이학석사 학위논문

A Survey on Homotopy Type Theory with the Proof Assistant AGDA

증명보조기 AGDA 를 활용한 Homotopy Type Theory 에 대한 조사

2023년 12월

서울대학교 대학원

수리과학부 수리과학전공

윤 상 진

A Survey on Homotopy Type Theory with the Proof Assistant AGDA

증명보조기 AGDA 를 활용한 Homotopy Type Theory 에 대한 조사

지도교수 이 승 진

이 논문을 이학석사 학위논문으로 제출함

2023년 10월

서울대학교 대학원

수리과학부 수리과학전공

윤 상 진

윤상진의 이학석사 학위 논문을 인준함

2023년 12월

위 원 장 _____

부위원장 _____

위 원 _____

Abstract

Homotopy Type Theory (abbreviated to HoTT) is a new field of mathematics lying on the intersection of homotopy theory in mathematics and type theory in computer science. Type theory is a sort of formal system like first order logic (on which ZFC set theory is based), which originated from Russell's book, Principia Mathematica, and was developed independently by computer scientists and logicians to present. The word "type theory" may refer to the common framework shared by formal systems belonging to this field, or may refer to each individual formal system. HoTT is a formal system as one of those individual formal systems, and HoTT itself contains various phenomena that can be interpreted homotopically, enabling it to be a new foundation of mathematics related to homotopy theory. In particular, type theory has the advantage of being easy to be transplanted into computers, and through this many proofs of homotopy theory have already been formalized with the language of HoTT in proof assistants such as Agda and Coq. In summary, HoTT is a new formal system with the ease of computer implementation, which can synthetically explore homotopy theory. Since the informal descriptions of this new theory have already been introduced in good textbooks, this paper addresses how the definitions and theorems of this formal system can be formalized in the Agda proof assistant. Especially, it aims to formalize that the fundamental group of S^1 is equal to $\mathbb{Z}$.

Keywords: Homotopy Type Theory, Proof Assistant, Agda

Student Number: 2022-27298

Contents

Abstract i

1 Introduction 1

2 Dependent Type Theory 3

2.1 Basic terminologies and structural rules . . . 3

2.2 Type constructors . . . 4

2.2.1 Dependent product Π . . . 6

2.2.2 Dependent sum Σ . . . 9

2.2.3 Binary disjoint sum + . . . 10

2.2.4 Empty type $\mathbb{0}$ . . . 11

2.2.5 Unit type $\mathbb{1}$ . . . 12

2.2.6 Inductive types . . . 13

2.2.7 Identity types _=_ . . . 14

3 Basic Constructions 23

3.1 Homotopy between functions . . . 23

3.2 Equivalence . . . 25

3.3 h-level . . . 29

4 Additional Rules in HoTT 41

4.1 Function extensionality . . . 41

4.2 Univalence axiom . . . 42

4.3 Higher inductive types . . . 45

4.3.1 Homotopy pushout . . . 45

4.3.2 Suspension, Spheres, and the circle . . . 54

5 The Fundamental Group of The Circle **62**

6 Conclusion **68**

Bibliography **68**

요 약 **70**

Chapter 1

Introduction

Modern mathematicians are formalists, but real mathematics is not done that way. It is our practice to regard something as valid if it is stated in a way that can be convinced by all mathematically well-trained people. It has already been confirmed in Rusell's work in modern times that the complete formalization of mathematics is virtually impossible by human hands. However, since mathematics is a discipline where even the slightest mistake can be fatal, such a convention seems quite unstable. Indeed, it was the minor error detected in his paper that immersed Voevodsky, one of the significant founders of this new field, in the work of developing the new foundation of mathematics (The anecdote is well introduced in his essay - [10]). With the development of modern computers and the advent of proof assistants, the complete formalization of mathematics has descended from 'virtually impossible' to 'labor-intensive but sufficiently possible,' and in particular, HoTT makes each type, which is its primitive objects, