy\text{-}refl}_g \\
&\mathsf{htpy\text{-}right\text{-}inv}(H) && : H \cdot H^{-1} \sim \mathsf{htpy\text{-}refl}_f
\end{aligned}$$

for any $H : f \sim g$, $K : g \sim h$ and $L : h \sim i$, where $f, g, h, i : \prod_{(x:A)} B(x)$.

Construction. We define

$$\begin{aligned}
\mathsf{htpy\text{-}refl}(f) &:\equiv \lambda x.\, \mathsf{refl}_{f(x)} \\
\mathsf{htpy\text{-}inv}(H) &:\equiv \lambda x.\, H(x)^{-1} \\
\mathsf{htpy\text{-}concat}(H, K) &:\equiv \lambda x.\, H(x) \cdot K(x),
\end{aligned}$$

where $H : f \sim g$ and $K : g \sim h$ are homotopies. Furthermore, we define

$$\begin{aligned}
\mathsf{htpy\text{-}assoc}(H, K, L) &:\equiv \lambda x.\, \mathsf{assoc}(H(x), K(x), L(x)) \\
\mathsf{htpy\text{-}left\text{-}unit}(H) &:\equiv \lambda x.\, \mathsf{left\text{-}unit}(H(x)) \\
\mathsf{htpy\text{-}right\text{-}unit}(H) &:\equiv \lambda x.\, \mathsf{right\text{-}unit}(H(x)) \\
\mathsf{htpy\text{-}left\text{-}inv}(H) &:\equiv \lambda x.\, \mathsf{left\text{-}inv}(H(x)) \\
\mathsf{htpy\text{-}right\text{-}inv}(h) &:\equiv \lambda x.\, \mathsf{right\text{-}inv}(H(x)).
\end{aligned}$$

□

Apart from the groupoid operations and their laws, we will occasionally need *whiskering* operations.

Definition 7.1.3. We define the following **whiskering** operations on homotopies:

(i) Suppose $H : f \sim g$ for two functions $f, g : A \to B$, and let $h : B \to C$. We define

$$h \cdot H :\equiv \lambda x.\, \mathsf{ap}_h(H(x)) : h \circ f \sim h \circ g.$$

(ii) Suppose $f : A \to B$ and $H : g \sim h$ for two functions $g, h : B \to C$. We define

$$H \cdot f :\equiv \lambda x.\, H(f(x)) : h \circ f \sim g \circ f.$$

We also use homotopies to express the commutativity of diagrams. For example, we say that a triangle

$$\begin{array}{ccc}
A & \xrightarrow{h} & B \\
& {\scriptstyle f}\searrow \quad \swarrow{\scriptstyle g} & \\
& X &
\end{array}$$

commutes if it comes equipped with a homotopy $H : f \sim g \circ h$, and we say that a square

$$\begin{array}{ccc}
A & \xrightarrow{g} & A' \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f'} \\
B & \xrightarrow[h]{} & B'
\end{array}$$

if it comes equipped with a homotopy $h \circ f \; g \circ f'$.

7.2 Bi-invertible maps

Definition 7.2.1. Let $f : A \to B$ be a function. We say that f has a **section** if there is a term of type

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

Dually, we say that f has a **retraction** if there is a term of type

$$\mathsf{retr}(f) :\equiv \textstyle\sum_{(h:B\to A)} h \circ f \sim \mathsf{id}_A.$$

If a map $f : A \to B$ has a retraction, we also say that A is a **retract** of B. We say that a function $f : A \to B$ is an **equivalence** if it has both a section and a retraction, i.e., if it comes equipped with a term of type

$$\mathsf{is\text{-}equiv}(f) :\equiv \mathsf{sec}(f) \times \mathsf{retr}(f).$$

We will write $A \simeq B$ for the type $\sum_{(f:A\to B)} \mathsf{is\text{-}equiv}(f)$.

Remark 7.2.2. An equivalence, as we defined it here, can be thought of as a *bi-invertible map*, since it comes equipped with a separate left and right inverse. Explicitly, if f is an equivalence, then there are

$$\begin{aligned} g &: B \to A & h &: B \to A \\ G &: f \circ g \sim \mathsf{id}_B & H &: h \circ f \sim \mathsf{id}_A. \end{aligned}$$

Clearly, if f has an inverse in the sense that it comes equipped with a function $g : B \to A$ such that $f \circ g \sim \mathsf{id}_B$ and $g \circ f \sim \mathsf{id}_A$, then f is an equivalence. We write

$$\mathsf{has\text{-}inverse}(f) :\equiv \textstyle\sum_{(g:B\to A)} (f \circ g \sim \mathsf{id}_B) \times (g \circ f \sim \mathsf{id}_A).$$

Lemma 7.2.3. *Any equivalence $e : A \simeq B$ can be given the structure of an invertible map. We define e^{-1} to be the section $g : B \to A$ of e.*

Proof. First we construct for any equivalence f with right inverse g and left inverse h a homotopy $K : g \sim h$. For any $y : B$, we have

$$g(y) \xrightarrow{H(g(y))^{-1}} hfg(y) \xrightarrow{\mathsf{ap}_h(G(y))} h(y).$$

Therefore we define a homotopy $K : g \sim h$ by $K :\equiv (H \cdot g)^{-1} \cdot h \cdot G$. Using the homotopy K we are able to show that g is also a left inverse of f. For $x : A$ we have the identification

$$gf(x) \xrightarrow{K(f(x))} hf(x) \xrightarrow{H(x)} x. \qquad \square$$

Corollary 7.2.4. *The inverse of an equivalence is again an equivalence.*

Proof. Let $f : A \to B$ be an equivalence. By Lemma 7.2.3 it follows that the section of f is also a retraction. Therefore it follows that the section is itself an invertible map, with inverse f. Hence it is an equivalence. $\square$

Remark 7.2.5. For any type A, the identity function id_A is an equivalence, since it is its own section and its own retraction

Example 7.2.6. For any type $C(x,y)$ indexed by $x : A$ and $y : B$, the swap function

$$\sigma : \left(\prod_{(x:A)}\prod_{(y:B)} C(x,y)\right) \to \left(\prod_{(y:B)}\prod_{(x:A)} C(x,y)\right)$$

that swaps the order of the arguments x and y is an equivalence by Exercise 2.5.

7.3 The identity type of a Σ-type

In this section we characterize the identity type of a Σ-type as a Σ-type of identity types. In this course we will be characterizing the identity types of many types, so we will follow the general outline of how such a characterization goes:

(i) First we define a binary relation $R : A \to A \to \mathcal{U}$ on the type A that we are interested in. This binary relation is intended to be equivalent to its identity type.

(ii) Then we will show that this binary relation is reflexive, by constructing a term of type

$$\prod_{(x:A)} R(x,x)$$

(iii) Using the reflexivity we will show that there is a canonical map

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

for every $x, y : A$. This map is just constructed by path induction, using the reflexivity of R.

(iv) Finally, it has to be shown that the map

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

is an equivalence for each $x, y : A$.

The last step is usually the most difficult, and we will refine our methods for this step in §9, where we establish the fundamental theorem of identity types.

In this section we consider a type family B over A. Given two pairs

$$(x,y), (x',y') : \textstyle\sum_{(x:A)} B(x),$$

if we have a path $\alpha : x = x'$ then we can compare $y : B(x)$ to $y' : B(x')$ by first transporting y along α, i.e., we consider the identity type

$$\mathsf{tr}_B(\alpha, y) = y'.$$

Thus it makes sense to think of (x,y) to be identical to (x',y') if there is an identification $\alpha : x = x'$ and an identification $\beta : \mathsf{tr}_B(\alpha, y) = y'$. In the following definition we turn this idea into a binary relation on the Σ-type.

Definition 7.3.1. We will define a relation

$$\mathsf{Eq}_\Sigma : \left(\textstyle\sum_{(x:A)} B(x)\right) \to \left(\textstyle\sum_{(x:A)} B(x)\right) \to \mathcal{U}$$

by defining

$$\mathsf{Eq}_\Sigma(s,t) :\equiv \textstyle\sum_{(\alpha:\mathsf{pr}_1(s)=\mathsf{pr}_1(t))} \mathsf{tr}_B(\alpha, \mathsf{pr}_2(s)) = \mathsf{pr}_2(t).$$

Lemma 7.3.2. *The relation* Eq_Σ *is reflexive, i.e., there is a term*

$$\mathsf{reflexive\text{-}Eq}_\Sigma : \textstyle\prod_{(s:\sum_{(x:A)} B(x))} \mathsf{Eq}_\Sigma(s,s).$$

Construction. This term is constructed by Σ-induction on $s : \sum_{(x:A)} B(x)$. Thus, it suffices to construct a term of type

$$\textstyle\prod_{(x:A)} \prod_{(y:B(x))} \sum_{(\alpha:x=x)} \mathsf{tr}_B(\alpha, y) = y.$$

Here we take $\lambda x.\, \lambda y.\, (\mathsf{refl}_x, \mathsf{refl}_y)$. □

Definition 7.3.3. Consider a type family B over A. Then for any $s, t : \sum_{(x:A)} B(x)$ we define a map

$$\mathsf{pair\text{-}eq} : (s = t) \to \mathsf{Eq}_\Sigma(s,t)$$

by path induction, taking $\mathsf{pair\text{-}eq}(\mathsf{refl}_s) :\equiv \mathsf{reflexive\text{-}Eq}_\Sigma(s)$.

Theorem 7.3.4. *Let B be a type family over A. Then the map*

$$\mathsf{pair\text{-}eq} : (s = t) \to \mathsf{Eq}_\Sigma(s,t)$$

is an equivalence for every $s, t : \sum_{(x:A)} B(x)$.

Proof. The maps in the converse direction

$$\mathsf{eq\text{-}pair} : \mathsf{Eq}_\Sigma(s,t) \to (s = t)$$

are defined by repeated Σ-induction. By Σ-induction on s and t we see that it suffices to define a map

$$\mathsf{eq\text{-}pair} : \left(\textstyle\sum_{(p:x=x')} \mathsf{tr}_B(p,y) = y'\right) \to ((x,y) = (x',y')).$$

A map of this type is again defined by Σ-induction. Thus it suffices to define a dependent function of type

$$\textstyle\prod_{(p:x=x')} (\mathsf{tr}_B(p,y) = y') \to ((x,y) = (x',y')).$$

Such a dependent function is defined by double path induction by sending $(\mathsf{refl}_x, \mathsf{refl}_y)$ to $\mathsf{refl}_{(x,y)}$. This completes the definition of the function eq-pair.

Next, we must show that eq-pair is a section of pair-eq. In other words, we must construct an identification

$$\mathsf{pair\text{-}eq}(\mathsf{eq\text{-}pair}(\alpha,\beta)) = (\alpha,\beta)$$

for each $(\alpha, \beta) : \sum_{(\alpha:x=x')} \mathsf{tr}_B(\alpha, y) = y'$. We proceed by path induction on α, followed by path induction on β. Then our goal becomes to construct a term of type

$$\mathsf{pair\text{-}eq}(\mathsf{eq\text{-}pair}(\mathsf{refl}_x, \mathsf{refl}_y)) = (\mathsf{refl}_x, \mathsf{refl}_y)$$

By the definition of eq-pair we have $\mathsf{eq\text{-}pair}(\mathsf{refl}_x, \mathsf{refl}_y) \equiv \mathsf{refl}_{(x,y)}$, and by the definition of pair-eq we have $\mathsf{pair\text{-}eq}(\mathsf{refl}_{(x,y)}) \equiv (\mathsf{refl}_x, \mathsf{refl}_y)$. Thus we may take $\mathsf{refl}_{(\mathsf{refl}_x, \mathsf{refl}_y)}$ to complete the construction of the homotopy $\mathsf{pair\text{-}eq} \circ \mathsf{eq\text{-}pair} \sim \mathsf{id}$.

To complete the proof, we must show that eq-pair is a retraction of pair-eq. In other words, we must construct an identification

$$\mathsf{eq\text{-}pair}(\mathsf{pair\text{-}eq}(p)) = p$$

for each $p : s = t$. We proceed by path induction on $p : s = t$, so it suffices to construct an identification

$$\mathsf{eq\text{-}pair}(\mathsf{refl}_{\mathsf{pr}_1(s)}, \mathsf{refl}_{\mathsf{pr}_2(s)}) = \mathsf{refl}_s.$$

Now we proceed by Σ-induction on $s : \sum_{(x:A)} B(x)$, so it suffices to construct an identification

$$\mathsf{eq\text{-}pair}(\mathsf{refl}_x, \mathsf{refl}_y) = \mathsf{refl}_{(x,y)}.$$

Since $\mathsf{eq\text{-}pair}(\mathsf{refl}_x, \mathsf{refl}_y)$ computes to $\mathsf{refl}_{(x,y)}$, we may simply take $\mathsf{refl}_{\mathsf{refl}_{(x,y)}}$. □

Exercises

7.1 Show that the functions

$$\begin{aligned} \mathsf{inv} &: (x = y) \to (y = x) \\ \mathsf{concat}(p) &: (y = z) \to (x = z) \\ \mathsf{concat}'(q) &: (x = y) \to (x = z) \\ \mathsf{tr}_B(p) &: B(x) \to B(y) \end{aligned}$$

are equivalences, where $\mathsf{concat}'(q, p) :\equiv p \cdot q$. Give their inverses explicitly.

7.2 Show that the maps

$$\begin{aligned} \mathsf{inl} &: X \to X + \emptyset & \mathsf{pr}_1 &: \emptyset \times X \to \emptyset \\ \mathsf{inr} &: X \to \emptyset + X & \mathsf{pr}_2 &: X \times \emptyset \to \emptyset \end{aligned}$$

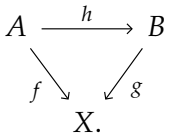
are equivalences.

7.3 (a) Consider two functions $f, g : A \to B$ and a homotopy $H : f \sim g$. Then

$$\mathsf{is\text{-}equiv}(f) \leftrightarrow \mathsf{is\text{-}equiv}(g).$$

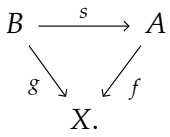
(b) Show that for any two homotopic equivalences $e, e' : A \simeq B$, their inverses are also homotopic.

7.4 Consider a commuting triangle



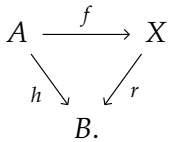
with $H : f \sim g \circ h$.

(a) Suppose that the map h has a section $s : B \to A$. Show that the triangle



commutes, and that f has a section if and only if g has a section.

(b) Suppose that the map g has a retraction $r : X \to B$. Show that the triangle



commutes, and that f has a retraction if and only if h has a retraction.

(c) (The **3-for-2 property** for equivalences.) Show that if any two of the functions

$$f, \qquad g, \qquad h$$

are equivalences, then so is the third.

7.5 (a) Show that the negation function on the booleans $\mathsf{neg}_{\mathbf{2}} : \mathbf{2} \to \mathbf{2}$ defined in Example 4.4.2 is an equivalence.

(b) Use the observational equality on the booleans, defined in Exercise 6.5, to show that $0_{\mathbf{2}} \neq 1_{\mathbf{2}}$.

(c) Show that for any $b : \mathbf{2}$, the constant function const_b is not an equivalence.

7.6 Show that the successor function on the integers is an equivalence.

7.7 Construct a equivalences $A + B \simeq B + A$ and $A \times B \simeq B \times A$.

7.8 Consider a section-retraction pair

$$A \xrightarrow{\ i\ } B \xrightarrow{\ r\ } A,$$

with $H : r \circ i \sim \mathsf{id}$. Show that $x = y$ is a retract of $i(x) = i(y)$.

7.9 Let B be a family of types over A, and let C be a family of types indexed by $x : A, y : B(x)$. Construct an equivalence

$$\mathsf{assoc}\text{-}\Sigma : \left(\textstyle\sum_{(p:\sum_{(x:A)} B(x))} C(\mathsf{pr}_1(p), \mathsf{pr}_2(p))\right) \simeq \left(\textstyle\sum_{(x:A)}\sum_{(y:B(x))} C(x,y)\right).$$

7.10 Let A and B be types, and let C be a family over $x : A, y : B$. Construct an equivalence

$$\mathsf{swap}\text{-}\Sigma : \left(\textstyle\sum_{(x:A)}\sum_{(y:B)} C(x,y)\right) \simeq \left(\textstyle\sum_{(y:B)}\sum_{(x:A)} C(x,y)\right).$$

7.11 In this exercise we will show that the laws for abelian groups hold for addition on the integers. Note: these are obvious facts, but the proof terms that show *how* the group laws hold are nevertheless fairly involved. This exercise is perfect for a formalization project.

(a) Show that addition satisfies the left and right unit laws, i.e., construct terms

$$\mathsf{left\text{-}unit\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x:\mathbb{Z})} 0 + x = x$$
$$\mathsf{right\text{-}unit\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x:\mathbb{Z})} x + 0 = x.$$

(b) Show that addition respects predecessors and successor on both sides, i.e., construct terms

$$\mathsf{left\text{-}predecessor\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x,y:\mathbb{Z})} \mathsf{pred}_{\mathbb{Z}}(x) + y = \mathsf{pred}_{\mathbb{Z}}(x + y)$$
$$\mathsf{right\text{-}predecessor\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x,y:\mathbb{Z})} x + \mathsf{pred}_{\mathbb{Z}}(y) = \mathsf{pred}_{\mathbb{Z}}(x + y)$$
$$\mathsf{left\text{-}successor\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x,y:\mathbb{Z})} \mathsf{succ}_{\mathbb{Z}}(x) + y = \mathsf{succ}_{\mathbb{Z}}(x + y)$$
$$\mathsf{right\text{-}successor\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x,y:\mathbb{Z})} x + \mathsf{succ}_{\mathbb{Z}}(y) = \mathsf{succ}_{\mathbb{Z}}(x + y).$$

Hint: to avoid an excessive number of cases, use induction on x but not on y. You may need to use the homotopies $\mathsf{succ}_{\mathbb{Z}} \circ \mathsf{pred}_{\mathbb{Z}} \sim \mathsf{id}$ and $\mathsf{pred}_{\mathbb{Z}} \circ \mathsf{succ}_{\mathbb{Z}}$ constructed in exercise Exercise 7.6.

(c) Use part (b) to show that addition on the integers is associative and commutative, i.e., construct terms

$$\mathsf{assoc\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x,y,z:\mathbb{Z})} (x + y) + z = x + (y + z)$$
$$\mathsf{comm\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x,y:\mathbb{Z})} x + y = y + x.$$

Hint: Especially in the construction of the associator there is a risk of running into an unwieldy amount of cases if you use $\mathbb{Z}$-induction on all arguments. Avoid induction on y and z.

(d) Show that addition satisfies the left and right inverse laws:

$$\mathsf{left\text{-}inverse\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x:\mathbb{Z})} (-x) + x = 0$$
$$\mathsf{right\text{-}inverse\text{-}law\text{-}add\text{-}}\mathbb{Z} : \textstyle\prod_{(x:\mathbb{Z})} x + (-x) = 0.$$

Conclude that the functions $y \mapsto x + y$ and $x \mapsto x + y$ are equivalences for any $x : \mathbb{Z}$ and $y : \mathbb{Z}$, respectively.

7.12 In this exercise we will construct the **functorial action** of coproducts.

(a) Construct for any two maps $f : A \to A'$ and $g : B \to B'$, a map

$$f + g : A + B \to A' + B'.$$

(b) Show that if $H : f \sim f'$ and $K : g \sim g'$, then there is a homotopy

$$H + K : (f + g) \sim (f' + g').$$

(c) Show that $\mathsf{id}_A + \mathsf{id}_B \sim \mathsf{id}_{A+B}$.
(d) Show that for any

$$A \xrightarrow{f} A' \xrightarrow{f'} A''$$

$$B \xrightarrow{g} B' \xrightarrow{g'} B''$$

there is a homotopy

$$(f' \circ f) + (g' \circ g) \sim (f' + g') \circ (f + g).$$

(e) Show that if f and g are equivalences, then so is $f + g$. (The converse of this statement also holds, see Exercise 9.5.)

7.13 Construct equivalences

$$\mathsf{Fin}(m + n) \simeq \mathsf{Fin}(m) + \mathsf{Fin}(n)$$
$$\mathsf{Fin}(mn) \simeq \mathsf{Fin}(m) \times \mathsf{Fin}(n).$$

8 Contractible types and contractible maps

A contractible type is a type which has, up to identification, only one term. In other words, a contractible type is a type that comes equipped with a point, and an identification of this point with any point.

We may think of contractible types as singletons up to homotopy, and indeed we show that the unit type is an example of a contractible type. Moreover, we show that contractible types satisfy an induction principle that is very similar to the induction principle of the unit type, provided that we formulate the computation rule using the identity type rather than postulating a judgmental computation rule. Another example of a contractible type is the total space of the family of identifications with a fixed starting point.

We then introduce the notion of fiber of a map, which is the type theoretic analogue of the pre-image of a map, and we say that a map in contractible if all its fibers are contractible. Thus, a map is contractible if the pre-image at any point in the codomain is a singleton. This condition is of course analogous to the set theoretic notion of bijective map, which suggests that on the type theoretical side of things a map should be contractible if and only if it is an equivalence.

The forward direction of this claim is straightforward, and we prove this direction immediately in Theorem 8.2.5. The converse direction can be done directly, but it is certainly more involved. Therefore we prepare the proof of the converse direction by first characterizing the identity type of a fiber of a map. Then we show that any equivalence e can be given the structure of a invertible map with an additional coherence relating the homotopies

$$e \circ e^{-1} \sim \mathsf{id}, \qquad \text{and} \qquad e^{-1} \circ e \sim \mathsf{id},$$

and finally we use these observations in Theorem 8.3.6 to conclude that the fibers of any equivalence must be contractible.

8.1 Contractible types

Definition 8.1.1. We say that a type A is **contractible** if it comes equipped with a term of type

$$\mathsf{is\text{-}contr}(A) :\equiv \sum_{(c:A)} \prod_{(x:A)} c = x.$$

Given a term $(c, C) : \mathsf{is\text{-}contr}(A)$, we call $c : A$ the **center of contraction** of A, and we call $C : \prod_{(x:A)} c = x$ the **contraction** of A.

Remark 8.1.2. Suppose A is a contractible type with center of contraction c and contraction C. Then the type of C is (judgmentally) equal to the type

$$\mathsf{const}_c \sim \mathsf{id}_A.$$

In other words, the contraction C is a *homotopy* from the constant function to the identity function.

Example 8.1.3. The unit type is easily seen to be contractible. For the center of contraction we take $\star : \mathbf{1}$. Then we define a contraction $\prod_{(x:\mathbf{1})} \star = x$ by the induction principle of $\mathbf{1}$. Applying the induction principle, it suffices to construct a term of type $\star = \star$, for which we just take $\mathsf{refl}_\star$.

Definition 8.1.4. Suppose A comes equipped with a term $a : A$. Then we say that A satisfies **singleton induction** if for every type family B over A, the map

$$\mathsf{ev\text{-}pt} : \left(\prod_{(x:A)} B(x)\right) \to B(a)$$

defined by $\mathsf{ev\text{-}pt}(f) :\equiv f(a)$ has a section. In other words, if A satisfies singleton induction we have a function and a homotopy

$$\mathsf{sing\text{-}ind}_a : B(a) \to \prod_{(x:A)} B(x)$$

$$\mathsf{sing\text{-}comp}_a : \mathsf{ev\text{-}pt} \circ \mathsf{sing\text{-}ind}_a \sim \mathsf{id}$$

for any type family B over A.

Example 8.1.5. Note that the singleton induction principle is almost the same as the induction principle for the unit type, the difference being that the "computation rule" in the singleton induction for A is stated using an *identification* rather than as a judgmental equality. The unit type $\mathbf{1}$ comes equipped with a function

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

for every type family B over $\mathbf{1}$, satisfying the judgmental equality $\mathsf{ind}_{\mathbf{1}}(b, \star) \equiv b$ for every $b : B(\star)$ by the computation rule. Thus we easily obtain the homotopy

$$\lambda b.\, \mathsf{refl}_b : \mathsf{ev\text{-}pt} \circ \mathsf{ind}_{\mathbf{1}} \sim \mathsf{id},$$

and we conclude that the unit type satisfies singleton induction.

Theorem 8.1.6. *Let A be a type. The following are equivalent:*

(i) The type A is contractible.

(ii) The type A comes equipped with a term $a : A$, and satisfies singleton induction.

Proof. Suppose A is contractible with center of contraction c and contraction C. First we observe that, without loss of generality, we may assume that C comes equipped with an identification $p : C(c) = \mathsf{refl}_c$. To see this, note that we can always define a new contraction C' by

$$C'(x) :\equiv C(c)^{-1} \cdot C(x),$$

which satisfies the requirement by the left inverse law, constructed in Definition 5.2.5.

To show that A satisfies singleton induction let B be a type family over A equipped with $b : B(a)$. To define $\mathsf{sing\text{-}ind}_a(b) : \prod_{(x:A)} B(x)$, let $x : A$. We have an identification $C(x) : a = x$, and b is in $B(a)$. Therefore we can transport b along the path $C(x)$ to obtain

$$\mathsf{sing\text{-}ind}_a(b) :\equiv \mathsf{tr}_B(C(x), b) : B(x).$$

To see that $\mathsf{sing\text{-}ind}(c) = b$ note that we have

$$\mathsf{tr}_B(C(c), b) \xrightarrow{\mathsf{ap}_{\lambda\omega.\, \mathsf{tr}_B(\omega,b)}(p)} \mathsf{tr}_B(\mathsf{refl}_c, b) \xrightarrow{\mathsf{refl}_b} b.$$

This completes the proof that A satisfies singleton induction.

For the converse, suppose that $a : A$ and that A satisfies singleton induction. Our goal is to show that A is contractible. For the center of contraction we take the term $a : A$. By singleton induction applied to $B(x) :\equiv a = x$ we have the map

$$\mathsf{sing\text{-}ind}_a : a = a \to \prod_{(x:A)} a = x.$$

Therefore $\mathsf{sing\text{-}ind}_{A,a}(\mathsf{refl}_a)$ is a contraction. □

Theorem 8.1.7. *For any $a : A$, the type*

$$\sum_{(x:A)} a = x$$

is contractible.

Proof. We will prove the statement by showing that $\sum_{(y:A)} x = y$ satisfies singleton induction, and then use Theorem 8.1.6 to conclude that $\sum_{(x:A)} a = x$ is contractible. We will use the term $(a, \mathsf{refl}_a) : \sum_{(x:A)} a = x$ as the center of contraction.

Now let P be a type family over $\sum_{(x:A)} a = x$. Note that we have a commuting triangle

$$\begin{array}{ccc} \prod_{(t:\sum_{(x:A)} a=x)} P(t) & \xrightarrow{\quad\mathsf{ev\text{-}pair}\quad} & \prod_{(x:A)} \prod_{(p:a=x)} P(x,p) \\ & {\scriptstyle \mathsf{ev\text{-}pt}} \searrow \quad \swarrow {\scriptstyle \mathsf{ev\text{-}refl}} & \\ & P(a, \mathsf{refl}_a) & \end{array}$$

where the maps ev-pair and ev-refl are defined as

$$f \mapsto \lambda x.\, \lambda p.\, f(x,p)$$

$$g \mapsto g(a, \mathsf{refl}_a),$$

respectively. By the induction principle for Σ-types it follows that ev-pair has a section, and by path induction it follows that ev-refl has a section. Therefore it follows from Exercise 7.4 that the composite ev-pt has a section. □

8.2 Contractible maps

Definition 8.2.1. Let $f : A \to B$ be a function, and let $b : B$. The **fiber** of f at b is defined to be the type

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

In other words, the fiber of f at b is the type of $a : A$ that get mapped by f to b. One may think of the fiber as a type theoretic version of the pre-image of a point.

It will be useful to have a characterization of the identity type of a fiber, so we will make such a characterization immediately.

Definition 8.2.2. Let $f : A \to B$ be a map, and let $(x, p), (x', p') : \mathsf{fib}_f(y)$ for some $y : B$. Then we define

$$\mathsf{Eq\text{-}fib}_f((x, p), (x', p')) :\equiv \textstyle\sum_{(\alpha : x = x')} p = \mathsf{ap}_f(\alpha) \cdot p'$$

The relation $\mathsf{Eq\text{-}fib}_f : \mathsf{fib}_f(y) \to \mathsf{fib}_f(y) \to \mathcal{U}$ is a reflexive relation, since we have

$$\lambda(x, p).\, (\mathsf{refl}_x, \mathsf{refl}_p) : \textstyle\prod_{((x,p):\mathsf{fib}_f(y))} \mathsf{Eq\text{-}fib}_f((x, p), (x, p)).$$

Lemma 8.2.3. *Consider a map $f : A \to B$ and let $y : B$. The canonical map*

$$((x, p) = (x', p')) \to \mathsf{Eq\text{-}fib}_f((x, p), (x', p'))$$

induced by the reflexivity of $\mathsf{Eq\text{-}fib}_f$ is an equivalence for any $(x, p), (x', p') : \mathsf{fib}_f(y)$.

Proof. The converse map

$$\mathsf{Eq\text{-}fib}_f((x, p), (x', p')) \to ((x, p) = (x', p'))$$

is easily defined by Σ-induction, and then path induction twice. The homotopies witnessing that this converse map is indeed a right inverse as well as a left inverse is similarly constructed by induction. □

Now we arrive at the notion of contractible map.

Definition 8.2.4. We say that a function $f : A \to B$ is **contractible** if there is a term of type

$$\mathsf{is\text{-}contr}(f) :\equiv \textstyle\prod_{(b:B)} \mathsf{is\text{-}contr}(\mathsf{fib}_f(b)).$$

Theorem 8.2.5. *Any contractible map is an equivalence.*

Proof. Let $f : A \to B$ be a contractible map. Using the center of contraction of each $\mathsf{fib}_f(y)$, we obtain a term of type

$$\lambda y.\, (g(y), G(y)) : \textstyle\prod_{(y:B)} \mathsf{fib}_f(y).$$

Thus, we get map $g : B \to A$, and a homotopy $G : \prod_{(y:B)} f(g(y)) = y$. In other words, we get a section of f.

It remains to construct a retraction of f. Taking g as our retraction, we have to show that $\prod_{(x:A)} g(f(x)) = x$. Note that we get an identification $p : f(g(f(x))) = f(x)$ since g is a section of f. It follows that $(g(f(x)), p) : \mathsf{fib}_f(f(x))$. Moreover, since $\mathsf{fib}_f(f(x))$ is contractible we get an identification $q : (g(f(x)), p) = (x, \mathsf{refl}_{f(x)})$. The base path $\mathsf{ap}_{\mathsf{pr}_1}(q)$ of this identification is an identification of type $g(f(x)) = x$, as desired. □

8.3 Equivalences are contractible maps

In Theorem 8.3.6 we will show the converse to Theorem 8.2.5, i.e., we will show that any equivalence is a contractible map. We will do this in two steps.

First we introduce a new notion of *coherently invertible map*, for which we can easily show that such maps have contractible fibers. Then we show that any equivalence is a coherently invertible map.

Recall that an invertible map is a map $f : A \to B$ equipped with $g : B \to A$ and homotopies

$$G : f \circ g \sim \mathsf{id} \qquad \text{and} \qquad H : g \circ f \sim \mathsf{id}.$$

Then we observe that both $G \cdot f$ and $f \cdot H$ are homotopies of the same type

$$f \circ g \circ f \sim f.$$

A coherently invertible map is an invertible map for which there is a further homotopy $G \cdot f \sim f \cdot H$.

Definition 8.3.1. Consider a map $f : A \to B$. We say that f is **coherently invertible** if it comes equipped with

$$\begin{aligned} g &: B \to A \\ G &: f \circ g \sim \mathsf{id} \\ H &: g \circ f \sim \mathsf{id} \\ K &: G \cdot f \sim f \cdot H. \end{aligned}$$

We will write $\mathsf{is\text{-}coh\text{-}invertible}(f)$ for the type of quadruples (g, G, H, K).

Although we will encounter the notion of coherently invertible map on some further occasions, the following lemma is our main motivation for considering it.

Lemma 8.3.2. *Any coherently invertible map has contractible fibers.*

Proof. Consider a map $f : A \to B$ equipped with

$$g : B \to A$$
$$G : f \circ g \sim \mathsf{id}$$
$$H : g \circ f \sim \mathsf{id}$$
$$K : G \cdot f \sim f \cdot H,$$

and let $y : B$. Our goal is to show that $\mathsf{fib}_f(y)$ is contractible. For the center of contraction we take $(g(y), G(y))$. In order to construct a contraction, it suffices to construct a term of type

$$\textstyle\prod_{(x:A)} \prod_{(p:f(x)=y)} \mathsf{Eq\text{-}fib}_f((g(y), G(y)), (x, p)).$$

By path induction on $p : f(x) = y$ it suffices to construct a term of type

$$\textstyle\prod_{(x:A)} \mathsf{Eq\text{-}fib}_f((g(f(x)), G(f(x))), (x, \mathsf{refl}_{f(x)})).$$

By definition of $\mathsf{Eq\text{-}fib}_f$, we have to construct a term of type

$$\textstyle\prod_{(x:A)} \sum_{(\alpha : g(f(x))=x)} G(f(x)) = \mathsf{ap}_f(\alpha) \cdot \mathsf{refl}_{f(x)}.$$

Such a term is constructed as $\lambda x.\, (H(x), K'(x))$, where the homotopy $H : g \circ f \sim \mathsf{id}$ is given by assumption, and the homotopy

$$K' : \textstyle\prod_{(x:A)} G(f(x)) = \mathsf{ap}_f(H(x)) \cdot \mathsf{refl}_{f(x)}$$

is defined as

$$K' :\equiv K \cdot \mathsf{htpy\text{-}right\text{-}unit}(f \cdot H)^{-1}. \qquad \square$$

Our next goal is to show that for any map $f : A \to B$ equipped with

$$g : B \to A, \qquad G : f \circ g \sim \mathsf{id}, \qquad \text{and} \qquad H : g \circ f \sim \mathsf{id},$$

we can improve the homotopy G to a new homotopy $G' : f \circ g \sim \mathsf{id}$ for which there is a further homotopy

$$f \cdot H \sim G' \cdot f.$$

Note that this situation is analogous to the situation in the proof of Theorem 8.1.6, where we improved the contraction C so that it satisfied $C(c) = \mathsf{refl}$. The extra coherence $f \cdot H \sim G' \cdot f$ is then used in the proof that the fibers of an equivalence are contractible.

Definition 8.3.3. Let $f, g : A \to B$ be functions, and consider $H : f \sim g$ and $p : x = y$ in A. We define the identification

$$\mathsf{htpy\text{-}nat}(H, p) :\equiv \mathsf{ap}_f(p) \cdot H(y) = H(x) \cdot \mathsf{ap}_g(p)$$

witnessing that the square

$$\begin{array}{ccc} f(x) & \overset{H(x)}{=\!=} & g(x) \\ {\scriptstyle \mathsf{ap}_f(p)}\big\| & & \big\|{\scriptstyle \mathsf{ap}_g(p)} \\ f(y) & \underset{H(y)}{=\!=} & g(y) \end{array}$$

commutes. This square is also called the **naturality square** of the homotopy H at p.

Construction. By path induction on p it suffices to construct an identification

$$\mathsf{ap}_f(\mathsf{refl}_x) \cdot H(x) = H(x) \cdot \mathsf{ap}_g(\mathsf{refl}_x)$$

since $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$ and $\mathsf{ap}_g(\mathsf{refl}_x) \equiv \mathsf{refl}_{g(x)}$, and since $\mathsf{refl}_{f(x)} \cdot H(x) \equiv H(x)$, we see that the path $\mathsf{right\text{-}unit}(H(x))^{-1}$ is of the asserted type. □

Definition 8.3.4. Consider $f : A \to A$ and $H : f \sim \mathsf{id}_A$. We construct an identification $H(f(x)) = \mathsf{ap}_f(H(x))$, for any $x : A$.

Construction. By the naturality of homotopies with respect to identifications the square

$$\begin{array}{ccc} ff(x) & \xrightarrow{H(f(x))} & f(x) \\ {\scriptstyle \mathsf{ap}_f(H(x))}\downarrow & & \downarrow{\scriptstyle H(x)} \\ f(x) & \xrightarrow[H(x)]{} & x \end{array}$$

commutes. This gives the desired identification $H(f(x)) = \mathsf{ap}_f(H(x))$. □

Lemma 8.3.5. *Let $f : A \to B$ be a map, and consider $(g, G, H) : \mathsf{has\text{-}inverse}(f)$. Then there is a homotopy $G' : f \circ g \sim \mathsf{id}$ equipped with a further homotopy*

$$K : G' \cdot f \sim f \cdot H.$$

Thus we obtain a map $\mathsf{has\text{-}inverse}(f) \to \mathsf{is\text{-}coh\text{-}invertible}(f)$.

Proof. For each $y : B$, we construct the identification $G'(y)$ as the concatenation

$$fg(y) \xrightarrow{G(fg(y))^{-1}} fgfg(y) \xrightarrow{\mathsf{ap}_f(H(g(y)))} fg(y) \xrightarrow{G(y)} y.$$

In order to construct a homotopy $G' \cdot f \sim f \cdot H$, it suffices to show that the square

$$\begin{array}{ccc} fgfgf(x) & \xrightarrow{G(fgf(x))} & fgf(x) \\ {\scriptstyle \mathsf{ap}_f(H(gf(x)))}\downarrow & & \downarrow{\scriptstyle \mathsf{ap}_f(H(x))} \\ fgf(x) & \xrightarrow[G(f(x))]{} & f(x) \end{array}$$

commutes for every $x : A$. Recall from Definition 8.3.4 that we have $H(gf(x)) = \mathsf{ap}_{gf}(H(x))$. Using this identification, we see that it suffices to show that the square

$$\begin{array}{ccc} fgfgf(x) & \xrightarrow{G(fgf(x))} & fgf(x) \\ {\scriptstyle \mathsf{ap}_{fgf}(H(x))}\downarrow & & \downarrow{\scriptstyle \mathsf{ap}_f(H(x))} \\ fgf(x) & \xrightarrow[G(f(x))]{} & f(x) \end{array}$$

commutes. Now we observe that this is just a naturality square the homotopy $Gf : fgf \sim f$, which commutes by Definition 8.3.3. □

Now we put the pieces together to conclude that any equivalence has contractible fibers.

Theorem 8.3.6. *Any equivalence is a contractible map.*

Proof. We have seen in Lemma 8.3.2 that any coherently invertible map is a contractible map. Moreover, any equivalence has the structure of an invertible map by Lemma 7.2.3, and any invertible map is coherently invertible by Lemma 8.3.5. □

Corollary 8.3.7. *Let A be a type, and let $a : A$. Then the type*

$$\textstyle\sum_{(x:A)} x = a$$

is contractible.

Proof. By Remark 7.2.5, the identity function is an equivalence. Therefore, the fibers of the identity function are contractible by Theorem 8.3.6. Note that $\sum_{(x:A)} x = a$ is exactly the fiber of id_A at $a : A$. □

Exercises

8.1 Show that if A is contractible, then for any $x, y : A$ the identity type $x = y$ is also contractible.

8.2 Suppose that A is a retract of B. Show that

$$\mathsf{is\text{-}contr}(B) \to \mathsf{is\text{-}contr}(A).$$

8.3 (a) Show that for any type A, the map $\mathsf{const}_\star : A \to \mathbf{1}$ is an equivalence if and only if A is contractible.

(b) Apply Exercise 7.4 to show that for any map $f : A \to B$, if any two of the three assertions

(i) A is contractible

(ii) B is contractible

(iii) f is an equivalence

hold, then so does the third.

8.4 Show that for any two types A and B, the following are equivalent:

(i) Both A and B are contractible.

(ii) The type $A \times B$ is contractible.

8.5 Let A be a contractible type with center of contraction $a : A$. Furthermore, let B be a type family over A. Show that the map $y \mapsto (a, y) : B(a) \to \sum_{(x:A)} B(x)$ is an equivalence.

8.6 Let B be a family of types over A, and consider the projection map

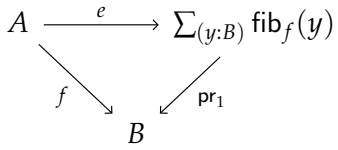
$$\mathsf{pr}_1 : \left(\textstyle\sum_{(x:A)} B(x)\right) \to A.$$

Show that for any $a : A$, the map

$$\lambda((x, y), p).\, \mathsf{tr}_B(p, y) : \mathsf{fib}_{\mathsf{pr}_1}(a) \to B(a),$$

is an equivalence. Conclude that pr_1 is an equivalence if and only if each $B(a)$ is contractible.

8.7 Construct for any map $f : A \to B$ an equivalence $e : A \simeq \sum_{(y:B)} \mathsf{fib}_f(y)$ and a homotopy $H : f \sim \mathsf{pr}_1 \circ e$ witnessing that the triangle



commutes. The projection $\mathsf{pr}_1 : (\sum_{(y:B)} \mathsf{fib}_f(y)) \to B$ is sometimes also called the **fibrant replacement** of f, because first projection maps are fibrations in the homotopy interpretation of type theory.

9 The fundamental theorem of identity types

For many types it is useful to have a characterization of their identity types. For example, we have used a characterization of the identity types of the fibers of a map in order to conclude that any equivalence is a contractible map. The fundamental theorem of identity types is our main tool to carry out such characterizations, and with the fundamental theorem it becomes a routine task to characterize an identity type whenever that is of interest.

Our first application of the fundamental theorem of identity types in the present lecture is a simple proof that any equivalence is an embedding. Embeddings are maps that induce equivalences on identity types, i.e., they are the homotopical analogue of injective maps. In our second application we characterize the identity types of coproducts.

Throughout the rest of this book we will encounter many more occasions to characterize identity types. For example, we will show in Theorem 10.2.6 that the identity type of the natural numbers is equivalent to its observational equality, and we will show in Theorem 15.5.2 that the loop space of the circle is equivalent to $\mathbb{Z}$.

In order to prove the fundamental theorem of identity types, we first prove the basic fact that a family of maps is a family of equivalences if and only if it induces an equivalence on total spaces.

9.1 Families of equivalences

Definition 9.1.1. Consider a family of maps

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

We define the map

$$\mathsf{tot}(f) : \sum_{(x:A)} B(x) \to \sum_{(x:A)} C(x)$$

by $\lambda(x, y).\, (x, f(x, y))$.

Lemma 9.1.2. *For any family of maps $f : \prod_{(x:A)} B(x) \to C(x)$ and any $t : \sum_{(x:A)} C(x)$, there is an equivalence*

$$\mathsf{fib}_{\mathsf{tot}(f)}(t) \simeq \mathsf{fib}_{f(\mathsf{pr}_1(t))}(\mathsf{pr}_2(t)).$$

Proof. For any $p : \mathsf{fib}_{\mathsf{tot}(f)}(t)$ we define $\varphi(t,p) : \mathsf{fib}_{\mathsf{pr}_1(t)}(\mathsf{pr}_2(t))$ by Σ-induction on p. Therefore it suffices to define $\varphi(t,(s,\alpha)) : \mathsf{fib}_{\mathsf{pr}_1(t)}(\mathsf{pr}_2(t))$ for any $s : \sum_{(x:A)} B(x)$ and $\alpha : \mathsf{tot}(f)(s) = t$. Now we proceed by path induction on α, so it suffices to define $\varphi(\mathsf{tot}(f)(s),(s,\mathsf{refl})) : \mathsf{fib}_{f(\mathsf{pr}_1(\mathsf{tot}(f)(s)))}(\mathsf{pr}_2(\mathsf{tot}(f)(s)))$. Finally, we use Σ-induction on s once more, so it suffices to define

$$\varphi((x,f(x,y)),((x,y),\mathsf{refl})) : \mathsf{fib}_{f(x)}(f(x,y)).$$

Now we take as our definition

$$\varphi((x,f(x,y)),((x,y),\mathsf{refl})) :\equiv (y,\mathsf{refl}).$$

For the proof that this map is an equivalence we construct a map

$$\psi(t) : \mathsf{fib}_{f(\mathsf{pr}_1(t))}(\mathsf{pr}_2(t)) \to \mathsf{fib}_{\mathsf{tot}(f)}(t)$$

equipped with homotopies $G(t) : \varphi(t) \circ \psi(t) \sim \mathsf{id}$ and $H(t) : \psi(t) \circ \varphi(t) \sim \mathsf{id}$. In each of these definitions we use Σ-induction and path induction all the way through, until an obvious choice of definition becomes apparent. We define $\psi(t)$, $G(t)$, and $H(t)$ as follows:

$$\begin{aligned}\psi((x,f(x,y)),(y,\mathsf{refl})) &:\equiv ((x,y),\mathsf{refl})\\ G((x,f(x,y)),(y,\mathsf{refl})) &:\equiv \mathsf{refl}\\ H((x,f(x,y)),((x,y),\mathsf{refl})) &:\equiv \mathsf{refl}.\end{aligned}$$

□

Theorem 9.1.3. *Let $f : \prod_{(x:A)} B(x) \to C(x)$ be a family of maps. The following are equivalent:*

*(i) For each $x : A$, the map $f(x)$ is an equivalence. In this case we say that f is a **family of equivalences**.*

(ii) The map $\mathsf{tot}(f) : \sum_{(x:A)} B(x) \to \sum_{(x:A)} C(x)$ is an equivalence.

Proof. By Theorems 8.2.5 and 8.3.6 it suffices to show that $f(x)$ is a contractible map for each $x : A$, if and only if $\mathsf{tot}(f)$ is a contractible map. Thus, we will show that $\mathsf{fib}_{f(x)}(c)$ is contractible if and only if $\mathsf{fib}_{\mathsf{tot}(f)}(x,c)$ is contractible, for each $x : A$ and $c : C(x)$. However, by Lemma 9.1.2 these types are equivalent, so the result follows by Exercise 8.3. □

Now consider the situation where we have a map $f : A \to B$, and a family C over B. Then we have the map

$$\lambda(x,z).\,(f(x),z) : \textstyle\sum_{(x:A)} C(f(x)) \to \sum_{(y:B)} C(y).$$

We claim that this map is an equivalence when f is an equivalence. The technique to prove this claim is the same as the technique we used in Theorem 9.1.3: first we note that the fibers are equivalent to the fibers of f, and then we use the fact that a map is an equivalence if and only if its fibers are contractible to finish the proof.

Lemma 9.1.4. *Consider an equivalence $e : A \simeq B$, and let C be a type family over B. Then the map*

$$\sigma_f(C) :\equiv \lambda(x,z).\,(f(x),z) : \textstyle\sum_{(x:A)} C(f(x)) \to \sum_{(y:B)} C(y)$$

is an equivalence.

Proof. We claim that for each $t : \sum_{(y:B)} C(y)$ there is an equivalence

$$\mathsf{fib}_{\sigma_f(C)}(t) \simeq \mathsf{fib}_f(\mathsf{pr}_1(t)).$$

We prove this by constructing

$$\begin{aligned}
\varphi(t) &: \mathsf{fib}_{\sigma_f(C)}(t) \to \mathsf{fib}_f(\mathsf{pr}_1(t)) \\
\psi(t) &: \mathsf{fib}_f(\mathsf{pr}_1(t)) \to \mathsf{fib}_{\sigma_f(C)}(t) \\
G(t) &: \varphi \circ \psi \sim \mathsf{id} \\
H(t) &: \psi \circ \varphi \sim \mathsf{id}.
\end{aligned}$$

The construction of these functions and homotopies is by using Σ-induction and path induction all the way through, just as in the proof of Lemma 9.1.2. We list the definitions

$$\begin{aligned}
\varphi((f(x),z),((x,z),\mathsf{refl})) &:\equiv (x,\mathsf{refl}) \\
\psi((f(x),z),(x,\mathsf{refl})) &:\equiv ((x,z),\mathsf{refl}) \\
G((f(x),z),(x,\mathsf{refl})) &:\equiv \mathsf{refl} \\
H((f(x),z),((x,z),\mathsf{refl})) &:\equiv \mathsf{refl}.
\end{aligned}$$

Now the claim follows, since we see that φ is a contractible map if and only if f is a contractible map. □

We now combine Theorem 9.1.3 and Lemma 9.1.4.

Definition 9.1.5. Consider a map $f : A \to B$ and a family of maps

$$g : \textstyle\prod_{(x:A)} C(x) \to D(f(x)),$$

where C is a type family over A, and D is a type family over B. In this situation we also say that g is a **family of maps over** f. Then we define

$$\mathsf{tot}_f(g) : \textstyle\sum_{(x:A)} C(x) \to \sum_{(y:B)} D(y)$$

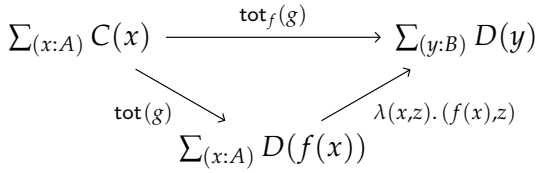
by $\mathsf{tot}_f(g)(x,z) :\equiv (f(x), g(x,z))$.

Theorem 9.1.6. *Suppose that g is a family of maps over f, and suppose that f is an equivalence. Then the following are equivalent:*

(i) The family of maps g over f is a family of equivalences.

(ii) The map $\mathsf{tot}_f(g)$ is an equivalence.

Proof. Note that we have a commuting triangle



By the assumption that f is an equivalence, it follows that the map $\sum_{(x:A)} D(f(x)) \to \sum_{(y:B)} D(y)$ is an equivalence. Therefore it follows that $\mathsf{tot}_f(g)$ is an equivalence if and only if $\mathsf{tot}(g)$ is an equivalence. Now the claim follows, since $\mathsf{tot}(g)$ is an equivalence if and only if g if a family of equivalences. □

9.2 The fundamental theorem

Many types come equipped with a reflexive relation that possesses a similar structure as the identity type. The observational equality on the natural numbers is such an example. We have see that it is a reflexive, symmetric, and transitive relation, and moreover it is contained in any other reflexive relation. Thus, it is natural to ask whether observational equality on the natural numbers is equivalent to the identity type.

The fundamental theorem of identity types (Theorem 9.2.2) is a general theorem that can be used to answer such questions. It describes a necessary and sufficient condition on a type family B over a type A equipped with a point $a : A$, for there to be a family of equivalences $\prod_{(x:A)}(a = x) \simeq B(x)$. In other words, it tells us when a family B is a characterization of the identity type of A.

Before we state the fundamental theorem of identity types we introduce the notion of *identity systems*. Those are families B over a A that satisfy an induction principle that is similar to the path induction principle, where the 'computation rule' is stated with an identification.

Definition 9.2.1. Let A be a type equipped with a term $a : A$. A **(unary) identity system** on A at a consists of a type family B over A equipped with $b : B(a)$, such that for any family of types $P(x, y)$ indexed by $x : A$ and $y : B(x)$, the function

$$h \mapsto h(a, b) : \left(\prod_{(x:A)}\prod_{(y:B(x))} P(x, y)\right) \to P(a, b)$$

has a section.

The most important implication in the fundamental theorem is that (ii) implies (i). Occasionally we will also use the third equivalent statement. We note that the fundamental theorem also appears as Theorem 5.8.4 in [2].

Theorem 9.2.2. *Let A be a type with $a : A$, and let B be be a type family over A with $b : B(a)$. Then the following are logically equivalent for any family of maps*

$$f : \prod_{(x:A)}(a = x) \to B(x).$$

(i) The family of maps f is a family of equivalences.

(ii) *The total space*

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

is contractible.

(iii) *The family B is an identity system.*

In particular the canonical family of maps

$$\mathsf{path\text{-}ind}_a(b) : \prod_{(x:A)} (a = x) \to B(x)$$

is a family of equivalences if and only if $\sum_{(x:A)} B(x)$ is contractible.

Proof. First we show that (i) and (ii) are equivalent. By Theorem 9.1.3 it follows that the family of maps f is a family of equivalences if and only if it induces an equivalence

$$\left(\sum_{(x:A)} a = x\right) \simeq \left(\sum_{(x:A)} B(x)\right)$$

on total spaces. We have that $\sum_{(x:A)} a = x$ is contractible. Now it follows by Exercise 8.3, applied in the case

$$\begin{array}{ccc} \sum_{(x:A)} a = x & \xrightarrow{\mathsf{tot}(f)} & \sum_{(x:A)} B(x) \\ & \searrow^{\simeq} \quad \swarrow & \\ & \mathbf{1} & \end{array}$$

that $\mathsf{tot}(f)$ is an equivalence if and only if $\sum_{(x:A)} B(x)$ is contractible.

Now we show that (ii) and (iii) are equivalent. Note that we have the following commuting triangle

$$\begin{array}{ccc} \prod_{(t:\sum_{(x:A)} B(x))} P(t) & \xrightarrow{\mathsf{ev\text{-}pair}} & \prod_{(x:A)} \prod_{(y:B(x))} P(x,y) \\ & \searrow^{\mathsf{ev\text{-}pt}(a,b)} \quad \swarrow_{\lambda h.\, h(a,b)} & \\ & P(a,b) & \end{array}$$

In this diagram the top map has a section. Therefore it follows by Exercise 7.4 that the left map has a section if and only if the right map has a section. Notice that the left map has a section for all P if and only if $\sum_{(x:A)} B(x)$ satisfies singleton induction, which is by Theorem 8.1.6 equivalent to $\sum_{(x:A)} B(x)$ being contractible. □

9.3 Embeddings

As an application of the fundamental theorem we show that equivalences are embeddings. The notion of embedding is the homotopical analogue of the set theoretic notion of injective map.

Definition 9.3.1. An **embedding** is a map $f : A \to B$ satisfying the property that

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

is an equivalence for every $x, y : A$. We write $\mathsf{is\text{-}emb}(f)$ for the type of witnesses that f is an embedding.

Another way of phrasing the following statement is that equivalent types have equivalent identity types.

Theorem 9.3.2. *Any equivalence is an embedding.*

Proof. Let $e : A \simeq B$ be an equivalence, and let $x : A$. Our goal is to show that

$$\mathsf{ap}_e : (x = y) \to (e(x) = e(y))$$

is an equivalence for every $y : A$. By Theorem 9.2.2 it suffices to show that

$$\textstyle\sum_{(y:A)} e(x) = e(y)$$

is contractible for every $y : A$. Now observe that there is an equivalence

$$\begin{aligned}\textstyle\sum_{(y:A)} e(x) = e(y) &\simeq \textstyle\sum_{(y:A)} e(y) = e(x)\\ &\equiv \mathsf{fib}_e(e(x))\end{aligned}$$

by Theorem 9.1.3, since for each $y : A$ the map

$$\mathsf{inv} : (e(x) = e(y)) \to (e(y) = e(x))$$

is an equivalence by Exercise 7.1. The fiber $\mathsf{fib}_e(e(x))$ is contractible by Theorem 8.3.6, so it follows by Exercise 8.3 that the type $\sum_{(y:A)} e(x) = e(y)$ is indeed contractible. □

9.4 Disjointness of coproducts

To give a second application of the fundamental theorem of identity types, we characterize the identity types of coproducts. Our goal in this section is to prove the following theorem.

Theorem 9.4.1. *Let A and B be types. Then there are equivalences*

$$\begin{aligned}(\mathsf{inl}(x) = \mathsf{inl}(x')) &\simeq (x = x')\\ (\mathsf{inl}(x) = \mathsf{inr}(y')) &\simeq \emptyset\\ (\mathsf{inr}(y) = \mathsf{inl}(x')) &\simeq \emptyset\\ (\mathsf{inr}(y) = \mathsf{inr}(y')) &\simeq (y = y')\end{aligned}$$

for any $x, x' : A$ and $y, y' : B$.

In order to prove Theorem 9.4.1, we first define a binary relation $\mathsf{Eq\text{-}coprod}_{A,B}$ on the coproduct $A + B$.

Definition 9.4.2. Let A and B be types. We define

$$\mathsf{Eq\text{-}coprod}_{A,B} : (A + B) \to (A + B) \to \mathcal{U}$$

by double induction on the coproduct, postulating

$$\begin{aligned}
\mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inl}(x), \mathsf{inl}(x')) &:\equiv (x = x') \\
\mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inl}(x), \mathsf{inr}(y')) &:\equiv \emptyset \\
\mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inr}(y), \mathsf{inl}(x')) &:\equiv \emptyset \\
\mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inr}(y), \mathsf{inr}(y')) &:\equiv (y = y')
\end{aligned}$$

The relation $\mathsf{Eq\text{-}coprod}_{A,B}$ is also called the **observational equality of coproducts**.

Lemma 9.4.3. *The observational equality relation* $\mathsf{Eq\text{-}coprod}_{A,B}$ *on* $A + B$ *is reflexive, and therefore there is a map*

$$\mathsf{Eq\text{-}coprod\text{-}eq} : \textstyle\prod_{(s,t:A+B)} (s = t) \to \mathsf{Eq\text{-}coprod}_{A,B}(s, t)$$

Construction. The reflexivity term ρ is constructed by induction on $t : A + B$, using

$$\begin{aligned}
\rho(\mathsf{inl}(x)) &:\equiv \mathsf{refl}_{\mathsf{inl}(x)} : \mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inl}(x)) \\
\rho(\mathsf{inr}(y)) &:\equiv \mathsf{refl}_{\mathsf{inr}(y)} : \mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inr}(y)).
\end{aligned}$$

□

To show that Eq-coprod-eq is a family of equivalences, we will use the fundamental theorem, Theorem 9.2.2. Moreover, we will use the functoriality of coproducts (established in Exercise 7.12), along with the following facts about Σ-types, coproducts, and the empty type:

$$\begin{aligned}
\textstyle\sum_{(t:A+B)} P(t) &\simeq \left(\textstyle\sum_{(x:A)} P(\mathsf{inl}(x))\right) + \left(\textstyle\sum_{(y:B)} P(\mathsf{inr}(y))\right) \\
\textstyle\sum_{(x:A)} \emptyset &\simeq \emptyset \\
A + \emptyset &\simeq A.
\end{aligned}$$

All of these equivalences are straightforward to construct, so we leave them as an exercise to the reader.

Lemma 9.4.4. *For any* $s : A + B$ *the total space*

$$\textstyle\sum_{(t:A+B)} \mathsf{Eq\text{-}coprod}_{A,B}(s, t)$$

is contractible.

Proof. We will do the proof by induction on s. The two cases are similar, so we only show that the total space

$$\textstyle\sum_{(t:A+B)} \mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inl}(x), t)$$

is contractible. Note that we have equivalences

$$\begin{aligned}
&\textstyle\sum_{(t:A+B)} \mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inl}(x), t) \\
&\simeq \left(\textstyle\sum_{(x':A)} \mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inl}(x), \mathsf{inl}(x'))\right) + \left(\textstyle\sum_{(y':B)} \mathsf{Eq\text{-}coprod}_{A,B}(\mathsf{inl}(x), \mathsf{inr}(y'))\right) \\
&\simeq \left(\textstyle\sum_{(x':A)} x = x'\right) + \left(\textstyle\sum_{(y':B)} \emptyset\right) \\
&\simeq \left(\textstyle\sum_{(x':A)} x = x'\right) + \emptyset \\
&\simeq \textstyle\sum_{(x':A)} x = x'.
\end{aligned}$$

The latter type is contractible by Theorem 8.1.7. □

Proof of Theorem 9.4.1. The proof is now concluded with an application of Theorem 9.2.2, using Lemma 9.4.4. □

Exercises

9.1 (a) Show that the map $\emptyset \to A$ is an embedding for every type A.
(b) Show that $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ are embeddings for any two types A and B.

9.2 Consider an equivalence $e : A \simeq B$. Construct an equivalence

$$(e(x) = y) \simeq (x = e^{-1}(y))$$

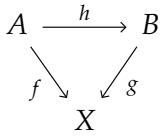
for every $x : A$ and $y : B$.

9.3 Show that

$$(f \sim g) \to (\mathsf{is\text{-}emb}(f) \leftrightarrow \mathsf{is\text{-}emb}(g))$$

for any $f, g : A \to B$.

9.4 Consider a commuting triangle



with $H : f \sim g \circ h$.

(a) Suppose that g is an embedding. Show that f is an embedding if and only if h is an embedding.
(b) Suppose that h is an equivalence. Show that f is an embedding if and only if g is an embedding.

9.5 Consider two maps $f : A \to A'$ and $g : B \to B'$.

(a) Show that if the map

$$f + g : (A + B) \to (A' + B')$$

is an equivalence, then so are both f and g (this is the converse of Exercise 7.12.e).

(b) Show that $f + g$ is an embedding if and only if both f and g are embeddings.

9.6 (a) Let $f, g : \prod_{(x:A)} B(x) \to C(x)$ be two families of maps. Show that

$$\left(\prod_{(x:A)} f(x) \sim g(x)\right) \to \left(\mathsf{tot}(f) \sim \mathsf{tot}(g)\right).$$

(b) Let $f : \prod_{(x:A)} B(x) \to C(x)$ and let $g : \prod_{(x:A)} C(x) \to D(x)$. Show that

$$\mathsf{tot}(\lambda x.\, g(x) \circ f(x)) \sim \mathsf{tot}(g) \circ \mathsf{tot}(f).$$

(c) For any family B over A, show that

$$\mathsf{tot}(\lambda x.\, \mathsf{id}_{B(x)}) \sim \mathsf{id}.$$

9.7 Let $a : A$, and let B be a type family over A.

(a) Use Exercises 8.2 and 9.6 to show that if each $B(x)$ is a retract of $a = x$, then $B(x)$ is equivalent to $a = x$ for every $x : A$.
(b) Conclude that for any family of maps

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

if each $f(x)$ has a section, then f is a family of equivalences.

9.8 Use Exercise 9.7 to show that for any map $f : A \to B$, if

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

has a section for each $x, y : A$, then f is an embedding.

9.9 We say that a map $f : A \to B$ is **path-split** if f has a section, and for each $x, y : A$ the map

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

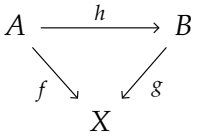
also has a section. We write $\mathsf{path\text{-}split}(f)$ for the type

$$\mathsf{sec}(f) \times \prod_{(x,y:A)} \mathsf{sec}(\mathsf{ap}_f(x, y)).$$

Show that for any map $f : A \to B$ the following are equivalent:

(i) The map f is an equivalence.
(ii) The map f is path-split.

9.10 Consider a triangle



with a homotopy $H : f \sim g \circ h$ witnessing that the triangle commutes.

(a) Construct a family of maps

$$\mathsf{fib\text{-}triangle}(h, H) : \prod_{(x:X)} \mathsf{fib}_f(x) \to \mathsf{fib}_g(x),$$

for which the square

$$\begin{array}{ccc} \sum_{(x:X)} \mathsf{fib}_f(x) & \xrightarrow{\mathsf{tot}(\mathsf{fib\text{-}triangle}(h,H))} & \sum_{(x:X)} \mathsf{fib}_g(x) \\ \downarrow & & \downarrow \\ A & \xrightarrow[h]{} & B \end{array}$$

commutes, where the vertical maps are as constructed in Exercise 8.7.

(b) Show that h is an equivalence if and only if $\mathsf{fib\text{-}triangle}(h, H)$ is a family of equivalences.

10 The hierarchy of homotopical complexity

10.1 Propositions and subtypes

Definition 10.1.1. A type A is said to be a **proposition** if there is a term of type

$$\mathsf{is\text{-}prop}(A) :\equiv \prod_{(x,y:A)} \mathsf{is\text{-}contr}(x = y).$$

Given a universe $\mathcal{U}$, we define $\mathsf{Prop}_{\mathcal{U}}$ to be the type of all small propositions, i.e.,

$$\mathsf{Prop}_{\mathcal{U}} :\equiv \sum_{(X:\mathcal{U})} \mathsf{is\text{-}prop}(A).$$

Example 10.1.2. Any contractible type is a proposition by Exercise 8.1. However, propositions do not need to be inhabited: the empty type is also a proposition, since

$$\prod_{(x,y:\emptyset)} \mathsf{is\text{-}contr}(x = y)$$

follows from the induction principle of the empty type.

In the following lemma we prove that in order to show that a type A is a proposition, it suffices to show that any two terms of A are equal. In other words, propositions are types with **proof irrelevance**.

Theorem 10.1.3. *Let A be a type. Then the following are equivalent:*

(i) The type A is a proposition.

(ii) Any two terms of type A can be identified, i.e., there is a dependent function

$$\mathsf{is\text{-}prop}'(A) :\equiv \prod_{(x,y:A)} x = y.$$

(iii) The type A is contractible as soon as it is inhabited, i.e., there is a function

$$A \to \mathsf{is\text{-}contr}(A).$$

(iv) The map $\mathsf{const}_\star : A \to \mathbf{1}$ *is an embedding.*

Proof. To show that (i) implies (ii), let A be a proposition. Then its identity types are contractible, so the center of contraction of $x = y$ is identification $x = y$, for each $x, y : A$.

To show that (ii) implies (iii), suppose that A comes equipped with $p : \prod_{(x,y:A)} x = y$. Then for any $x : A$ the dependent function $p(x) : \prod_{(y:A)} x = y$ is a contraction of A. Thus we obtain the function

$$\lambda x.\, (x, p(x)) : A \to \mathsf{is\text{-}contr}(A).$$

To show that (iii) implies (iv), suppose that $A \to \mathsf{is\text{-}contr}(A)$ and let $x, y : A$. We have to show that

$$\mathsf{ap}_{\mathsf{const}_\star} : (x = y) \to (\star = \star)$$

is an equivalence. Since we have $x : A$ it follows that A is contractible. Since the unit type is contractible it follows that $\mathsf{const}_\star$ is an equivalence. Therefore we conclude by Theorem 9.3.2 that it is an embedding.

To show that (iv) implies (i), note that if $A \to \mathbf{1}$ is an embedding, then the identity types of A are equivalent to contractible types and therefore they must be contractible. □

In the following lemma we show that propositions are closed under equivalences.

Lemma 10.1.4. *Let A and B be types, and let $e : A \simeq B$. Then we have*

$$\mathsf{is\text{-}prop}(A) \leftrightarrow \mathsf{is\text{-}prop}(B).$$

Proof. We will show that $\mathsf{is\text{-}prop}(B)$ implies $\mathsf{is\text{-}prop}(A)$. This suffices, because the converse follows from the fact that $e^{-1} : B \to A$ is also an equivalence.

Since e is assumed to be an equivalence, it follows by Theorem 9.3.2 that

$$\mathsf{ap}_e : (x = y) \to (e(x) = e(y))$$

is an equivalence for any $x, y : A$. If B is a proposition, then in particular the type $e(x) = e(y)$ is contractible for any $x, y : A$, so the claim follows from Theorem 8.3.6. □

In set theory, a set y is said to be a subset of a set x, if any element of y is an element of x, i.e., if the condition

$$\forall_z (z \in y) \to (z \in x)$$

holds. We have already noted that type theory is different from set theory in that terms in type theory come equipped with a *unique* type. Moreover, in set theory the proposition $x \in y$ is well-formed for any two sets x and y, whereas in type theory the judgment $a : A$ is only well-formed if it is derived using the postulated inference rules. Because of these differences we must find a different way to talk about subtypes.

Note that in set theory there is a correspondence between the subsets of a set x, and the *predicates* on x. A predicate on x is just a proposition $P(z)$ that varies over the elements $z \in x$. Indeed, if y is a subset of x, then the corresponding predicate is the proposition $z \in y$. Conversely, if P is a predicate on x, then we obtain the subset

$$\{z \in x \mid P(z)\}$$

of x. Now we have the right idea of subtypes in type theory: they are families of propositions.

Definition 10.1.5. A type family B over A is said to be a **subtype** of A if for each $x : A$ the type $B(x)$ is a proposition. When B is a subtype of A, we also say that $B(x)$ is a **property** of $x : A$.

We will show in Corollary 10.3.8 that a type family B over A is a subtype of A if and only if the projection map $\mathsf{pr}_1 : \left(\sum_{(x:A)} B(x)\right) \to A$ is an embedding.

10.2 Sets

Definition 10.2.1. A type A is said to be a **set** if it comes equipped with a term of type

$$\mathsf{is\text{-}set}(A) :\equiv \prod_{(x,y:A)} \mathsf{is\text{-}prop}(x = y).$$

Lemma 10.2.2. *A type A is a set if and only if it satisfies **axiom K**, i.e., if and only if it comes equipped with a term of type*

$$\mathsf{axiom\text{-}K}(A) :\equiv \prod_{(x:A)} \prod_{(p:x=x)} \mathsf{refl}_x = p.$$

Proof. If A is a set, then $x = x$ is a proposition, so any two of its elements are equal. This implies axiom K.

For the converse, if A satisfies axiom K, then for any $p, q : x = y$ we have $p \cdot q^{-1} = \mathsf{refl}_x$, and hence $p = q$. This shows that $x = y$ is a proposition, and hence that A is a set. □

Theorem 10.2.3. *Let A be a type, and let $R : A \to A \to \mathcal{U}$ be a binary relation on A satisfying*

(i) Each $R(x, y)$ is a proposition,

(ii) R is reflexive, as witnessed by $\rho : \prod_{(x:A)} R(x, x)$,

(iii) There is a map

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

for each $x, y : A$.

Then any family of maps

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

is a family of equivalences. Consequently, the type A is a set.

Proof. Let $f : \prod_{(x,y:A)} R(x, y) \to (x = y)$. Since R is assumed to be reflexive, we also have a family of maps

$$\mathsf{path\text{-}ind}_x(\rho(x)) : \prod_{(y:A)} (x = y) \to R(x, y).$$

Since each $R(x, y)$ is assumed to be a proposition, it therefore follows that each $R(x, y)$ is a retract of $x = y$. Therefore it follows that $\sum_{(y:A)} R(x, y)$ is a retract of $\sum_{(y:A)} x = y$, which is contractible. We conclude that $\sum_{(y:A)} R(x, y)$ is contractible, and therefore that any family of maps

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

is a family of equivalences.

Now it also follows that A is a set, since its identity types are equivalent to propositions, and therefore they are propositions by Lemma 10.1.4. □

Definition 10.2.4. A map $f : A \to B$ is said to be **injective** if for any $x, y : A$ there is a map

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

Corollary 10.2.5. *Any injective map into a set is an embedding.*

Proof. Let $f : A \to B$ be an injective map between sets. Now consider the relation

$$R(x, y) :\equiv (f(x) = f(y)).$$

Note that R is reflexive, and that $R(x, y)$ is a proposition for each $x, y : A$. Moreover, by the assumption that f is injective, we have

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

for any $x, y : A$. Therefore we are in the situation of Theorem 10.2.3, so it follows that the map $\mathsf{ap}_f : (x = y) \to (f(x) = f(y))$ is an equivalence. □

Theorem 10.2.6. *The type of natural numbers is a set.*

Proof. We will apply Theorem 10.2.3. Note that the observational equality $\mathsf{Eq}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathcal{U})$ on $\mathbb{N}$ (Definition 6.4.2) is a reflexive relation by Exercise 6.1, and moreover that $\mathsf{Eq}_{\mathbb{N}}(n, m)$ is a proposition for every $n, m : \mathbb{N}$ (proof by double induction). Therefore it suffices to show that

$$\textstyle\prod_{(m,n:\mathbb{N})} \mathsf{Eq}_{\mathbb{N}}(m, n) \to (m = n).$$

This follows from the fact that observational equality is the *least* reflexive relation, which was shown in Exercise 6.2. □

10.3 General truncation levels

Definition 10.3.1. We define $\mathsf{is\text{-}trunc} : \mathbb{Z}_{\geq -2} \to \mathcal{U} \to \mathcal{U}$ by induction on $k : \mathbb{Z}_{\geq -2}$, taking

$$\begin{aligned} \mathsf{is\text{-}trunc}_{-2}(A) &:\equiv \mathsf{is\text{-}contr}(A) \\ \mathsf{is\text{-}trunc}_{k+1}(A) &:\equiv \textstyle\prod_{(x,y:A)} \mathsf{is\text{-}trunc}_k(x = y). \end{aligned}$$

For any type A, we say that A is **k-truncated**, or a **k-type**, if there is a term of type $\mathsf{is\text{-}trunc}_k(A)$. We say that a map $f : A \to B$ is k-truncated if its fibers are k-truncated.

Theorem 10.3.2. *If A is a k-type, then A is also a $(k+1)$-type.*

Proof. We have seen in Example 10.1.2 that contractible types are propositions. This proves the base case. For the inductive step, note that if any k-type is also a $(k+1)$-type, then any $(k+1)$-type is a $(k+2)$-type, since its identity types are k-types and therefore $(k+1)$-types. □

Theorem 10.3.3. *If $e : A \simeq B$ is an equivalence, and B is a k-type, then so is A.*

Proof. We have seen in Exercise 8.3 that if B is contractible and $e : A \simeq B$ is an equivalence, then A is also contractible. This proves the base case.

For the inductive step, assume that the k-types are stable under equivalences, and consider $e : A \simeq B$ where B is a $(k+1)$-type. In Theorem 9.3.2 we have seen that

$$\mathsf{ap}_e : (x = y) \to (e(x) = e(y))$$

is an equivalence for any x, y. Note that $e(x) = e(y)$ is a k-type, so by the induction hypothesis it follows that $x = y$ is a k-type. This proves that A is a $(k+1)$-type. □

Corollary 10.3.4. *If $f : A \to B$ is an embedding, and B is a $(k+1)$-type, then so is A.*

Proof. By the assumption that f is an embedding, the action on paths

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

is an equivalence for every $x, y : A$. Since B is assumed to be a $(k+1)$-type, it follows that $f(x) = f(y)$ is a k-type for every $x, y : A$. Therefore we conclude by Theorem 10.3.3 that $x = y$ is a k-type for every $x, y : A$. In other words, A is a $(k+1)$-type. □

Theorem 10.3.5. *Let B be a type family over A. Then the following are equivalent:*

*(i) For each $x : A$ the type $B(x)$ is k-truncated. In this case we say that the family B is **k-truncated**.*

(ii) The projection map

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

is k-truncated.

Proof. By Exercise 8.6 we obtain equivalences

$$\mathsf{fib}_{\mathsf{pr}_1}(x) \simeq B(x)$$

for every $x : A$. Therefore the claim follows from Theorem 10.3.3. □

Theorem 10.3.6. *Let $f : A \to B$ be a map. The following are equivalent:*

(i) The map f is $(k+1)$-truncated.

(ii) For each $x, y : A$, the map

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

is k-truncated.

Proof. First we show that for any $s, t : \mathsf{fib}_f(b)$ there is an equivalence

$$(s = t) \simeq \mathsf{fib}_{\mathsf{ap}_f}(\mathsf{pr}_2(s) \cdot \mathsf{pr}_2(t)^{-1})$$

We do this by Σ-induction on s and t, and then we calculate

$$((x, p) = (y, q)) \simeq \mathsf{Eq\text{-}fib}_f((x, p), (y, q))$$

$$\begin{aligned}
&\equiv \textstyle\sum_{(\alpha:x=y)} p = \mathsf{ap}_f(\alpha) \cdot q \\
&\simeq \textstyle\sum_{(\alpha:x=y)} \mathsf{ap}_f(\alpha) \cdot q = p \\
&\simeq \textstyle\sum_{(\alpha:x=y)} \mathsf{ap}_f(\alpha) = p \cdot q^{-1} \\
&\equiv \mathsf{fib}_{\mathsf{ap}_f}(p \cdot q^{-1}).
\end{aligned}$$

By these equivalences, it follows that if ap_f is k-truncated, then for each $s, t : \mathsf{fib}_f(b)$ the identity type $s = t$ is equivalent to a k-truncated type, and therefore we obtain by Theorem 10.3.3 that f is $(k+1)$-truncated.

For the converse, note that we have equivalences

$$\mathsf{fib}_{\mathsf{ap}_f}(p) \simeq ((x, p) = (y, \mathsf{refl}_{f(y)})).$$

It follows that if f is $(k+1)$-truncated, then the identity type $(x, p) = (y, \mathsf{refl}_{f(y)})$ in $\mathsf{fib}_f(f(y))$ is k-truncated for any $p : f(x) = f(y)$. We conclude by Theorem 10.3.3 that the fiber $\mathsf{fib}_{\mathsf{ap}_f}(p)$ is k-truncated. □

Corollary 10.3.7. *A map is an embedding if and only if its fibers are propositions.*

Corollary 10.3.8. *A type family B over A is a subtype if and only if the projection map*

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

is an embedding.

Theorem 10.3.9. *Let $f : \prod_{(x:A)} B(x) \to C(x)$ be a family of maps. Then the following are equivalent:*

(i) For each $x : A$ the map $f(x)$ is k-truncated.

(ii) The induced map

$$\mathsf{tot}(f) : \left(\textstyle\sum_{(x:A)} B(x)\right) \to \left(\textstyle\sum_{(x:A)} C(x)\right)$$

is k-truncated.

Proof. This follows directly from Lemma 9.1.2 and Theorem 10.3.3. □

Exercises

10.1 (a) Show that $\mathsf{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ is an embedding.
(b) Show that $n \mapsto m + n$ is an embedding, for each $m : \mathbb{N}$. Moreover, conclude that there is an equivalence

$$\mathsf{fib}_{\mathsf{add}_{\mathbb{N}}(m)}(n) \simeq (m \leq n).$$

(c) Show that $n \mapsto mn$ is an embedding, for each $m > 0$ in $\mathbb{N}$. Conclude that the divisibility relation

$$d \mid n$$

is a proposition for each $d, n : \mathbb{N}$ such that $d > 0$.

10.2 Let A be a type, and let the **diagonal** of A be the map $\delta_A : A \to A \times A$ given by $\lambda x.\,(x, x)$.

(a) Show that
$$\mathsf{is\text{-}equiv}(\delta_A) \leftrightarrow \mathsf{is\text{-}prop}(A).$$
(b) Construct an equivalence $\mathsf{fib}_{\delta_A}((x, y)) \simeq (x = y)$ for any $x, y : A$.
(c) Show that A is $(k+1)$-truncated if and only if $\delta_A : A \to A \times A$ is k-truncated.

10.3 (a) Let B be a type family over A. Show that if A is a k-type, and $B(x)$ is a k-type for each $x : A$, then so is $\sum_{(x:A)} B(x)$. Conclude that for any two k-types A and B, the type $A \times B$ is also a k-type. Hint: for the base case, use Exercises 8.3 and 8.5.
(b) Show that for any k-type A, the identity types of A are also k-types.
(c) Show that any maps $f : A \to B$ between k-types A and B is a k-truncated map.
(d) Use Exercise 8.6 to show that for any type family $B : A \to \mathcal{U}$, if A and $\sum_{(x:A)} B(x)$ are k-types, then so is $B(x)$ for each $x : A$.

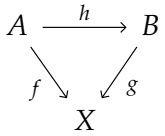
10.4 Show that $\mathbf{2}$ is a set by applying Theorem 10.2.3 with the observational equality on $\mathbf{2}$ defined in Exercise 6.5.

10.5 Show that for any two $(k+2)$-types A and B, the disjoint sum $A + B$ is again a $(k+2)$-type. Conclude that $\mathbb{Z}$ is a set.

10.6 Use Exercises 8.2 and 7.8 to show that if A is a retract of a k-type B, then A is also a k-type.

10.7 Show that a type A is a $(k+1)$-type if and only if the map $\mathsf{const}_x : \mathbf{1} \to A$ is k-truncated for every $x : A$.

10.8 Consider a commuting triangle



with $H : f \sim g \circ h$, and suppose that g is k-truncated. Show that f is k-truncated if and only if h is k-truncated.

Chapter III

Univalent mathematics

11 Function extensionality

11.1 Equivalent forms of function extensionality

Definition 11.1.1. The **axiom of function extensionality** asserts that for any type family B over A, and any two dependent functions $f, g : \prod_{(x:A)} B(x)$, the canonical map

$$\mathsf{htpy\text{-}eq} : (f = g) \to (f \sim g)$$

that sends refl_f to $\mathsf{htpy\text{-}refl}_f$ is an equivalence. We will write $\mathsf{eq\text{-}htpy}$ for its inverse, if it is assumed to exist.

In other words, the axiom of function extensionality asserts that for any two dependent functions $f, g : \prod_{(x:A)} B(x)$, the type of identifications $f = g$ is equivalent to the type of homotopies $f \sim g$ from f to g. By the fundamental theorem of identity types (Theorem 9.2.2) there are three equivalent ways of asserting function extensionality. In the following theorem we state one further equivalent condition.

Theorem 11.1.2. *The following are equivalent:*

(i) The axiom of function extensionality.

(ii) For any type family B over A and any dependent function $f : \prod_{(x:A)} B(x)$, the total space

$$\sum_{(g:\prod_{(x:A)} B(x))} f \sim g$$

is contractible.

*(iii) The principle of **homotopy induction**: for any type family B over A, any dependent function $f : \prod_{(x:A)} B(x)$, and any family of types $P(g, H)$ indexed by $g : \prod_{(x:A)} B(x)$ and $H : f \sim g$, the evaluation function*

$$\left(\prod_{(g:\prod_{(x:A)} B(x))} \prod_{(H:f\sim g)} P(g, H)\right) \to P(f, \mathsf{htpy\text{-}refl}_f)$$

given by $s \mapsto s(f, \mathsf{htpy\text{-}refl}_f)$ has a section.

(iv) *The* ***weak function extensionality principle*** *holds: For every type family B over A one has*

$$\left(\prod_{(x:A)} \mathsf{is\text{-}contr}(B(x))\right) \to \mathsf{is\text{-}contr}\left(\prod_{(x:A)} B(x)\right).$$

Proof. The fact that function extensionality is equivalent to (ii) and (iii) follows directly from Theorem 9.2.2.

To show that function extensionality implies weak function extensionality, suppose that each $B(a)$ is contractible with center of contraction $c(a)$ and contraction $C_a : \prod_{(y:B(a))} c(a) = y$. Then we take $c :\equiv \lambda a.\, c(a)$ to be the center of contraction of $\prod_{(x:A)} B(x)$. To construct the contraction we have to define a term of type

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

Let $f : \prod_{(x:A)} B(x)$. By function extensionality we have a map $(c \sim f) \to (c = f)$, so it suffices to construct a term of type $c \sim f$. Here we take $\lambda a.\, C_a(f(a))$. This completes the proof that function extensionality implies weak function extensionality.

In the remaining part of the proof, we will show that weak function extensionality implies that the type

$$\textstyle\sum_{(g:\prod_{(x:A)} B(x))} f \sim g$$

is contractible for any $f : \prod_{(x:A)} B(x)$. In order to do this, we first note that we have a section-retraction pair

$$\left(\textstyle\sum_{(g:\prod_{(x:A)} B(x))} f \sim g\right) \xrightarrow{i} \left(\prod_{(x:A)} \sum_{(b:B(x))} f(x) = b\right) \xrightarrow{r} \left(\textstyle\sum_{(g:\prod_{(x:A)} B(x))} f \sim g\right).$$

Here we have the functions

$$\begin{aligned} i &:\equiv \lambda(g, H).\, \lambda x.\, (g(x), H(x)) \\ r &:\equiv \lambda p.\, (\lambda x.\, \mathsf{pr}_1(p(x)), \lambda x.\, \mathsf{pr}_2(p(x))). \end{aligned}$$

Their composite is homotopic to the identity function by the computation rule for Σ-types and the η-rule for Π-types:

$$\begin{aligned} r(i(g, H)) &\equiv r(\lambda x.\, (g(x), H(x))) \\ &\equiv (\lambda x.\, g(x), \lambda x.\, H(x)) \\ &\equiv (g, H). \end{aligned}$$

Now we observe that the type $\prod_{(x:A)} \sum_{(b:B(x))} f(x) = b$ is a product of contractible types, so it is contractible by our assumption of the weak function extensionality principle. The claim therefore follows, since retracts of contractible types are contractible by Exercise 8.2. □

For the remainder of this chapter we will assume that the function extensionality axiom holds. In Theorem 12.2.2 we will derive function extensionality from the univalence axiom.

As a first application of the function extensionality axiom we generalize the weak function extensionality axiom to k-types.

Theorem 11.1.3. *Assume function extensionality. Then for any type family B over A one has*

$$\left(\prod_{(x:A)} \mathsf{is\text{-}trunc}_k(B(x))\right) \to \mathsf{is\text{-}trunc}_k\left(\prod_{(x:A)} B(x)\right).$$

Proof. The theorem is proven by induction on $k \geq -2$. The base case is just the weak function extensionality principle, which was shown to follow from function extensionality in Theorem 11.1.2.

For the inductive hypothesis, assume that the k-types are closed under dependent function types. Assume that B is a family of $(k+1)$-types. By function extensionality, the type $f = g$ is equivalent to $f \sim g$ for any two dependent functions $f, g : \prod_{(x:A)} B(x)$. Now observe that $f \sim g$ is a dependent product of k-types, and therefore it is an k-type by our inductive hypotheses. Therefore, it follows by Theorem 10.3.3 that $f = g$ is an k-type, and hence that $\prod_{(x:A)} B(x)$ is an $(k+1)$-type. □

Corollary 11.1.4. *Suppose B is a k-type. Then $A \to B$ is also a k-type, for any type A.*

11.2 The type theoretic principle of choice

The type theoretic principle of choice asserts that Π distributes over Σ. More precisely, it asserts that the canonical map

$$\mathsf{choice} : \left(\prod_{(x:A)} \sum_{(y:B(x))} C(x,y)\right) \to \left(\sum_{(f:\prod_{(x:A)} B(x))} \prod_{(x:A)} C(x, f(x))\right)$$

given by $\lambda h.\, (\mathsf{pr}_1(h(x)), \mathsf{pr}_2(h(x)))$, is an equivalence. In order to see this as a principle of choice, one can view the left hand side as the type of functions h that pick for every $x : A$ a term $y : B(x)$ equipped with a term of type $C(x,y)$. The function choice then constructs a dependent function $f : \prod_{(x:A)} B(x)$ equipped with a term of type $\prod_{(x:A)} C(x, f(x))$. In this section we show that the map choice is an equivalence, and we use this to characterize the identity of any dependent function type $\prod_{(x:A)} B(x)$ in terms of any characterization of the identity types of the individual types $B(x)$.

Theorem 11.2.1. *Consider a family of types $C(x,y)$ indexed by $x : A$ and $y : B(x)$. Then the map*

$$\mathsf{choice} : \left(\prod_{(x:A)} \sum_{(y:B(x))} C(x,y)\right) \to \left(\sum_{(f:\prod_{(x:A)} B(x))} \prod_{(x:A)} C(x, f(x))\right)$$

given by $\lambda h.\, (\mathsf{pr}_1(h(x)), \mathsf{pr}_2(h(x)))$ is an equivalence.

Proof. We define the map

$$\mathsf{choice}^{-1} : \left(\sum_{(f:\prod_{(x:A)} B(x))} \prod_{(x:A)} C(x, f(x))\right) \to \left(\prod_{(x:A)} \sum_{(y:B(x))} C(x,y)\right)$$

by $\lambda(f,g).\, \lambda x.\, (f(x), g(x))$. Then we have to construct homotopies

$$\mathsf{choice} \circ \mathsf{choice}^{-1} \sim \mathsf{id}, \qquad \text{and} \qquad \mathsf{choice}^{-1} \circ \mathsf{choice} \sim \mathsf{id}.$$

For the first homotopy it suffices to construct an identification

$$\mathsf{choice}(\mathsf{choice}^{-1}(f,g)) = (f,g)$$

for any $f : \prod_{(x:A)} B(x)$ and any $g : \prod_{(x:A)} C(x, f(x))$. We compute the left-hand side as follows:

$$\begin{aligned}\mathsf{choice}(\mathsf{choice}^{-1}(f,g)) &\equiv \mathsf{choice}(\lambda x.\,(f(x), g(x))) \\ &\equiv (\lambda x.\, f(x), \lambda x.\, g(x)).\end{aligned}$$

By the η-rule it follows that $f \equiv \lambda x.\, f(x)$ and $g \equiv \lambda x.\, g(x)$. Therefore we have the identification

$$\mathsf{refl}_{(f,g)} : \mathsf{choice}(\mathsf{choice}^{-1}(f,g)) = (f,g).$$

This completes the construction of the first homotopy.

For the second homotopy we have to construct an identification

$$\mathsf{choice}^{-1}(\mathsf{choice}(h)) = h$$

for any $h : \prod_{(x:A)} \sum_{(y:B(x))} C(x,y)$. We compute the left-hand side as follows:

$$\begin{aligned}\mathsf{choice}^{-1}(\mathsf{choice}(h)) &\equiv \mathsf{choice}^{-1}(\lambda x.\, \mathsf{pr}_1(h(x)), (\lambda x.\, \mathsf{pr}_2(h(x)))) \\ &\equiv \lambda x.\, (\mathsf{pr}_1(h(x)), \mathsf{pr}_2(h(x)))\end{aligned}$$

However, it is *not* the case that $(\mathsf{pr}_1(h(x)), \mathsf{pr}_2(h(x))) \equiv h(x)$ for any $h : \prod_{(x:A)} \sum_{(y:B(x))} C(x,y)$. Nevertheless, we have the identification

$$\mathsf{eq\text{-}pair}(\mathsf{refl}, \mathsf{refl}) : (\mathsf{pr}_1(h(x)), \mathsf{pr}_2(h(x))) = h(x).$$

Therefore we obtain the required homotopy by function extensionality:

$$\lambda h.\, \mathsf{eq\text{-}htpy}(\lambda x.\, \mathsf{eq\text{-}pair}(\mathsf{refl}_{\mathsf{pr}_1(h(x))}, \mathsf{refl}_{\mathsf{pr}_2(h(x))})) : \mathsf{choice}^{-1} \circ \mathsf{choice} \sim \mathsf{id}. \qquad \square$$

Corollary 11.2.2. *For type A and any type family C over B, the map*

$$\left(\sum_{(f:A\to B)} \prod_{(x:A)} C(f(x))\right) \to \left(A \to \sum_{(y:B)} C(x)\right)$$

given by $\lambda(f,g).\, \lambda x.\, (f(x), g(x))$ is an equivalence.

Remark 11.2.3. The type theoretic choice principle can be used to derive the binomial theorem. We give an informal argument of how this goes. Recall that the binomial theorem asserts that

$$(n+m)^k = \sum_{l=0}^{k} \binom{k}{l} n^l m^{k-l}$$

for any three natural numbers k, m, n.

Consider the types $A :\equiv \mathsf{Fin}(k)$, $B :\equiv \mathsf{Fin}(n)$ and $C :\equiv \mathsf{Fin}(m)$. Then we can define the type family $P : \mathbf{2} \to \mathcal{U}$ given by

$$\begin{aligned}P(1_{\mathbf{2}}) &:\equiv B \\ P(0_{\mathbf{2}}) &:\equiv C.\end{aligned}$$

Now, the type theoretic principle of choice gives us an equivalence

$$\left(\prod_{(x:A)}\sum_{(t:\mathbf{2})}P(t)\right) \simeq \left(\sum_{(f:A\to\mathbf{2})}\prod_{(x:A)}P(f(x))\right).$$

Now we note that the type $(f(x) = 1) + (f(x) = 0)$ is contractible for any $f : A \to \mathbf{2}$ and $x : A$. Therefore we have equivalences

$$\begin{aligned}\sum_{(f:A\to\mathbf{2})}\prod_{(x:A)}P(f(x) &\simeq \sum_{(f:A\to\mathbf{2})}\prod_{(x:A)}\prod_{(t:(f(x)=1)+(f(x)=0))}P(f(x))\\ &\simeq \sum_{(f:A\to\mathbf{2})}(\mathsf{fib}_f(1)\to B)\times(\mathsf{fib}_f(0)\to C)\end{aligned}$$

Now we note that, because there are $\binom{k}{l}$ ways to choose a subset of l elements of A, there are

$$\sum_{l=0}^{k}\binom{k}{l}n^l m^{k-l}$$

elements in the above type.

11.3 Universal properties

The function extensionality principle allows us to prove *universal properties*. Universal properties are characterizations of all maps out of or into a given type, so they are very important. Among other applications, universal properties characterize a type up to equivalence. In the following theorem we prove the universal property of dependent pair types.

Theorem 11.3.1. *Let B be a type family over A, and let X be a type. Then the map*

$$\mathsf{ev\text{-}pair} : \left(\left(\sum_{(x:A)}B(x)\right)\to X\right)\to\left(\prod_{(x:A)}(B(x)\to X)\right)$$

given by $f \mapsto \lambda a.\,\lambda b.\,f(a,b)$ is an equivalence.

Proof. The map in the converse direction is simply

$$\mathsf{ind}_\Sigma : \left(\prod_{(x:A)}(B(x)\to X)\right)\to\left(\left(\sum_{(x:A)}B(x)\right)\to X\right).$$

By the computation rules for Σ-types we have

$$\lambda f.\,\mathsf{refl}_f : \mathsf{ev\text{-}pair}\circ\mathsf{ind}_\Sigma \sim \mathsf{id}$$

To show that $\mathsf{ind}_\Sigma \circ \mathsf{ev\text{-}pair} \sim \mathsf{id}$ we will also apply function extensionality. Thus, it suffices to show that $\mathsf{ind}_\Sigma(\lambda x.\,\lambda y.\,f((x,y))) = f$. We apply function extensionality again, so it suffices to show that

$$\prod_{(t:\sum_{(x:A)}B(x))}\mathsf{ind}_\Sigma\big(\lambda x.\,\lambda y.\,f((x,y))\big)(t) = f(t).$$

We obtain this homotopy by another application of Σ-induction. □

Corollary 11.3.2. *Let A, B, and X be types. Then the map*

$$\mathsf{ev\text{-}pair} : (A \times B \to X) \to (A \to (B \to X))$$

given by $f \mapsto \lambda a.\, \lambda b.\, f((a,b))$ is an equivalence.

The universal property of identity types is sometimes called the *type theoretical Yoneda lemma*: families of maps out of the identity type are uniquely determined by their action on the reflexivity identification.

Theorem 11.3.3. *Let B be a type family over A, and let $a : A$. Then the map*

$$\mathsf{ev\text{-}refl} : \left(\prod_{(x:A)} (a = x) \to B(x)\right) \to B(a)$$

given by $\lambda f.\, f(a, \mathsf{refl}_a)$ is an equivalence.

Proof. The inverse φ is defined by path induction, taking $b : B(a)$ to the function f satisfying $f(a, \mathsf{refl}_a) \equiv b$. It is immediate that $\mathsf{ev\text{-}refl} \circ \varphi \sim \mathsf{id}$.

To see that $\varphi \circ \mathsf{ev\text{-}refl} \sim \mathsf{id}$, let $f : \prod_{(x:A)} (a = x) \to B(x)$. To show that $\varphi(f(a, \mathsf{refl}_a)) = f$ we use function extensionality (twice), so it suffices to show that

$$\prod_{(x:A)} \prod_{(p:a=x)} \varphi(f(a, \mathsf{refl}_a), x, p) = f(x, p).$$

This follows by path induction on p, since $\varphi(f(a, \mathsf{refl}_a), a, \mathsf{refl}_a) \equiv f(a, \mathsf{refl}_a)$. □

11.4 Composing with equivalences

We show in this section that a map $f : A \to B$ is an equivalence if and only if for any type X the precomposition map

$$- \circ f : (B \to X) \to (A \to X)$$

is an equivalence. Moreover, we will show in Theorem 11.4.1 that the 'dependent version' of this statement also holds: a map $f : A \to B$ is an equivalence if and only if for any type family P over B, the precomposition map

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

is an equivalence.

Theorem 11.4.1. *For any map $f : A \to B$, the following are equivalent:*

(i) f is an equivalence.

(ii) For any type family P over B the map

$$\left(\prod_{(y:B)} P(y)\right) \to \left(\prod_{(x:A)} P(f(x))\right)$$

given by $h \mapsto h \circ f$ is an equivalence.

(iii) For any type X the map

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

given by $g \mapsto g \circ f$ is an equivalence.

Proof. To show that (i) implies (ii), we first recall from Lemma 8.3.5 that any equivalence is also coherently invertible. Therefore f comes equipped with

$$\begin{aligned} g &: B \to A \\ G &: f \circ g \sim \mathsf{id}_B \\ H &: g \circ f \sim \mathsf{id}_A \\ K &: G \cdot f \sim f \cdot H. \end{aligned}$$

Then we define the inverse of $- \circ f$ to be the map

$$\varphi : \left(\prod_{(x:A)} P(f(x))\right) \to \left(\prod_{(y:B)} P(y)\right)$$

given by $h \mapsto \lambda y.\, \mathsf{tr}_P(G(y), h(g(y)))$.

To see that φ is a section of $- \circ f$, let $h : \prod_{(x:A)} P(f(x))$. By function extensionality it suffices to construct a homotopy $\varphi(h) \circ f \sim h$. In other words, we have to show that

$$\mathsf{tr}_P(G(f(x)), h(g(f(x)))) = h(x)$$

for any $x : A$. Now we use the additional homotopy K from our assumption that f is coherently invertible. Since we have $K(x) : G(f(x)) = \mathsf{ap}_f(H(x))$ it suffices to show that

$$\mathsf{tr}_P(\mathsf{ap}_f(H(x)), hgf(x)) = h(x).$$

A simple path-induction argument yields that

$$\mathsf{tr}_P(\mathsf{ap}_f(p)) \sim \mathsf{tr}_{P \circ f}(p)$$

for any path $p : x = y$ in A, so it suffices to construct an identification

$$\mathsf{tr}_{P \circ f}(H(x), hgf(x)) = h(x).$$

We have such an identification by $\mathsf{apd}_h(H(x))$.

To see that φ is a retraction of $- \circ f$, let $h : \prod_{(y:B)} P(y)$. By function extensionality it suffices to construct a homotopy $\varphi(h \circ f) \sim h$. In other words, we have to show that

$$\mathsf{tr}_P(G(y), hfg(y)) = h(y)$$

for any $y : B$. We have such an identification by $\mathsf{apd}_h(G(y))$. This completes the proof that (i) implies (ii).

Note that (iii) is an immediate consequence of (ii), since we can just choose P to be the constant family X.

It remains to show that (iii) implies (i). Suppose that

$$- \circ f : (B \to X) \to (A \to X)$$

is an equivalence for every type X. Then its fibers are contractible by Theorem 8.3.6. In particular, choosing $X \equiv A$ we see that the fiber

$$\mathsf{fib}_{-\circ f}(\mathsf{id}_A) \equiv \textstyle\sum_{(h:B\to A)} h \circ f = \mathsf{id}_A$$

is contractible. Thus we obtain a function $h : B \to A$ and a homotopy $H : h \circ f \sim \mathsf{id}_A$ showing that h is a retraction of f. We will show that h is also a section of f. To see this, we use that the fiber

$$\mathsf{fib}_{-\circ f}(f) \equiv \textstyle\sum_{(i:B\to B)} i \circ f = f$$

is contractible (choosing $X \equiv B$). Of course we have $(\mathsf{id}_B, \mathsf{refl}_f)$ in this fiber. However we claim that there also is an identification $p : (f \circ h) \circ f = f$, showing that $(f \circ h, p)$ is in this fiber, because

$$\begin{aligned}(f \circ h) \circ f &\equiv f \circ (h \circ f)\\ &= f \circ \mathsf{id}_A\\ &\equiv f\end{aligned}$$

Now we conclude by the contractibility of the fiber that $(\mathsf{id}_B, \mathsf{refl}_f) = (f \circ h, p)$. In particular we obtain that $\mathsf{id}_B = f \circ h$, showing that h is a section of f. □

Exercises

11.1 Show that the functions

$$\begin{aligned}\mathsf{htpy\text{-}inv} &: (f \sim g) \to (g \sim f)\\ \mathsf{htpy\text{-}concat}(H) &: (g \sim h) \to (f \sim h)\\ \mathsf{htpy\text{-}concat}'(K) &: (f \sim g) \to (f \sim h)\end{aligned}$$

are equivalences for every $f, g, h : \prod_{(x:A)} B(x)$. Here, $\mathsf{htpy\text{-}concat}'(K)$ is the function defined by $H \mapsto H \cdot K$.

11.2 (a) Show that for any type A the type $\mathsf{is\text{-}contr}(A)$ is a proposition.
(b) Show that for any type A and any $k \geq -2$, the type $\mathsf{is\text{-}trunc}_k(A)$ is a proposition.

11.3 Let $f : X \to Y$ be a map. Show that the following are equivalent:

(i) f is an equivalence.
(ii) The map $f \circ - : X^A \to Y^A$ is an equivalence for every type A.

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

(a) Show that if f is an equivalence, then the type $\sum_{(g:B\to A)} f \circ g \sim \mathsf{id}$ of sections of f is contractible.
(b) Show that if f is an equivalence, then the type $\sum_{(h:B\to A)} h \circ f \sim \mathsf{id}$ of retractions of f is contractible.
(c) Show that $\mathsf{is\text{-}equiv}(f)$ is a proposition.
(d) Use Exercises 11.2 and 11.5 to show that $\mathsf{is\text{-}equiv}(f) \simeq \mathsf{is\text{-}contr}(f)$.

Conclude that $A \simeq B$ is a subtype of $A \to B$, and in particular that the map $\mathsf{pr}_1 : (A \simeq B) \to (A \to B)$ is an embedding.

11.5 (a) Let P and Q be propositions. Show that

$$(P \leftrightarrow Q) \simeq (P \simeq Q).$$

(b) Show that P is a proposition if and only if $P \to P$ is contractible.

11.6 Show that $\mathsf{path\text{-}split}(f)$ and $\mathsf{is\text{-}coh\text{-}invertible}(f)$ are propositions for any map $f : A \to B$. Conclude that we have equivalences

$$\mathsf{is\text{-}equiv}(f) \simeq \mathsf{path\text{-}split}(f) \simeq \mathsf{is\text{-}coh\text{-}invertible}(f).$$

11.7 Construct for any type A an equivalence

$$\mathsf{has\text{-}inverse}(\mathsf{id}_A) \simeq \Big(\mathsf{id}_A \sim \mathsf{id}_A\Big).$$

Note: We will use this fact in Exercise 15.6 to show that there are types for which $\mathsf{is\text{-}invertible}(\mathsf{id}_A) \not\simeq \mathsf{is\text{-}equiv}(\mathsf{id}_A)$.

11.8 (a) Show that the type

$$\textstyle\prod_{(t:\emptyset)} P(t)$$

is contractible for any $P : \emptyset \to \mathcal{U}$.

(b) Show that for any type X the following are equivalent:

(i) the unique map $\emptyset \to X$ is an equivalence.

(ii) The type Y^X is contractible for any type Y.

11.9 Consider two types A and B.

(a) Show that the map

$$\mathsf{ev\text{-}inl\text{-}inr} : \Big(\textstyle\prod_{(t:A+B)} P(t)\Big) \to \Big(\textstyle\prod_{(x:A)} P(\mathsf{inl}(x))\Big) \times \Big(\textstyle\prod_{(y:B)} P(\mathsf{inr}(y))\Big)$$

given by $f \mapsto (f \circ \mathsf{inl}, f \circ \mathsf{inr})$ is an equivalence.

(b) Show that the following are equivalent for any type X equipped with maps $i : A \to X$ and $j : B \to X$:

(i) The map $\mathsf{ind}_+(i, j) : A + B \to X$ is an equivalence.

(ii) For any type Y, the map

$$\lambda f.\, (f \circ i, f \circ j) : (X \to Y) \to (A \to Y) \times (B \to Y)$$

is an equivalence.

11.10 (a) Show that the map

$$\Big(\textstyle\prod_{(t:\mathbf{1})} P(t)\Big) \to P(\star)$$

given by $\lambda f.\, f(\star)$ is an equivalence.

(b) Consider a type X equipped with a point $x : X$. Show that the following are equivalent:

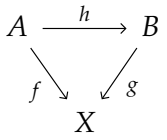
(i) The map $\mathsf{ind}_{\mathbf{1}}(x) : \mathbf{1} \to X$ is an equivalence (i.e., X is contractible).

(ii) For any type Y the map

$$\lambda f.\, f(x) : (X \to Y) \to Y$$

is an equivalence.

11.11 Consider a commuting triangle



with $H : f \sim g \circ h$.

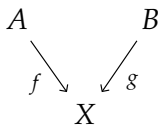
(a) Show that if h has a section, then $\mathsf{sec}(g)$ is a retract of $\mathsf{sec}(f)$.
(b) Show that if g has a retraction, then $\mathsf{retr}(h)$ is a retract of $\mathsf{sec}(f)$.

11.12 Let $e_i : A_i \simeq B_i$ be an equivalence for every $i : I$. Show that the map

$$\lambda f.\, \lambda i.\, e_i \circ f : \left(\prod_{(i:I)} A_i\right) \to \left(\prod_{(i:I)} B_i\right)$$

is an equivalence.

11.13 Consider a diagram of the form



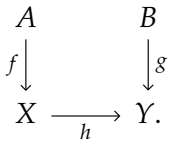
(a) Show that the type $\sum_{(h:A\to B)} f \sim g \circ h$ is equivalent to the type of families of maps

$$\prod_{(x:X)} \mathsf{fib}_f(x) \to \mathsf{fib}_g(x).$$

(b) Show that the type $\sum_{(h:A\simeq B)} f \sim g \circ h$ is equivalent to the type of families of equivalences

$$\prod_{(x:X)} \mathsf{fib}_f(x) \simeq \mathsf{fib}_g(x).$$

11.14 Consider a diagram of the form



Show that the type $\sum_{(i:A\to B)} h \circ f \sim g \circ i$ is equivalent to the type of families of maps

$$\prod_{(x:X)} \mathsf{fib}_f(x) \to \mathsf{fib}_g(h(x)).$$

11.15 Let A and B be sets. Show that type type $A \simeq B$ of equivalences from A to B is equivalent to the type $A \cong B$ of **isomorphisms** from A to B, i.e., the type of quadruples (f, g, H, K) consisting of

$$f : A \to B$$
$$g : B \to A$$
$$H : f \circ g = \mathsf{id}_B$$
$$K : g \circ f = \mathsf{id}_A.$$

11.16 Let B be a type family over A, and consider the maps

$$\mathsf{pr}_1 : \textstyle\sum_{(x:A)} B(x) \to A$$
$$\mathsf{pr}_1 \circ - : \left(\textstyle\sum_{(x:A)} B(x)\right)^A \to A^A.$$

Construct equivalences

$$\left(\textstyle\prod_{(x:A)} B(x)\right) \simeq \mathsf{sec}(\mathsf{pr}_1) \simeq \mathsf{fib}_{\mathsf{pr}_1 \circ -}(\mathsf{id}_A).$$

11.17 Suppose that $A : I \to \mathcal{U}$ is a type family over a set I with decidable equality. Show that

$$\left(\textstyle\prod_{(i:I)} \mathsf{is\text{-}contr}(A_i)\right) \leftrightarrow \mathsf{is\text{-}contr}\left(\textstyle\prod_{(i:I)} A_i\right).$$

11.18 Construct equivalences

$$\mathsf{Fin}(n^m) \simeq (\mathsf{Fin}(m) \to \mathsf{Fin}(n))$$
$$\mathsf{Fin}(n!) \simeq (\mathsf{Fin}(n) \simeq \mathsf{Fin}(n)).$$

12 The univalence axiom

12.1 Equivalent forms of the univalence axiom

The univalence axiom characterizes the identity type of the universe. Roughly speaking, it asserts that equivalent types are equal. It is considered to be an *extensionality principle* for types.

Definition 12.1.1. The **univalence axiom** on a universe $\mathcal{U}$ is the statement that for any $A : \mathcal{U}$ the family of maps

$$\mathsf{equiv\text{-}eq} : \textstyle\prod_{(B:\mathcal{U})} (A = B) \to (A \simeq B).$$

that sends refl_A to the identity equivalence $\mathsf{id} : A \simeq A$ is a family of equivalences. A universe satisfying the univalence axiom is referred to as a **univalent universe**. If $\mathcal{U}$ is a univalent universe we will write $\mathsf{eq\text{-}equiv}$ for the inverse of $\mathsf{equiv\text{-}eq}$.

The following theorem is a special case of the fundamental theorem of identity types (Theorem 9.2.2). Subsequently we will assume that any type is contained in a univalent universe.

Theorem 12.1.2. *The following are equivalent:*

(i) The univalence axiom holds.

(ii) The type

$$\textstyle\sum_{(B:\mathcal{U})} A \simeq B$$

is contractible for each $A : \mathcal{U}$.

(iii) *The principle of* ***equivalence induction*** *holds: for every* $A : \mathcal{U}$ *and for every type family*

$$P : \prod_{(B:\mathcal{U})}(A \simeq B) \to \mathcal{U},$$

the map

$$\left(\prod_{(B:\mathcal{U})}\prod_{(e:A\simeq B)} P(B,e)\right) \to P(A,\mathsf{id}_A)$$

given by $f \mapsto f(A,\mathsf{id}_A)$ *has a section.*

12.2 Univalence implies function extensionality

One of the first applications of the univalence axiom was Voevodsky's theorem that the univalence axiom on a universe $\mathcal{U}$ implies function extensionality for types in $\mathcal{U}$. The proof uses the fact that weak function extensionality implies function extensionality.

We will also make use of the following lemma. Note that this statement was also part of Exercise 11.3. That exercise is solved using function extensionality. Since our present goal is to derive function extensionality from the univalence axiom, we cannot make use of that exercise.

Lemma 12.2.1. *For any equivalence* $e : X \simeq Y$ *in a univalent universe* $\mathcal{U}$*, and any type* A*, the post-composition map*

$$e \circ - : (A \to X) \to (A \to Y)$$

is an equivalence.

Proof. The statement is obvious for the identity equivalence $\mathsf{id} : X \simeq X$. Therefore the claim follows by equivalence induction, which is by Theorem 12.1.2 one of the equivalent forms of the univalence axiom. □

Theorem 12.2.2. *For any universe* $\mathcal{U}$*, the univalence axiom on* $\mathcal{U}$ *implies function extensionality on* $\mathcal{U}$*.*

Proof. Note that by Theorem 11.1.2 it suffices to show that univalence implies weak function extensionality, where we note that Theorem 11.1.2 also holds when it is restricted to small types.

Suppose that $B : A \to \mathcal{U}$ is a family of contractible types. Our goal is to show that the product $\prod_{(x:A)} B(x)$ is contractible. Since each $B(x)$ is contractible, the projection map $\mathsf{pr}_1 : \left(\sum_{(x:A)} B(x)\right) \to A$ is an equivalence by Exercise 8.6.

Now it follows by Lemma 12.2.1 that $\mathsf{pr}_1 \circ -$ is an equivalence. Consequently, it follows from Theorem 8.3.6 that the fibers of

$$\mathsf{pr}_1 \circ - : \left(A \to \sum_{(x:A)} B(x)\right) \to (A \to A)$$

are contractible. In particular, the fiber at id_A is contractible. Therefore it suffices to show that $\prod_{(x:A)} B(x)$ is a retract of $\sum_{(f:A\to\sum_{(x:A)} B(x))} \mathsf{pr}_1 \circ f = \mathsf{id}_A$. In other words, we will construct

$$\left(\prod_{(x:A)} B(x)\right) \xrightarrow{\ i\ } \left(\sum_{(f:A\to\sum_{(x:A)} B(x))} \mathsf{pr}_1 \circ f = \mathsf{id}_A\right) \xrightarrow{\ r\ } \left(\prod_{(x:A)} B(x)\right),$$

and a homotopy $r \circ i \sim \mathsf{id}$.

We define the function i by

$$i(f) :\equiv (\lambda x.\, (x, f(x)), \mathsf{refl}_{\mathsf{id}}).$$

To see that this definition is correct, we need to know that

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

This is indeed the case, by the η-rule for Π-types.

Next, we define the function r. Let $h : A \to \sum_{(x:A)} B(x)$, and let $p : \mathsf{pr}_1 \circ h = \mathsf{id}$. Then we have the homotopy $H :\equiv \mathsf{htpy\text{-}eq}(p) : \mathsf{pr}_1 \circ h \sim \mathsf{id}$. Then we have $\mathsf{pr}_2(h(x)) : B(\mathsf{pr}_1(h(x)))$ and we have the identification $H(x) : \mathsf{pr}_1(h(x)) = x$. Therefore we define r by

$$r((h, p), x) :\equiv \mathsf{tr}_B(H(x), \mathsf{pr}_2(h(x))).$$

We note that if $p \equiv \mathsf{refl}_{\mathsf{id}}$, then $H(x) \equiv \mathsf{refl}_x$. In this case we have the judgmental equality $r((h, \mathsf{refl}), x) \equiv \mathsf{pr}_2(h(x))$. Thus we see that $r \circ i \equiv \mathsf{id}$ by another application of the η-rule for Π-types. □

12.3 Propositional extensionality and posets

Theorem 12.3.1. *Propositions satisfy **propositional extensionality**: for any two propositions P and Q, the canonical map*

$$\mathsf{iff\text{-}eq} : (P = Q) \to (P \leftrightarrow Q)$$

that sends refl_P to $(\mathsf{id}, \mathsf{id})$ is an equivalence. It follows that the type Prop of propositions in $\mathcal{U}$ is a set.

Note that for any $P : \mathsf{Prop}$, we usually also write P for the underlying type of the proposition P. If we would be more formal about it we would have to write $\mathsf{pr}_1(P)$ for the underlying type, since Prop is the Σ-type $\sum_{(X:\mathcal{U})} \mathsf{is\text{-}prop}(X)$. In the following proof it is clearer if we use the more formal notation $\mathsf{pr}_1(P)$ for the underlying type of a proposition P.

Proof. We note that the identity type $P = Q$ is an identity type in Prop. However, since $\mathsf{is\text{-}prop}(X)$ is a proposition for any type X, it follows that the map

$$\mathsf{ap}_{\mathsf{pr}_1} : (P = Q) \to (\mathsf{pr}_1(P) = \mathsf{pr}_1(Q))$$

is an equivalence. Now we observe that we have a commuting square

$$\begin{array}{ccc} (P = Q) & \longrightarrow & (P \leftrightarrow Q) \\ {\scriptstyle \mathsf{ap}_{\mathsf{pr}_1}}\downarrow & & \uparrow{\scriptstyle \simeq} \\ (\mathsf{pr}_1(P) = \mathsf{pr}_1(Q)) & \xrightarrow[\mathsf{equiv\text{-}eq}]{} & (\mathsf{pr}_1(P) \simeq \mathsf{pr}_1(Q)) \end{array}$$

Since the left, bottom, and right map are equivalences, it follows that the top map is an equivalence. □

Definition 12.3.2. A **partially ordered set (poset)** is a set P equipped with a relation

$$- \leq - : P \to (P \to \mathsf{Prop})$$

that is **reflexive** (for every $x : P$ we have $x \leq x$), **transitive** (for every $x, y, z : P$ such that $x \leq y$ and $y \leq z$ we have $x \leq z$), and **anti-symmetric** (for every $x, y : P$ such that $x \leq y$ and $y \leq x$ we have $x = y$).

Remark 12.3.3. The condition that X is a set can be omitted from the definition of a poset. Indeed, if X is any type that comes equipped with a Prop-valued ordering relation $\leq$ that is reflexive and anti-symmetric, then X is a set by Theorem 10.2.3.

Example 12.3.4. The type Prop is a poset, where the ordering relation is given by implication: P is less than Q if $P \to Q$. The fact that $P \to Q$ is a proposition is a special case of Corollary 11.1.4. The relation $P \to Q$ is reflexive by the identity function, and transitive by function composition. Moreover, the relation $P \to Q$ is anti-symmetric by Theorem 12.3.1.

Example 12.3.5. The type of natural numbers comes equipped with at least two important poset structures. The first is given by the usual ordering relation $\leq$, and the second is given by the relation $d \mid n$ that d divides n.

Theorem 12.3.6. *For any poset P and any type X, the set P^X is a poset. In particular the type of subtypes of any type is a poset.*

Proof. Let P be a poset with ordering $\leq$, and let X be a type. Then P^X is a set by Corollary 11.1.4. For any $f, g : X \to P$ we define

$$(f \leq g) :\equiv \textstyle\prod_{(x:X)} f(x) \leq g(x).$$

Reflexivity and transitivity follow immediately from reflexivity and transitivity of the original relation. Moreover, by the anti-symmetry of the original relation it follows that

$$(f \leq g) \times (g \leq f) \to (f \sim g).$$

Therefore we obtain an identification $f = g$ by function extensionality. The last claim follows immediately from the fact that a subtype of X is a map $X \to \mathsf{Prop}$, and the fact that Prop is a poset. □

Exercises

12.1 (a) Use the univalence axiom to show that the type $\sum_{(A:\mathcal{U})} \mathsf{is\text{-}contr}(A)$ of all contractible types in $\mathcal{U}$ is contractible.
(b) Use Corollaries 10.3.4 and 11.1.4 and Exercise 11.4 to show that if A and B are $(k+1)$-types, then the type $A \simeq B$ is also a $(k+1)$-type.
(c) Use univalence to show that the universe of k-types

$$\mathcal{U}^{\leq k} :\equiv \textstyle\sum_{(X:\mathcal{U})} \mathsf{is\text{-}trunc}_k(X)$$

is a $(k+1)$-type, for any $k \geq -2$.
(d) Show that $\mathcal{U}^{\leq -1}$ is not a proposition.

(e) Show that $(\mathbf{2} \simeq \mathbf{2}) \simeq \mathbf{2}$, and conclude by the univalence axiom that the universe of sets $\mathcal{U}^{\leq 0}$ is not a set.

12.2 Use the univalence axiom to show that the type $\sum_{(P:\mathsf{Prop})} P$ is contractible.

12.3 Let A and B be small types.

(a) Construct an equivalence

$$(A \to (B \to \mathcal{U})) \simeq \left(\sum_{(S:\mathcal{U})} (S \to A) \times (S \to B)\right)$$

(b) We say that a relation $R : A \to (B \to \mathcal{U})$ is **functional** if it comes equipped with a term of type

$$\mathsf{is\text{-}function}(R) :\equiv \prod_{(x:A)} \mathsf{is\text{-}contr}\left(\sum_{(y:B)} R(x,y)\right)$$

For any function $f : A \to B$, show that the **graph** of f

$$\mathsf{graph}_f : A \to (B \to \mathcal{U})$$

given by $\mathsf{graph}_f(a,b) :\equiv (f(a) = b)$ is a functional relation from A to B.

(c) Construct an equivalence

$$\left(\sum_{(R:A\to(B\to\mathcal{U}))} \mathsf{is\text{-}function}(R)\right) \simeq (A \to B)$$

(d) Given a relation $R : A \to (B \to \mathcal{U})$ we define the **opposite relation**

$$R^{\mathsf{op}} : B \to (A \to \mathcal{U})$$

by $R^{\mathsf{op}}(y,x) :\equiv R(x,y)$. Construct an equivalence

$$\left(\sum_{(R:A\to(B\to\mathcal{U}))} \mathsf{is\text{-}function}(R) \times \mathsf{is\text{-}function}(R^{\mathsf{op}})\right) \simeq (A \simeq B).$$

12.4 (a) Show that $\mathsf{is\text{-}decidable}(P)$ is a proposition, for any proposition P.

(b) Show that $\mathsf{classical\text{-}Prop}$ is equivalent to $\mathbf{2}$.

12.5 Recall that $\mathcal{U}_*$ is the universe of pointed types.

(a) For any (A,a) and (B,b) in $\mathcal{U}_*$, write $(A,a) \simeq_* (B,b)$ for the type of **pointed equivalences** from A to B, i.e.,

$$(A,a) \simeq_* (B,b) :\equiv \sum_{(e:A\simeq B)} e(a) = b.$$

Show that the canonical map

$$\big((A,a) = (B,b)\big) \to \Big((A,a) \simeq (B,b)\Big)$$

sending $\mathsf{refl}_{(A,a)}$ to the pair $(\mathsf{id}, \mathsf{refl}_a)$, is an equivalence.

(b) Construct for any pointed type (X,x_0) an equivalence

$$\left(\sum_{(P:X\to\mathcal{U})} P(x_0)\right) \simeq \sum_{((A,a_0):\mathcal{U}_*)} (A,a_0) \to_* (X,x_0).$$

12.6 Show that any subuniverse is closed under equivalences, i.e., show that there is a map

$$(X \simeq Y) \to (P(X) \to P(Y))$$

for any subuniverse $P : \mathcal{U} \to \mathsf{Prop}$, and any $X, Y : \mathcal{U}$.

13 Groups in univalent mathematics

In this section we demonstrate a typical way to use the univalence axiom, showing that isomorphic groups can be identified. This is an instance of the *structure identity principle*, which is described in more detail in section 9.8 of [2]. We will see that in order to establish the fact that isomorphic groups can be identified, it has to be part of the definition of a group that its underlying type is a set. This is an important observation: in many branches of algebra the objects of study are *set-level* structures[1].

13.1 Semi-groups and groups

We introduce the type of groups in two stages: first we introduce the type of *semi-groups*, and then we introduce groups as semi-groups that possess further structure. It will turn out that this further structure is in fact a property, and this fact will help us to prove that isomorphic groups are equal.

Definition 13.1.1. A **semi-group** consists of a set G equipped with a term of type $\mathsf{has\text{-}associative\text{-}mul}(G)$, which is the type of pairs $(\mu_G, \mathsf{assoc}_G)$ consisting of a binary operation

$$\mu_G : G \to (G \to G)$$

and a homotopy

$$\mathsf{assoc}_G : \prod_{(x,y,z:G)} \mu_G(\mu_G(x,y),z) = \mu_G(x,\mu_G(y,z)).$$

We write Semi-Group for the type of all semi-groups in $\mathcal{U}$.

Definition 13.1.2. A semi-group G is said to be **unital** if it comes equipped with a **unit** $e_G : G$ that satisfies the left and right unit laws

$$\mathsf{left\text{-}unit}_G : \prod_{(y:G)} \mu_G(e_G, y) = y$$
$$\mathsf{right\text{-}unit}_G : \prod_{(x:G)} \mu_G(x, e_G) = x.$$

We write $\mathsf{is\text{-}unital}(G)$ for the type of such triples $(e_G, \mathsf{left\text{-}unit}_G, \mathsf{right\text{-}unit}_G)$. Unital semi-groups are also called **monoids**.

The unit of a semi-group is of course unique once it exists. In univalent mathematics we express this fact by asserting that the type $\mathsf{is\text{-}unital}(G)$ is a proposition for each semi-group G. In other words, being unital is a *property* of semi-groups rather than structure on it. This is typical for univalent mathematics: we express that a structure is a property by proving that this structure is a proposition.

Lemma 13.1.3. *For a semi-group G the type $\mathsf{is\text{-}unital}(G)$ is a proposition.*

[1] A notable exception is that of categories, which are objects at truncation level 1, i.e., at the level of *groupoids*. We will briefly introduce categories in §13.5. For more about categories we recommend Chapter 9 of [2].

Proof. Let G be a semi-group. Note that since G is a set, it follows that the types of the left and right unit laws are propositions. Therefore it suffices to show that any two terms $e, e' : G$ satisfying the left and right unit laws can be identified. This is easy:

$$e = \mu_G(e, e') = e'. \qquad \square$$

Definition 13.1.4. Let G be a unital semi-group. We say that G **has inverses** if it comes equipped with an operation $x \mapsto x^{-1}$ of type $G \to G$, satisfying the left and right inverse laws

$$\mathsf{left\text{-}inv}_G : \textstyle\prod_{(x:G)} \mu_G(x^{-1}, x) = e_G$$
$$\mathsf{right\text{-}inv}_G : \textstyle\prod_{(x:G)} \mu_G(x, x^{-1}) = e_G.$$

We write $\mathsf{is\text{-}group}'(G, e)$ for the type of such triples $((-)^{-1}, \mathsf{left\text{-}inv}_G, \mathsf{right\text{-}inv}_G)$, and we write

$$\mathsf{is\text{-}group}(G) :\equiv \textstyle\sum_{(e:\mathsf{is\text{-}unital}(G))} \mathsf{is\text{-}group}'(G, e)$$

A **group** is a unital semi-group with inverses. We write Group for the type of all groups in $\mathcal{U}$.

Lemma 13.1.5. *For any semi-group G the type $\mathsf{is\text{-}group}(G)$ is a proposition.*

Proof. We have already seen that the type $\mathsf{is\text{-}unital}(G)$ is a proposition. Therefore it suffices to show that the type $\mathsf{is\text{-}group}'(G, e)$ is a proposition for any $e : \mathsf{is\text{-}unital}(G)$.

Since a semi-group G is assumed to be a set, we note that the types of the inverse laws are propositions. Therefore it suffices to show that any two inverse operations satisfying the inverse laws are homotopic.

Let $x \mapsto x^{-1}$ and $x \mapsto \bar{x}^{-1}$ be two inverse operations on a unital semi-group G, both satisfying the inverse laws. Then we have the following identifications

$$\begin{aligned} x^{-1} &= \mu_G(e_G, x^{-1}) \\ &= \mu_G(\mu_G(\bar{x}^{-1}, x), x^{-1}) \\ &= \mu_G(\bar{x}^{-1}, \mu_G(x, x^{-1})) \\ &= \mu_G(\bar{x}^{-1}, e_G) \\ &= \bar{x}^{-1} \end{aligned}$$

for any $x : G$. Thus the two inverses of x are the same, so the claim follows. $\square$

Example 13.1.6. An important class of examples consists of **loop spaces** $x = x$ of a 1-type X, for any $x : X$. We will write $\Omega(X, x)$ for the loop space of X at x. Since X is assumed to be a 1-type, it follows that the type $\Omega(X, x)$ is a set. Then we have

$$\mathsf{refl}_x : \Omega(X, x)$$
$$\mathsf{inv} : \Omega(X, x) \to \Omega(X, x)$$
$$\mathsf{concat} : \Omega(X, x) \to (\Omega(X, x) \to \Omega(X, x)),$$

and these operations satisfy the group laws, since the group laws are just a special case of the groupoid laws for identity types, constructed in §5.2.

Example 13.1.7. The type $\mathbb{Z}$ of integers can be given the structure of a group, with the group operation being addition. The fact that $\mathbb{Z}$ is a set follows from Theorem 10.2.6 and Exercise 10.5. The group laws were shown in Exercise 7.11.

Example 13.1.8. Our last class of examples consists of the **automorphism groups** on sets. Given a set X, we define

$$\mathrm{Aut}(X) :\equiv (X \simeq X).$$

The group operation of $\mathrm{Aut}(X)$ is just composition of equivalences, and the unit of the group is the identity function. Note however, that although function composition is strictly associative and satisfies the unit laws strictly, composition of equivalences only satisfies the group laws up to identification because the proof that composites are equivalences is carried along.

Important special cases of the automorphism groups are the **symmetric groups**

$$\mathcal{S}_n :\equiv \mathrm{Aut}(\mathsf{Fin}(n)).$$

13.2 Homomorphisms of semi-groups and groups

Definition 13.2.1. Let G and H be semi-groups. A **homomorphismhomomorphism!of semi-groups** of semi-groups from G to H is a pair (f, μ_f) consisting of a function $f : G \to H$ between their underlying types, and a term

$$\mu_f : \textstyle\prod_{(x,y:G)} f(\mu_G(x,y)) = \mu_H(f(x), f(y))$$

witnessing that f preserves the binary operation of G. We will write

$$\mathsf{hom}(G, H)$$

for the type of all semi-group homomorphisms from G to H.

Remark 13.2.2. Since it is a property for a function to preserve the multiplication of a semi-group, it follows easily that equality of semi-group homomorphisms is equivalent to the type of homotopies between their underlying functions. In particular, it follows that the type of homomorphisms of semi-groups is a set.

Remark 13.2.3. The **identity homomorphism** on a semi-group G is defined to be the pair consisting of

$$\mathsf{id} : G \to G$$

$$\lambda x.\, \lambda y.\, \mathsf{refl}_{xy} : \textstyle\prod_{(x,y:G)} xy = xy.$$

Let $f : G \to H$ and $g : H \to K$ be semi-group homomorphisms. Then the composite function $g \circ f : G \to K$ is also a semi-group homomorphism, since we have the identifications

$$g(f(xy)) \jdeq g(f(x)f(y)) \jdeq g(f(x))g(f(y)).$$

Since the identity type of semi-group homomorphisms is equivalent to the type of homotopies between semi-group homomorphisms it is easy to see that semi-group homomorphisms satisfy the laws of a category, i.e., that we have the identifications

$$\mathsf{id} \circ f = f$$

$$g \circ \mathsf{id} = g$$
$$(h \circ g) \circ f = h \circ (g \circ f)$$

for any composable semi-group homomorphisms f, g, and h. Note, however that these equalities are not expected to hold judgmentally, since preservation of the semi-group operation is part of the data of a semi-group homomorphism.

Definition 13.2.4. Let G and H be groups. A **homomorphism** of groups from G to H is defined to be a semi-group homomorphism between their underlying semi-groups. We will write

$$\mathsf{hom}(G, H)$$

for the type of all group homomorphisms from G to H.

Remark 13.2.5. Since a group homomorphism is just a semi-group homomorphism between the underlying semi-groups, we immediately obtain the identity homomorphism, composition, and the category laws are satisfied.

13.3 Isomorphic semi-groups are equal

Definition 13.3.1. Let $h : \mathsf{hom}(G, H)$ be a homomorphism of semi-groups. Then h is said to be an **isomorphism** if it comes equipped with a term of type $\mathsf{is\text{-}iso}(h)$, consisting of triples (h^{-1}, p, q) consisting of a homomorphism $h^{-1} : \mathsf{hom}(H, G)$ of semi-groups and identifications

$$p : h^{-1} \circ h = \mathsf{id}_G \qquad \text{and} \qquad q : h \circ h^{-1} = \mathsf{id}_H$$

witnessing that h^{-1} satisfies the inverse lawsWe write $G \cong H$ for the type of all isomorphisms of semi-groups from G to H, i.e.,

$$G \cong H :\equiv \textstyle\sum_{(h:\mathsf{hom}(G,H))} \sum_{(k:\mathsf{hom}(H,G))} (k \circ h = \mathsf{id}_G) \times (h \circ k = \mathsf{id}_H).$$

If f is an isomorphism, then its inverse is unique. In other words, being an isomorphism is a property.

Lemma 13.3.2. *For any semi-group homomorphism $h : \mathsf{hom}(G, H)$, the type*

$$\mathsf{is\text{-}iso}(h)$$

is a proposition. It follows that the type $G \cong H$ is a set for any two semi-groups G and H.

Proof. Let k and k' be two inverses of h. In Remark 13.2.2 we have observed that the type of semi-group homomorphisms between any two semi-groups is a set. Therefore it follows that the types $h \circ k = \mathsf{id}$ and $k \circ h = \mathsf{id}$ are propositions, so it suffices to check that $k = k'$. In Remark 13.2.2 we also observed that the equality type $k = k'$ is equivalent to the type of homotopies $k \sim k'$ between their underlying functions. We construct a homotopy $k \sim k'$ by the usual argument:

$$k(y) \;\overset{}{=\!=}\; k(h(k'(y)) \;\overset{}{=\!=}\; k'(y). \qquad \square$$

Lemma 13.3.3. *A semi-group homomorphism $h : \mathsf{hom}(G, H)$ is an isomorphism if and only if its underlying map is an equivalence. Consequently, there is an equivalence*

$$(G \cong H) \simeq \sum_{(e:G\simeq H)} \prod_{(x,y:G)} e(\mu_G(x,y)) = \mu_H(e(x), e(y))$$

Proof. If $h : \mathsf{hom}(G, H)$ is an isomorphism, then the inverse semi-group homomorphism also provides an inverse of the underlying map of h. Thus we obtain that h is an equivalence. The standard proof showing that if the underlying map $f : G \to H$ of a group homomorphism is invertible then its inverse is again a group homomorphism also works in type theory. □

Definition 13.3.4. Let G and H be a semi-groups. We define the map

$$\mathsf{iso\text{-}eq} : (G = H) \to (G \cong H)$$

by path induction, taking refl_G to isomorphism id_G.

Theorem 13.3.5. *The map*

$$\mathsf{iso\text{-}eq} : (G = H) \to (G \cong H)$$

is an equivalence for any two semi-groups G and H.

Proof. By the fundamental theorem of identity types Theorem 9.2.2 it suffices to show that the total space

$$\sum_{(G':\mathsf{Semi\text{-}Group})} G \cong G'$$

is contractible. Since the type of isomorphisms from G to G' is equivalent to the type of equivalences from G to G' it suffices to show that the type

$$\sum_{(G':\mathsf{Semi\text{-}Group})} \sum_{(e:G\simeq G')} \prod_{(x,y:G)} e(\mu_G(x,y)) = \mu_{G'}(e(x), e(y)))$$

is contractible[2]. Since Semi-Group is the Σ-type

$$\sum_{(G':\mathsf{Set})} \mathsf{has\text{-}associative\text{-}mul}(G'),$$

it suffices to show that the types

$$\sum_{(G':\mathsf{Set})} G \simeq G'$$

$$\sum_{(\mu':\mathsf{has\text{-}associative\text{-}mul}(G))} \prod_{(x,y:G)} \mu_G(x,y) = \mu'(x,y)$$

is contractible. The first type is contractible by the univalence axiom. The second type is contractible by function extensionality. □

Corollary 13.3.6. *The type* Semi-Group *is a* 1*-type.*

Proof. It is straightforward to see that the type of group isomorphisms $G \cong H$ is a set, for any two groups G and H. □

[2] In order to show that a type of the form

$$\sum_{((x,y):\sum_{(x:A)} B(x))} \sum_{(z:C(x))} D(x,y,z)$$

is contractible, a useful strategy is to first show that the type $\sum_{(x:A)} C(x)$ is contractible. Once this is established, say with center of contraction (x_0, z_0), it suffices to show that the type $\sum_{(y:B(x_0))} D(x_0, y, z_0)$ is contractible.

13.4 Isomorphic groups are equal

Analogously to the map iso-eq of semi-groups, we have a map iso-eq of groups. Note, however, that the domain of this map is now the identity type $G = H$ of the *groups* G and H, so the maps iso-eq of semi-groups and groups are not exactly the same maps.

Definition 13.4.1. Let G and H be groups. We define the map

$$\mathsf{iso\text{-}eq} : (G = H) \to (G \cong H)$$

by path induction, taking refl_G to the identity isomorphism $\mathsf{id} : G \cong G$.

Theorem 13.4.2. *For any two groups G and H, the map*

$$\mathsf{iso\text{-}eq} : (G = H) \to (G \cong H)$$

is an equivalence.

Proof. Let G and H be groups, and write UG and UH for their underlying semi-groups, respectively. Then we have a commuting triangle

$$\begin{array}{ccc} (G = H) & \xrightarrow{\mathsf{ap}_{\mathsf{pr}_1}} & (UG = UH) \\ & {\scriptstyle \mathsf{iso\text{-}eq}} \searrow \quad \swarrow {\scriptstyle \mathsf{iso\text{-}eq}} & \\ & (G \cong H) & \end{array}$$

Since being a group is a property of semi-groups it follows that the projection map $\mathsf{Group} \to \mathsf{Semi\text{-}Group}$ forgetting the unit and inverses, is an embedding. Thus the top map in this triangle is an equivalence. The map on the right is an equivalence by Theorem 13.3.5, so the claim follows by the 3-for-2 property. □

Corollary 13.4.3. *The type of groups is a* 1*-type.*

13.5 Categories in univalent mathematics

In our proof of the fact that isomorphic groups are equal we have made extensive use of the notion of group homomorphism. What we have shown, in fact, is that there is a category of groups which is *Rezk complete* in the sense that the type of isomorphisms between two objects is equivalent to the type of identifications between those objects. In this final section we briefly introduce the notion of Rezk complete category. There are many more examples of categories, such as the categories of rings, or modules over a ring.

Definition 13.5.1. A **pre-category** $\mathcal{C}$ consists of

(i) A type A of **objects**.

(ii) For every two objects $x, y : A$ a set

$$\mathsf{hom}(x, y)$$

of **morphisms** from x to y.

(iii) For every object $x : A$ an **identity morphism**

$$\mathsf{id} : \mathsf{hom}(x, x)$$

(iv) For every two morphisms $f : \mathsf{hom}(x, y)$ and $g : \mathsf{hom}(y, z)$, a morphism

$$g \circ f : \mathsf{hom}(x, z)$$

called the **composition** of f and g.

(v) the following terms

$$\mathsf{left\text{-}unit}_{\mathcal{C}} : \mathsf{id} \circ f = f$$
$$\mathsf{right\text{-}unit}_{\mathcal{C}} : g \circ \mathsf{id} = g$$
$$\mathsf{assoc}_{\mathcal{C}} : (h \circ g) \circ f = h \circ (g \circ f)$$

witnessing that the category laws are satisfied.

Example 13.5.2. Since the type $X \to Y$ of functions between sets is again a set, we have a pre-category of sets.

Example 13.5.3. By Remarks 13.2.3 and 13.2.5 we have pre-categories of semi-groups and of groups.

Example 13.5.4. A pre-category satisfying the condition that every hom-set is a proposition is a **preorder**.

Definition 13.5.5. Given a pre-category $\mathcal{C}$, a morphism $f : \mathsf{hom}(x, y)$ is said to be an **isomorphism** if there exists a morphism $g : \mathsf{hom}(y, x)$ such that

$$g \circ f = \mathsf{id}$$
$$f \circ g\mathsf{id}.$$

We will write $\mathsf{iso}(x, y)$ for the type of all isomorphisms in $\mathcal{C}$ from x to y.

Remark 13.5.6. Just as in the case for semi-groups and groups, the condition that $f : \mathsf{hom}(x, y)$ is an isomorphism is a property of f.

Definition 13.5.7. A pre-category $\mathcal{C}$ is said to be **Rezk-complete** if the canonical map

$$(x = y) \to \mathsf{iso}(x, y)$$

is an equivalence for any two objects x and y of $\mathcal{C}$. Rezk-complete pre-categories are also called **categories**.

Example 13.5.8. The pre-category of sets is Rezk complete by the univalence axiom, so it is a category.

Example 13.5.9. The pre-categories of semi-groups and groups are Rezk-complete. Therefore they form categories.

Example 13.5.10. A pre-order is Rezk-complete if and only if it is anti-symmetric. In other words, a poset is precisely a category for which all the hom-sets are propositions. Thus, we see that the anti-symmetry axiom can be seen as a univalence axiom for pre-orders.

Exercises

13.1 Let X be a set. Show that the map

$$\mathsf{equiv\text{-}eq} : (X = X) \to (X \simeq X)$$

is a group isomorphism.

13.2 (a) Consider a group G. Show that the function

$$\mu_G : G \to (G \simeq G)$$

is an injective group homomorphism.

(b) Consider a pointed type A. Show that the concatenation function

$$\mathsf{concat} : \Omega(A) \to (\Omega(A) \simeq \Omega(A))$$

is an embedding.

13.3 Let $f : \mathsf{hom}(G, H)$ be a group homomorphism. Show that f preserves units and inverses, i.e., show that

$$f(e_G) = e_H$$
$$f(x^{-1}) = f(x)^{-1}.$$

13.4 Give a direct proof and a proof using the univalence axiom of the fact that all semi-group isomorphisms between unital semi-groups preserve the unit. Conclude that isomorphic monoids are equal.

13.5 Consider a monoid M with multiplication $\mu : M \to (M \to M)$ and unit e. Write

$$\bar{\mu} :\equiv \mathsf{fold\text{-}list}(e, \mu) : \mathsf{list}(M) \to M$$

for the iterated multiplication operation (see Exercise 4.8). Show that the square

$$\begin{array}{ccc} \mathsf{list}(\mathsf{list}(M)) & \xrightarrow{\mathsf{flatten\text{-}list}(M)} & \mathsf{list}(M) \\ {\scriptstyle \mathsf{list}(\bar{\mu})}\downarrow & & \downarrow{\scriptstyle \bar{\mu}} \\ \mathsf{list}(M) & \xrightarrow[\bar{\mu}]{} & M \end{array}$$

commutes.

13.6 Construct the category of posets.

14 The circle

We have seen inductive types, in which we describe a type by its constructors and an induction principle that allows us to construct sections of dependent types. Inductive types are freely generated by their constructors, which describe how we can construct their terms.

However, many familiar constructions in algebra involve the construction of algebras by generators and relations. For example, the free abelian group with two generators is described as the group with generators x and y, and the relation $xy = yx$.

In this chapter we introduce higher inductive types, where we follow a similar idea: to allow in the specification of inductive types not only *point constructors*, but also *path constructors* that give us relations between the point constructors. The ideas behind the definition of higher inductive types are introduced by studying the simplest non-trivial example: the *circle*.

14.1 The induction principle of the circle

The *circle* is defined as a higher inductive type $\mathbf{S}^1$ that comes equipped with

$$\mathsf{base} : \mathbf{S}^1$$

$$\mathsf{loop} : \mathsf{base} = \mathsf{base}.$$

Just like for ordinary inductive types, the induction principle for higher inductive types provides us with a way of constructing sections of dependent types. However, we need to take the *path constructor* loop into account in the induction principle.

By applying a section $f : \prod_{(x:\mathbf{S}^1)} P(x)$ to the base point of the circle, we obtain a term $f(\mathsf{base}) : P(\mathsf{base})$. Moreover, using the dependent action on paths of f of Definition 5.4.2 we also obtain for any dependent function $f : \prod_{(x:\mathbf{S}^1)} P(x)$ a path

$$\mathsf{apd}_f(\mathsf{loop}) : \mathsf{tr}_P(\mathsf{loop}, f(\mathsf{base})) = f(\mathsf{base})$$

in the fiber $P(\mathsf{base})$.

Definition 14.1.1. Let P be a type family over the circle. The **dependent action on generators** is the map

$$\mathsf{dgen}_{\mathbf{S}^1} : \left(\prod_{(x:\mathbf{S}^1)} P(x)\right) \to \left(\sum_{(y:P(\mathsf{base}))} \mathsf{tr}_P(\mathsf{loop}, y) = y\right) \tag{14.1}$$

given by $\mathsf{dgen}_{\mathbf{S}^1}(f) :\equiv (f(\mathsf{base}), \mathsf{apd}_f(\mathsf{loop}))$.

We now give the full specification of the circle.

Definition 14.1.2. The **circle** is a type $\mathbf{S}^1$ that comes equipped with

$$\mathsf{base} : \mathbf{S}^1$$

$$\mathsf{loop} : \mathsf{base} = \mathsf{base},$$

and satisfies the **induction principle of the circle**, which provides for each type family P over $\mathbf{S}^1$ a map

$$\mathsf{ind}_{\mathbf{S}^1} : \left(\sum_{(y:P(\mathsf{base}))} \mathsf{tr}_P(\mathsf{loop}, y) = y\right) \to \left(\prod_{(x:\mathbf{S}^1)} P(x)\right),$$

and a homotopy witnessing that $\mathsf{ind}_{\mathbf{S}^1}$ is a section of $\mathsf{dgen}_{\mathbf{S}^1}$

$$\mathsf{comp}_{\mathbf{S}^1} : \mathsf{dgen}_{\mathbf{S}^1} \circ \mathsf{ind}_{\mathbf{S}^1} \sim \mathsf{id}$$

for the computation rule.

Remark 14.1.3. The type of identifications $(y, p) = (y', p')$ in the type

$$\sum_{(y:P(\mathsf{base}))} \mathsf{tr}_P(\mathsf{loop}, y) = y$$

is equivalent to the type of pairs (α, β) consisting of an identification $\alpha : y = y'$, and an identification β witnessing that the square

$$\begin{array}{ccc} \mathsf{tr}_P(\mathsf{loop}, y) & \xrightarrow{\mathsf{ap}_{\mathsf{tr}_P(\mathsf{loop})}(\alpha)} & \mathsf{tr}_P(\mathsf{loop}, y') \\ {\scriptstyle p}\| & & \|{\scriptstyle p'} \\ y & \xrightarrow[\alpha]{} & y' \end{array}$$

commutes. Therefore it follows from the induction principle of the circle that for any $(y, p) : \sum_{(y:P(\mathsf{base}))} \mathsf{tr}_P(\mathsf{loop}, y) = y$, there is a dependent function $f : \prod_{(x:\mathbf{S}^1)} P(x)$ equipped with an identification

$$\alpha : f(\mathsf{base}) = y,$$

and an identification β witnessing that the square

$$\begin{array}{ccc} \mathsf{tr}_P(\mathsf{loop}, f(\mathsf{base})) & \xrightarrow{\mathsf{ap}_{\mathsf{tr}_P(\mathsf{loop})}(\alpha)} & \mathsf{tr}_P(\mathsf{loop}, y) \\ {\scriptstyle \mathsf{apd}_f(\mathsf{loop})}\| & & \|{\scriptstyle p} \\ f(\mathsf{base}) & \xrightarrow[\alpha]{} & y \end{array}$$

commutes.

14.2 The (dependent) universal property of the circle

Our goal is now to use the induction principle of the circle to derive the **universal property** of the circle. This universal property states that, for any type X the canonical map

$$\left(\mathbf{S}^1 \to X\right) \to \left(\sum_{(x:X)} x = x\right)$$

given by $f \mapsto (f(\mathsf{base}), \mathsf{ap}_f(\mathsf{loop}))$ is an equivalence. It turns out that it is easier to prove the **dependent universal property** first. The dependent universal property states that for any type family P over the circle, the canonical map

$$\left(\prod_{(x:\mathbf{S}^1)} P(x)\right) \to \left(\sum_{(y:P(\mathsf{base}))} \mathsf{tr}_P(\mathsf{loop}, y) = y\right)$$

given by $f \mapsto (f(\mathsf{base}), \mathsf{apd}_f(\mathsf{loop}))$ is an equivalence.

Theorem 14.2.1. *For any type family P over the circle, the map*

$$\left(\prod_{(x:\mathbf{S}^1)} P(x)\right) \to \left(\sum_{(y:P(\mathsf{base}))} \mathsf{tr}_P(\mathsf{loop}, y) = y\right)$$

given by $f \mapsto (f(\mathsf{base}), \mathsf{apd}_f(\mathsf{loop}))$ is an equivalence.

Proof. By the induction principle of the circle we know that the map has a section, i.e., we have

$$\mathsf{ind}_{\mathbf{S}^1} : \left(\textstyle\sum_{(y:P(\mathsf{base}))} \mathsf{tr}_P(\mathsf{loop}, y) = y\right) \to \left(\prod_{(x:\mathbf{S}^1)} P(x)\right)$$
$$\mathsf{comp}_{\mathbf{S}^1} : \mathsf{dgen}_{\mathbf{S}^1} \circ \mathsf{ind}_{\mathbf{S}^1} \sim \mathsf{id}$$

Therefore it remains to construct a homotopy

$$\mathsf{ind}_{\mathbf{S}^1} \circ \mathsf{dgen}_{\mathbf{S}^1} \sim \mathsf{id}.$$

Thus, for any $f : \prod_{(x:\mathbf{S}^1)} P(x)$ our task is to construct an identification

$$\mathsf{ind}_{\mathbf{S}^1}(\mathsf{dgen}_{\mathbf{S}^1}(f)) = f.$$

By function extensionality it suffices to construct a homotopy

$$\prod_{(x:\mathbf{S}^1)} \mathsf{ind}_{\mathbf{S}^1}(\mathsf{dgen}_{\mathbf{S}^1}(f))(x) = f(x).$$

We proceed by the induction principle of the circle using the family of types $E_{g,f}(x) :\equiv g(x) = f(x)$ indexed by $x : \mathbf{S}^1$, where g is the function

$$g :\equiv \mathsf{ind}_{\mathbf{S}^1}(\mathsf{dgen}_{\mathbf{S}^1}(f)).$$

Thus, it suffices to construct

$$\alpha : g(\mathsf{base}) = f(\mathsf{base})$$
$$\beta : \mathsf{tr}_{E_{g,f}}(\mathsf{loop}, \alpha) = \alpha.$$

An argument by path induction on p yields that

$$\left(\mathsf{apd}_g(p) \cdot r = \mathsf{ap}_{\mathsf{tr}_P(p)}(q) \cdot \mathsf{apd}_f(p)\right) \to \left(\mathsf{tr}_{E_{g,f}}(p, q) = r\right),$$

for any $f, g : \prod_{(x:X)} P(x)$ and any $p : x = x'$, $q : g(x) = f(x)$ and $r : g(x') = f(x')$. Therefore it suffices to construct an identification $\alpha : g(\mathsf{base}) = f(\mathsf{base})$ equipped with an identification β witnessing that the square

$$\begin{array}{ccc} \mathsf{tr}_P(\mathsf{loop}, g(\mathsf{base})) & \xrightarrow{\mathsf{ap}_{\mathsf{tr}_P(\mathsf{loop})}(\alpha)} & \mathsf{tr}_P(\mathsf{loop}, f(\mathsf{base})) \\ {\scriptstyle \mathsf{apd}_g(\mathsf{loop})}\Big\| & & \Big\|{\scriptstyle \mathsf{apd}_f(\mathsf{loop})} \\ g(\mathsf{base}) & \xrightarrow[\alpha]{} & f(\mathsf{base})'' \end{array}$$

commutes. Notice that we get exactly such a pair (α, β) from the computation rule of the circle, by Remark 14.1.3. □

As a corollary we obtain the following uniqueness principle for dependent functions defined by the induction principle of the circle.

Corollary 14.2.2. *Consider a type family P over the circle, and let*

$$y : P(\mathsf{base})$$
$$p : \mathsf{tr}_P(\mathsf{loop}, y) = y.$$

Then the type of functions $f : \prod_{(x:\mathbf{S}^1)} P(x)$ equipped with an identification

$$\alpha : f(\mathsf{base}) = y$$

and an identification β witnessing that the square

$$\begin{array}{ccc} \mathsf{tr}_P(\mathsf{loop}, f(\mathsf{base})) & \xrightarrow{\mathsf{ap}_{\mathsf{tr}_P(\mathsf{loop})}(\alpha)} & \mathsf{tr}_P(\mathsf{loop}, y) \\ {\scriptstyle \mathsf{apd}_f(\mathsf{loop})}\Big\| & & \Big\|{\scriptstyle p} \\ f(\mathsf{base}) & \xrightarrow[\alpha]{} & y \end{array}$$

commutes, is contractible.

Now we use the dependent universal property to derive the ordinary universal property of the circle. It would be tempting to say that it is a direct corollary, but we need to address the transport that occurs in the dependent universal property.

Theorem 14.2.3. *For each type X, the **action on generators***

$$\mathsf{gen}_{\mathbf{S}^1} : (\mathbf{S}^1 \to X) \to \textstyle\sum_{(x:X)} x = x$$

given by $f \mapsto (f(\mathsf{base}), \mathsf{ap}_f(\mathsf{loop}))$ is an equivalence.

Proof. We prove the claim by constructing a commuting triangle

$$\begin{array}{ccc} & (\mathbf{S}^1 \to X) & \\ {\scriptstyle \mathsf{gen}_{\mathbf{S}^1}}\swarrow & & \searrow{\scriptstyle \mathsf{dgen}_{\mathbf{S}^1}} \\ \left(\sum_{(x:X)} x = x\right) & \xrightarrow{\simeq} & \left(\sum_{(x:X)} \mathsf{tr}_{\mathsf{const}_X}(\mathsf{loop}, x) = x\right) \end{array}$$

in which the bottom map is an equivalence. Indeed, once we have such a triangle, we use the fact from Theorem 14.2.1 that $\mathsf{dgen}_{\mathbf{S}^1}$ is an equivalence to conclude that $\mathsf{gen}_{\mathbf{S}^1}$ is an equivalence.

To construct the bottom map, we first observe that for any constant type family const_B over a type A, any $p : a = a'$ in A, and any $b : B$, there is an identification

$$\mathsf{tr\text{-}const}_B(p, b) = b.$$

This identification is easily constructed by path induction on p. Now we construct the bottom map as the induced map on total spaces of the family of maps

$$l \mapsto \mathsf{tr\text{-}const}_X(\mathsf{loop}, x) \cdot l,$$

indexed by $x : X$. Since concatenating by a path is an equivalence, it follows by Theorem 9.1.3 that the induced map on total spaces is indeed an equivalence.

To show that the triangle commutes, it suffices to construct for any $f : \mathbf{S}^1 \to X$ an identification witnessing that the triangle

$$\begin{array}{ccc} \mathsf{tr}_{\mathsf{const}_X}(\mathsf{loop}, f(\mathsf{base})) & \xrightarrow{\mathsf{tr\text{-}const}_X(\mathsf{loop}, f(\mathsf{base}))} & f(\mathsf{base}) \\ & {\scriptstyle \mathsf{apd}_f(\mathsf{loop})} \searrow \quad \swarrow {\scriptstyle \mathsf{ap}_f(\mathsf{loop})} & \\ & f(\mathsf{base}) & \end{array}$$

commutes. This again follows from general considerations: for any $f : A \to B$ and any $p : a = a'$ in A, the triangle

$$\begin{array}{ccc} \mathsf{tr}_{\mathsf{const}_B}(p, f(a)) & \xrightarrow{\mathsf{tr\text{-}const}_B(p, f(a))} & f(a) \\ & {\scriptstyle \mathsf{apd}_f(p)} \searrow \quad \swarrow {\scriptstyle \mathsf{ap}_f(p)} & \\ & f(a') & \end{array}$$

commutes by path induction on p. □

Corollary 14.2.4. *For any loop $l : x = x$ in a type X, the type of maps $f : \mathbf{S}^1 \to X$ equipped with an identification*

$$\alpha : f(\mathsf{base}) = x$$

and an identification β witnessing that the square

$$\begin{array}{ccc} f(\mathsf{base}) & \xrightarrow{\alpha} & x \\ {\scriptstyle \mathsf{ap}_f(\mathsf{loop})}\downarrow & & \downarrow{\scriptstyle l} \\ f(\mathsf{base}) & \xrightarrow[\alpha]{} & x \end{array}$$

commutes, is contractible.

14.3 Multiplication on the circle

One way the circle arises classically, is as the set of complex numbers at distance 1 from the origin. It is an elementary fact that $|xy| = |x||y|$ for any two complex numbers $x, y \in \mathbb{C}$, so it follows that when we multiply two complex numbers that both lie on the unit circle, then the result lies again on the unit circle. Thus, using complex multiplication we see that there is a multiplication operation on the circle. And there is a shadow of this operation in type theory, even though our circle arises in a very different way!

Definition 14.3.1. We define a binary operation

$$\mathsf{mul}_{\mathbf{S}^1} : \mathbf{S}^1 \to (\mathbf{S}^1 \to \mathbf{S}^1).$$

Construction. Using the universal property of the circle, we define $\mathsf{mul}_{\mathbf{S}^1}$ as the unique map $\mathbf{S}^1 \to (\mathbf{S}^1 \to \mathbf{S}^1)$ equipped with an identification

$$\mathsf{base\text{-}mul}_{\mathbf{S}^1} : \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}) = \mathsf{id}$$

and an identification $\mathsf{loop\text{-}mul}_{\mathbf{S}^1}$ witnessing that the square

$$\begin{array}{ccc} \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}) & \xrightarrow{\mathsf{base\text{-}mul}_{\mathbf{S}^1}} & \mathsf{id} \\ {\scriptstyle \mathsf{ap}_{\mathsf{mul}_{\mathbf{S}^1}}(\mathsf{loop})}\Big\| & & \Big\|{\scriptstyle \mathsf{eq\text{-}htpy}(H)} \\ \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}) & \xrightarrow[\mathsf{base\text{-}mul}_{\mathbf{S}^1}]{} & \mathsf{id} \end{array}$$

commutes. Note that in this square we have a homotopy $H : \mathsf{id} \sim \mathsf{id}$, which is not yet defined. We use the dependent universal property of the circle with respect to the family $E_{\mathsf{id},\mathsf{id}}$ given by

$$E_{\mathsf{id},\mathsf{id}}(x) :\equiv (x = x),$$

to define H as the unique homotopy equipped with an identification

$$\alpha : H(\mathsf{base}) = \mathsf{loop}$$

and an identification β witnessing that the square

$$\begin{array}{ccc} \mathsf{tr}_{E_{\mathsf{id},\mathsf{id}}}(\mathsf{loop}, H(\mathsf{base})) & \xrightarrow{\mathsf{ap}_{\mathsf{tr}_{E_{\mathsf{id},\mathsf{id}}}(\mathsf{loop})}(\alpha)} & \mathsf{tr}_{E_{\mathsf{id},\mathsf{id}}}(\mathsf{loop}, \mathsf{loop}) \\ {\scriptstyle \mathsf{apd}_H(\mathsf{loop})}\Big\| & & \Big\|{\scriptstyle \gamma} \\ H(\mathsf{base}) & \xrightarrow[\alpha]{} & \mathsf{loop} \end{array}$$

commutes. Now it remains to define the path $\gamma : \mathsf{tr}_{E_{\mathsf{id},\mathsf{id}}}(\mathsf{loop}, \mathsf{loop}) = \mathsf{loop}$ in the above square. To proceed, we first observe that a simple path induction argument yields a function

$$\Big(p \cdot r = q \cdot p\Big) \to \Big(\mathsf{tr}_{E_{\mathsf{id},\mathsf{id}}}(p, q) = r\Big),$$

for any $p : \mathsf{base} = x$, $q : \mathsf{base} = \mathsf{base}$ and $r : x = x$. In particular, we have a function

$$\Big(\mathsf{loop} \cdot \mathsf{loop} = \mathsf{loop} \cdot \mathsf{loop}\Big) \to \Big(\mathsf{tr}_{E_{\mathsf{id},\mathsf{id}}}(\mathsf{loop}, \mathsf{loop}) = \mathsf{loop}\Big).$$

Now we apply this function to $\mathsf{refl}_{\mathsf{loop} \cdot \mathsf{loop}}$ to obtain the desired identification

$$\gamma : \mathsf{tr}_{E_{\mathsf{id},\mathsf{id}}}(\mathsf{loop}, \mathsf{loop}) = \mathsf{loop}.$$ □

Remark 14.3.2. In the definition of $H : \mathsf{id} \sim \mathsf{id}$ above, it is important that we didn't choose H to be $\mathsf{htpy\text{-}refl}$. If we had done so, the resulting operation would be homotopic to $x, y \mapsto y$, which is clearly not what we had in mind with the multiplication operation on the circle. See also Exercise 14.3.

The left unit law $\mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}, x) = x$ holds by the computation rule of the universal property. More precisely, we define

$$\mathsf{left\text{-}unit}_{\mathbf{S}^1} :\equiv \mathsf{htpy\text{-}eq}(\mathsf{base\text{-}mul}_{\mathbf{S}^1}).$$

For the right unit law, however, we need to give a separate argument that is surprisingly involved, because all the aspects of the definition of $\mathsf{mul}_{\mathbf{S}^1}$ will come out and play their part.

Theorem 14.3.3. *The multiplication operation on the circle satisfies the right unit law, i.e., we have*

$$\mathsf{mul}_{\mathbf{S}^1}(x, \mathsf{base}) = x$$

for any $x : \mathbf{S}^1$.

Proof. The proof is by induction on the circle. In the base case we use the left unit law

$$\mathsf{left\text{-}unit}_{\mathbf{S}^1}(\mathsf{base}) : \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}, \mathsf{base}) = \mathsf{base}.$$

Thus, it remains to show that

$$\mathsf{tr}_P(\mathsf{loop}, \mathsf{left\text{-}unit}_{\mathbf{S}^1}(\mathsf{base})) = \mathsf{left\text{-}unit}_{\mathbf{S}^1}(\mathsf{base}),$$

where P is the family over the circle given by

$$P(x) :\equiv \mathsf{mul}_{\mathbf{S}^1}(x, \mathsf{base}) = x.$$

Now we observe that there is a function

$$\Big(\mathsf{htpy\text{-}eq}(\mathsf{ap}_{\mathsf{mul}_{\mathbf{S}^1}}(p))(\mathsf{base}) \cdot r = q \cdot p\Big) \to \Big(\mathsf{tr}_P(p, q) = r\Big),$$

for any

$$\begin{aligned} p &: \mathsf{base} = x \\ q &: \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}, \mathsf{base}) = \mathsf{base} \\ r &: \mathsf{mul}_{\mathbf{S}^1}(x, \mathsf{base}) = x. \end{aligned}$$

Thus we see that, in order to construct an identification

$$\mathsf{tr}_P(\mathsf{loop}, \mathsf{left\text{-}unit}_{\mathbf{S}^1}) = \mathsf{left\text{-}unit}_{\mathbf{S}^1},$$

it suffices to show that the square

$$\begin{array}{ccc} \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}, \mathsf{base}) & \xrightarrow{\mathsf{left\text{-}unit}_{\mathbf{S}^1}(\mathsf{base})} & \mathsf{base} \\ {\scriptstyle \mathsf{htpy\text{-}eq}(\mathsf{ap}_{\mathsf{mul}_{\mathbf{S}^1}}(\mathsf{loop}))(\mathsf{base})}\Big\| & & \Big\|{\scriptstyle \mathsf{loop}} \\ \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base}, \mathsf{base}) & \xrightarrow[\mathsf{left\text{-}unit}_{\mathbf{S}^1}(\mathsf{base})]{} & \mathsf{base} \end{array}$$

commutes. Now we note that we have an identification $H(\mathsf{base}) = \mathsf{loop}$. It is indeed at this point, where it is important that H is not the trivial homotopy, because now we can proceed by observing that the above square commutes if and only if the square

$$\begin{array}{ccc} \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base},\mathsf{base}) & \xrightarrow{\mathsf{htpy\text{-}eq}(\mathsf{base\text{-}mul}_{\mathbf{S}^1})(\mathsf{base})} & \mathsf{base} \\ {\scriptstyle \mathsf{htpy\text{-}eq}(\mathsf{ap}_{\mathsf{mul}_{\mathbf{S}^1}}(\mathsf{loop}))(\mathsf{base})}\Big\| & & \Big\|{\scriptstyle H(\mathsf{base})} \\ \mathsf{mul}_{\mathbf{S}^1}(\mathsf{base},\mathsf{base}) & \xrightarrow[\mathsf{htpy\text{-}eq}(\mathsf{base\text{-}mul}_{\mathbf{S}^1})(\mathsf{base})]{} & \mathsf{base} \end{array}$$

commutes. The commutativity of this square easily follows from the identification $\mathsf{loop\text{-}mul}_{\mathbf{S}^1}$ constructed in Definition 14.3.1. □

Exercises

14.1 Let $P : \mathbf{S}^1 \to \mathsf{Prop}$ be a family of propositions over the circle. Show that

$$P(\mathsf{base}) \to \prod_{(x:\mathbf{S}^1)} P(x).$$

In this sense the circle is *connected*.

14.2 Show that

$$\prod_{(x,y:\mathbf{S}^1)} \neg\neg(x = y).$$

14.3 Show that for any type X and any $x : X$, the map

$$\mathsf{ind}_{\mathbf{S}^1}(x, \mathsf{refl}_x) : \mathbf{S}^1 \to X$$

is homotopic to the constant map const_x.

14.4 (a) Show that for any $x : \mathbf{S}^1$, both functions

$$\mathsf{mul}_{\mathbf{S}^1}(x, -) \qquad \text{and} \qquad \mathsf{mul}_{\mathbf{S}^1}(-, x)$$

are equivalences.

(b) Show that the function

$$\mathsf{mul}_{\mathbf{S}^1} : \mathbf{S}^1 \to (\mathbf{S}^1 \to \mathbf{S}^1)$$

is an embedding. Compare this fact with Exercise 13.2.

(c) Show that multiplication on the circle is associative and commutative.

14.5 (a) Show that a type X is a set if and only if the map

$$\lambda x.\, \lambda t.\, x : X \to (\mathbf{S}^1 \to X)$$

is an equivalence.

(b) Show that a type X is a set if and only if the map

$$\lambda f.\, f(\mathsf{base}) : (\mathbf{S}^1 \to X) \to X$$

is an equivalence.

15 The fundamental cover of the circle

In this lecture we show that the loop space of the circle is equivalent to $\mathbb{Z}$ by constructing the universal cover of the circle as an application of the univalence axiom.

15.1 Families over the circle

The type of small families over $\mathbf{S}^1$ is just the function type $\mathbf{S}^1 \to \mathcal{U}$, so in fact we may use the universal property of the circle to construct small dependent types over the circle. By the universal property, small type families over $\mathbf{S}^1$ are equivalently described as pairs (X, p) consisting of a type $X : \mathcal{U}$ and an identification $p : X = X$. This is where the univalence axiom comes in. By the map

$$\mathsf{eq\text{-}equiv}_{X,X} : (X \simeq X) \to (X = X)$$

it suffices to provide an equivalence $X \simeq X$.

Definition 15.1.1. Consider a type X and every equivalence $e : X \simeq X$. We will construct a dependent type $\mathcal{D}(X, e) : \mathbf{S}^1 \to \mathcal{U}$ with an equivalence $x \mapsto x_{\mathcal{D}} : X \simeq \mathcal{D}(X, e, \mathsf{base})$ for which the square

$$\begin{array}{ccc} X & \xrightarrow{\simeq} & \mathcal{D}(X, e, \mathsf{base}) \\ {\scriptstyle e}\downarrow & & \downarrow{\scriptstyle \mathsf{tr}_{\mathcal{D}(X,e)}(\mathsf{loop})} \\ X & \xrightarrow[\simeq]{} & \mathcal{D}(X, e, \mathsf{base}) \end{array}$$

commutes. We also write $d \mapsto d_X$ for the inverse of this equivalence, so that the relations

$$(x_{\mathcal{D}})_X = x \qquad (e(x)_{\mathcal{D}}) = \mathsf{tr}_{\mathcal{D}(X,e)}(\mathsf{loop}, x_{\mathcal{D}})$$
$$(d_X)_{\mathcal{D}} = d \qquad (\mathsf{tr}_{\mathcal{D}(X,e)}(d))_X = e(d_X)$$

hold.

The type $\sum_{(X:\mathcal{U})} X \simeq X$ is also called the type of **descent data** for the circle.

Construction. An easy path induction argument reveals that

$$\mathsf{equiv\text{-}eq}(\mathsf{ap}_P(\mathsf{loop})) = \mathsf{tr}_P(\mathsf{loop})$$

for each dependent type $P : \mathbf{S}^1 \to \mathcal{U}$. Therefore we see that the triangle

$$\begin{array}{ccc} & (\mathbf{S}^1 \to \mathcal{U}) & \\ {\scriptstyle \mathsf{gen}_{\mathbf{S}^1}}\swarrow & & \searrow{\scriptstyle \mathsf{desc}_{\mathbf{S}^1}} \\ \sum_{(X:\mathcal{U})} X = X & \xrightarrow[\mathsf{tot}(\lambda X.\, \mathsf{equiv\text{-}eq}_{X,X})]{} & \sum_{(X:\mathcal{U})} X \simeq X \end{array}$$

commutes, where the map $\mathsf{desc}_{\mathbf{S}^1}$ is given by $P \mapsto (P(\mathsf{base}), \mathsf{tr}_P(\mathsf{loop}))$ and the bottom map is an equivalence by the univalence axiom and Theorem 9.1.3. Now it follows by the 3-for-2 property that $\mathsf{desc}_{\mathbf{S}^1}$ is an equivalence, since $\mathsf{gen}_{\mathbf{S}^1}$ is an equivalence by

Theorem 14.2.3. This means that for every type X and every $e : X \simeq X$ there is a type family $\mathcal{D}(X, e) : \mathbf{S}^1 \to \mathcal{U}$ such that

$$(\mathcal{D}(X, e, \mathsf{base}), \mathsf{tr}_{\mathcal{D}(X,e)}(\mathsf{loop})) = (X, e).$$

Equivalently, we have $p : \mathcal{D}(X, e, \mathsf{base}) = X$ and $\mathsf{tr}(p, \mathsf{tr}_{\mathcal{D}(X,e)}(\mathsf{loop})) = e$. Thus, we obtain $\mathsf{equiv\text{-}eq}(p) : \mathcal{D}(X, e, \mathsf{base}) \simeq X$, for which the square

$$\begin{array}{ccc} \mathcal{D}(X, e, \mathsf{base}) & \xrightarrow{\mathsf{equiv\text{-}eq}(p)} & X \\ {\scriptstyle \mathsf{tr}_{\mathcal{D}(X,e)}(\mathsf{loop})}\downarrow & & \downarrow{\scriptstyle e} \\ \mathcal{D}(X, e, \mathsf{base}) & \xrightarrow[\mathsf{equiv\text{-}eq}(p)]{} & X \end{array}$$

commutes. □

15.2 The fundamental cover of the circle

The *fundamental cover* of the circle is a family of sets over the circle with contractible total space. Classically, the fundamental cover is described as a map $\mathbb{R} \to \mathbf{S}^1$ that winds the real line around the circle. In homotopy type theory there is no analogue of such a construction.

Recall from Exercise 7.6 that the successor function $\mathsf{succ} : \mathbb{Z} \to \mathbb{Z}$ is an equivalence. Its inverse is the predecessor function defined in Exercise 4.4.

Definition 15.2.1. The **fundamental cover** of the circle is the dependent type $\mathcal{E}_{\mathbf{S}^1} :\equiv \mathcal{D}(\mathbb{Z}, \mathsf{succ}) : \mathbf{S}^1 \to \mathcal{U}$.

Remark 15.2.2. The fundamental cover of the circle comes equipped with an equivalence

$$e : \mathbb{Z} \simeq \mathcal{E}_{\mathbf{S}^1}(\mathsf{base})$$

and a homotopy witnessing that the square

$$\begin{array}{ccc} \mathbb{Z} & \xrightarrow{e} & \mathcal{E}_{\mathbf{S}^1}(\mathsf{base}) \\ {\scriptstyle \mathsf{succ}}\downarrow & & \downarrow{\scriptstyle \mathsf{tr}_{\mathcal{E}_{\mathbf{S}^1}}(\mathsf{loop})} \\ \mathbb{Z} & \xrightarrow[e]{} & \mathcal{E}_{\mathbf{S}^1}(\mathsf{base}) \end{array}$$

commutes.

For convenience, we write $k_{\mathcal{E}}$ for the term $e(k) : \mathcal{E}_{\mathbf{S}^1}(\mathsf{base})$, for any $k : \mathbb{Z}$.

The picture of the fundamental cover is that of a helix over the circle. This picture emerges from the path liftings of loop in the total space. The segments of the helix connecting k to $k + 1$ in the total space of the helix, are constructed in the following lemma.

Lemma 15.2.3. *For any $k : \mathbb{Z}$, there is an identification*

$$\mathsf{segment\text{-}helix}_k : (\mathsf{base}, k_{\mathcal{E}}) = (\mathsf{base}, \mathsf{succ}(k)_{\mathcal{E}})$$

in the total space $\sum_{(t:\mathbf{S}^1)} \mathcal{E}(t)$.

Proof. By Theorem 7.3.4 it suffices to show that

$$\prod_{(k:\mathbb{Z})} \sum_{(\alpha:\mathsf{base}=\mathsf{base})} \mathsf{tr}_{\mathcal{E}}(\alpha, k_{\mathcal{E}}) = \mathsf{succ}(k)_{\mathcal{E}}.$$

We just take $\alpha :\equiv \mathsf{loop}$. Then we have $\mathsf{tr}_{\mathcal{E}}(\alpha, k_{\mathcal{E}}) = \mathsf{succ}(k)_{\mathcal{E}}$ by the commuting square provided in the definition of $\mathcal{E}$. □

15.3 Contractibility of general total spaces

Consider a type X, a family P over X, and a term $c : \sum_{(x:X)} P(x)$, and suppose our goal is to construct a contraction

$$\prod_{(t:\sum_{(x:X)} P(x))} c = t.$$

Of course, the first step is to apply the induction principle of Σ-types, so it suffices to construct a term of type

$$\prod_{(x:X)} \prod_{(y:P(x))} c = (x, y).$$

In the case where P is the fundamental cover of the circle, we are given an equivalence $e : \mathbb{Z} \simeq \mathcal{E}(\mathsf{base})$. Using this equivalence, we obtain an equivalence

$$\left(\prod_{(y:\mathcal{E}(y))} c = (\mathsf{base}, y)\right) \to \left(\prod_{(k:\mathbb{Z})} c = (\mathsf{base}, k_{\mathcal{E}})\right).$$

More generally, if we are given an equivalence $e : F \simeq P(x)$ for some $x : X$, then we have an equivalence

$$\left(\prod_{(y:P(x))} c = (x, y)\right) \to \left(\prod_{(y:F)} c = (x, e(y))\right) \tag{15.1}$$

by precomposing with the equivalence e. Therefore we can construct a term of type $\prod_{(y:P(x))} c = (x, y)$ by constructing a term of type $\prod_{(y:F)} c = (x, e(y))$.

Furthermore, if we consider a path $p : x = x'$ in X and a commuting square

$$\begin{array}{ccc} F & \xrightarrow{e} & P(x) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{tr}_P(p)} \\ F' & \xrightarrow{e'} & P(x') \end{array}$$

where e, e', and f are all equivalences, then we obtain a function

$$\psi : \left(\prod_{(y:F)} c = (x, e(y))\right) \to \left(\prod_{(y':F')} c = (x, e'(y'))\right).$$

The function ψ is constructed as follows. Given $h : \prod_{(y:F)} c = (x, e(y))$ and $y' : F'$ we have the path $h(f^{-1}(y')) : c = (x, e(f^{-1}(y')))$. Moreover, writing G for the homotopy $f \circ f^{-1} \sim \mathsf{id}$, we have the path

$$\mathsf{tr}_P(p, e(f^{-1}(y'))) \xrightarrow{H(f^{-1}(y'))} e'(f(f^{-1}(y'))) \xrightarrow{\mathsf{ap}_{e'}(G(y'))} e'(y').$$

From this concatenated path we obtain the path

$$(x, e(f^{-1}(y'))) \xrightarrow{\mathsf{eq\text{-}pair}(p, H(f^{-1}(y')) \cdot \mathsf{ap}_{e'}(G(y')))} (x', e'(y')).$$

Now we define the function ψ by

$$h \mapsto \lambda y'.\, h(f^{-1}(y')) \cdot \mathsf{eq\text{-}pair}(p, H(f^{-1}(y')) \cdot \mathsf{ap}_{e'}(G(y'))).$$

Note that ψ is an equivalence, since it is given as precomposition by the equivalence f^{-1}, followed by postcomposition by concatenation, which is also an equivalence. Now we state the main technical result of this section, which will help us prove the contractibility of the total space of the fundamental cover of the circle by computing transport in the family $x \mapsto \prod_{(y:P(x))} c = (x, y)$.

Definition 15.3.1. Consider a path $p : x = x'$ in X and a commuting square

$$\begin{array}{ccc} F & \xrightarrow{e} & P(x) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{tr}_P(p)} \\ F' & \xrightarrow{e'} & P(x') \end{array}$$

with $H : e' \circ f \ \mathsf{tr}_P(p) \circ e$, where e, e', and f are all equivalences. Then there is for any $y : F$ an identification

$$\mathsf{segment\text{-}tot}(y) : (x, e(y)) = (x', e'(f(y)))$$

defined as $\mathsf{segment\text{-}tot}(y) :\equiv \mathsf{eq\text{-}pair}(p, H(y)^{-1})$.

Lemma 15.3.2. *Consider a path $p : x = x'$ in X and a commuting square*

$$\begin{array}{ccc} F & \xrightarrow{e} & P(x) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{tr}_P(p)} \\ F' & \xrightarrow{e'} & P(x') \end{array}$$

with $H : e' \circ f \ \mathsf{tr}_P(p) \circ e$, where e, e', and f are all equivalences. Furthermore, let

$$h : \prod_{(y:F)} c = (x, e(y))$$
$$h' : \prod_{(y':F')} c = (x', e'(y')).$$

Then there is an equivalence

$$\left(\prod_{(y:F)} h'(f(y)) = h(y) \cdot \mathsf{segment\text{-}tot}(y)\right) \simeq \left(\mathsf{tr}_C(p, \varphi(h)) = \varphi'(h')\right).$$

Proof. We first note that we have a commuting square

$$\begin{array}{ccc} \prod_{(y:B(x))} c = (x, y) & \xrightarrow{-\circ e} & \prod_{(y:F)} c = (x, e(y)) \\ {\scriptstyle \mathsf{tr}_C(p)}\downarrow & & \uparrow{\scriptstyle \psi} \\ \prod_{(y':B(x'))} c = (x', y') & \xrightarrow[-\circ e']{} & \prod_{(y':F')} c = (x', e'(y')) \end{array}$$

where $\psi(h') = \lambda y.\, h'(f(y)) \cdot \mathsf{segment\text{-}tot}(y)^{-1}$. All the maps in this square are equivalences. In particular, the inverses of the top and bottom maps are φ and φ', respectively. The claim follows from this observation, but we will spell out the details.

Since any equivalence is an embedding, we see immediately that the type $\mathsf{tr}_C(p)(\varphi(h)) = \varphi'(h')$ is equivalent to the type

$$\psi(\mathsf{tr}_C(p)(\varphi(h)) \circ e') = \psi(\varphi'(h') \circ e').$$

By the commutativity of the square, the left hand side is h. The right hand side is $\psi(h')$. Therefore it follows that

$$\begin{aligned}\Big(\mathsf{tr}_C(p)(\varphi(h)) = \varphi'(h')\Big) &\simeq \Big(h = \lambda y.\, h'(f(y)) \cdot \mathsf{segment\text{-}tot}(y)^{-1}\Big) \\ &\simeq \Big(h' \circ f \sim (\lambda y.\, h(y) \cdot \mathsf{segment\text{-}tot}(y)\Big).\end{aligned}$$ □

Applying these observations to the fundamental cover of the circle, we obtain the following lemma that we will use to prove that the total space of $\mathcal{E}$ is contractible.

Corollary 15.3.3. *In order to show that the total space of $\mathcal{E}$ is contractible, it suffices to construct a function*

$$h : \textstyle\prod_{(k:\mathbb{Z})} (\mathsf{base}, 0_{\mathcal{E}}) = (\mathsf{base}, k_{\mathcal{E}})$$

equipped with a homotopy

$$H : \textstyle\prod_{(k:\mathbb{Z})} h(\mathsf{succ}(k)_{\mathcal{E}}) = h(k) \cdot \mathsf{segment\text{-}helix}(k).$$

In the next section we establish the dependent universal property of the integers, which we will use with Corollary 15.3.3 to show that the total space of the fundamental cover is contractible.

15.4 The dependent universal property of the integers

Lemma 15.4.1. *Let B be a family over $\mathbb{Z}$, equipped with a term $b_0 : B(0)$, and an equivalence*

$$e_k : B(k) \simeq B(\mathsf{succ}(k))$$

for each $k : \mathbb{Z}$. Then there is a dependent function $f : \prod_{(k:\mathbb{Z})} B(k)$ equipped with identifications $f(0) = b_0$ and

$$f(\mathsf{succ}(k)) = e_k(f(k))$$

for any $k : \mathbb{Z}$.

Proof. The map is defined using the induction principle for the integers, stated in Lemma 4.5.3. First we take

$$\begin{aligned} f(-1) &:\equiv e^{-1}(b_0) \\ f(0) &:\equiv b_0 \\ f(1) &:\equiv e(b_0). \end{aligned}$$

For the induction step on the negative integers we use

$$\lambda n.\, e^{-1}_{\mathsf{neg}(S(n))} : \textstyle\prod_{(n:\mathbb{N})} B(\mathsf{neg}(n)) \to B(\mathsf{neg}(S(n)))$$

For the induction step on the positive integers we use

$$\lambda n.\, e(\mathsf{pos}(n)) : \textstyle\prod_{(n:\mathbb{N})} B(\mathsf{pos}(n)) \to B(\mathsf{pos}(S(n))).$$

The computation rules follow in a straightforward way from the computation rules of $\mathbb{Z}$-induction and the fact that e^{-1} is an inverse of e. □

Example 15.4.2. For any type A, we obtain a map $f : \mathbb{Z} \to A$ from any $x : A$ and any equivalence $e : A \simeq A$, such that $f(0) = x$ and the square

$$\begin{array}{ccc} \mathbb{Z} & \xrightarrow{f} & A \\ {\scriptstyle \mathsf{succ}}\downarrow & & \downarrow{\scriptstyle e} \\ \mathbb{Z} & \xrightarrow[f]{} & A \end{array}$$

commutes. In particular, if we take $A \equiv (x = x)$ for some $x : X$, then for any $p : x = x$ we have the equivalence $\lambda q.\, p \cdot q : (x = x) \to (x = x)$. This equivalence induces a map

$$k \mapsto p^k : \mathbb{Z} \to (x = x),$$

for any $p : x = x$. This induces the **degree** k **map** on the circle

$$\mathsf{deg}(k) : \mathbf{S}^1 \to \mathbf{S}^1,$$

for any $k : \mathbb{Z}$, see Exercise 15.2.

In the following theorem we show that the dependent function constructed in Lemma 15.4.1 is unique.

Theorem 15.4.3. *Consider a type family $B : \mathbb{Z} \to \mathcal{U}$ equipped with $b : B(0)$ and a family of equivalences*

$$e : \textstyle\prod_{(k:\mathbb{Z})} B(k) \simeq B(\mathsf{succ}(k)).$$

Then the type

$$\textstyle\sum_{(f:\prod_{(k:\mathbb{Z})} B(k))} (f(0) = b) \times \prod_{(k:\mathbb{Z})} f(\mathsf{succ}(k)) = e_k(f(k))$$

is contractible.

Proof. In Lemma 15.4.1 we have already constructed a term of the asserted type. Therefore it suffices to show that any two terms of this type can be identified. Note that the type $(f, p, H) = (f', p', H')$ is equivalent to the type

$$\textstyle\sum_{(K:f\sim f')} (K(0) = p \cdot (p')^{-1}) \times \prod_{(k:\mathbb{Z})} K(\mathsf{succ}(k)) = (H(k) \cdot \mathsf{ap}_{e_k}(K(k))) \cdot H'(k)^{-1}.$$

We obtain a term of this type by applying Lemma 15.4.1 to the family C over $\mathbb{Z}$ given by $C(k) :\equiv f(k) = f'(k)$, which comes equipped with a base point

$$p \cdot (p')^{-1} : C(0),$$

and the family of equivalences

$$\lambda(\alpha : f(k) = f'(k)).\, (H(k) \cdot \mathsf{ap}_{e_k}(\alpha)) \cdot H'(k)^{-1} : \textstyle\prod_{(k:\mathbb{Z})} C(k) \simeq C(\mathsf{succ}(k)).$$ □

One way of phrasing the following corollary, is that $\mathbb{Z}$ is the 'initial type equipped with a point and an automorphism'.

Corollary 15.4.4. *For any type X equipped with a base point $x_0 : X$ and an automorphism $e : X \simeq X$, the type*

$$\sum_{(f:\mathbb{Z}\to X)} (f(0) = x_0) \times ((f \circ \mathsf{succ}) \sim (e \circ f))$$

is contractible.

15.5 The identity type of the circle

Lemma 15.5.1. *The total space $\sum_{(t:\mathbf{S}^1)} \mathcal{E}(t)$ of the fundamental cover of $\mathbf{S}^1$ is contractible.*

Proof. By Corollary 15.3.3 it suffices to construct a function

$$h : \prod_{(k:\mathbb{Z})} (\mathsf{base}, 0_{\mathcal{E}}) = (\mathsf{base}, k_{\mathcal{E}})$$

equipped with a homotopy

$$H : \prod_{(k:\mathbb{Z})} h(\mathsf{succ}(k)_{\mathcal{E}}) = h(k) \cdot \mathsf{segment\text{-}helix}(k).$$

We obtain h and H by the elimination principle of Lemma 15.4.1. Indeed, the family P over the integers given by $P(k) :\equiv (\mathsf{base}, 0_{\mathcal{E}}) = (\mathsf{base}, k_{\mathcal{E}})$ comes equipped with a term $\mathsf{refl}_{(\mathsf{base},0_{\mathcal{E}})} : P(0)$, and a family of equivalences

$$\prod_{(k:\mathbb{Z})} P(k) \simeq P(\mathsf{succ}(k))$$

given by $k, p \mapsto p \cdot \mathsf{segment\text{-}helix}(k)$. □

Theorem 15.5.2. *The family of maps*

$$\prod_{(t:\mathbf{S}^1)} (\mathsf{base} = t) \to \mathcal{E}(t)$$

sending $\mathsf{refl}_{\mathsf{base}}$ to $0_{\mathcal{E}}$ is a family of equivalences. In particular, the loop space of the circle is equivalent to $\mathbb{Z}$.

Proof. This is a direct corollary of Lemma 15.5.1 and Theorem 9.2.2. □

Corollary 15.5.3. *The circle is a 1-type and not a 0-type.*

Proof. To see that the circle is a 1-type we have to show that $s = t$ is a 0-type for every $s, t : \mathbf{S}^1$. By Exercise 14.1 it suffices to show that the loop space of the circle is a 0-type. This is indeed the case, because $\mathbb{Z}$ is a 0-type, and we have an equivalence $(\mathsf{base} = \mathsf{base}) \simeq \mathbb{Z}$.

Furthermore, since $\mathbb{Z}$ is a 0-type and not a (-1)-type, it follows that the circle is a 1-type and not a 0-type. □

Exercises

15.1 Show that the map

$$\mathbb{Z} \to \Omega(\mathbf{S}^1)$$

is a group homomorphism. Conclude that the loop space $\Omega(\mathbf{S}^1)$ as a group is isomorphic to $\mathbb{Z}$.

15.2 Use the fundamental cover of the circle to show that

$$\neg\Big(\prod_{(t:\mathbf{S}^1)} \mathsf{base} = t\Big).$$

15.3 (a) Show that for every $x : X$, we have an equivalence

$$\Big(\sum_{(f:\mathbf{S}^1\to X)} f(\mathsf{base}) = x\Big) \simeq (x = x)$$

(b) Show that for every $t : \mathbf{S}^1$, we have an equivalence

$$\Big(\sum_{(f:\mathbf{S}^1\to \mathbf{S}^1)} f(\mathsf{base}) = t\Big) \simeq \mathbb{Z}$$

The base point preserving map $f : \mathbf{S}^1 \to \mathbf{S}^1$ corresponding to $k : \mathbb{Z}$ is called the **degree** k **map** on the circle, and is denoted by $\mathsf{deg}(k)$.

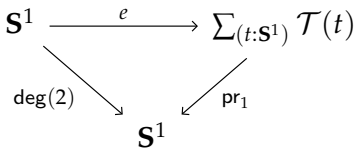
(c) Show that for every $t : \mathbf{S}^1$, we have an equivalence

$$\Big(\sum_{(e:\mathbf{S}^1\simeq \mathbf{S}^1)} e(\mathsf{base}) = t\Big) \simeq \mathbf{2}$$

15.4 The **(twisted) double cover** of the circle is defined as the type family $\mathcal{T} :\equiv \mathcal{D}(\mathbf{2}, \mathsf{neg}) : \mathbf{S}^1 \to \mathcal{U}$, where $\mathsf{neg} : \mathbf{2} \simeq \mathbf{2}$ is the negation equivalence of Exercise 7.5.

(a) Show that $\neg(\prod_{(t:\mathbf{S}^1)} \mathcal{T}(t))$.

(b) Construct an equivalence $e : \mathbf{S}^1 \simeq \sum_{(t:\mathbf{S}^1)} \mathcal{T}(t)$ for which the triangle



commutes.

15.5 Show that $(\mathbf{S}^1 \simeq \mathbf{S}^1) \simeq \mathbf{S}^1 + \mathbf{S}^1$. Conclude that a univalent universe containing a circle is not a 1-type.

15.6 (a) Construct a family of equivalences

$$\prod_{(t:\mathbf{S}^1)} \big((t = t) \simeq \mathbb{Z}\big).$$

(b) Use Exercise 14.5 to show that $(\mathsf{id}_{\mathbf{S}^1} \sim \mathsf{id}_{\mathbf{S}^1}) \simeq \mathbb{Z}$.

(c) Use Exercise 11.7 to show that

$$\mathsf{has\text{-}inverse}(\mathsf{id}_{\mathbf{S}^1}) \simeq \mathbb{Z},$$

and conclude that $\mathsf{has\text{-}inverse}(\mathsf{id}_{\mathbf{S}^1}) \not\simeq \mathsf{is\text{-}equiv}(\mathsf{id}_{\mathbf{S}^1})$.

15.7 Consider a map $i : A \to \mathbf{S}^1$, and assume that i has a retraction. Construct a term of type

$$\mathsf{is\text{-}contr}(A) + \mathsf{is\text{-}equiv}(i).$$

Bibliography

[1] Erret Bishop. *Foundations of constructive analysis*. New York: McGraw-Hill Book Co., 1967, pp. xiii+370.

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

Index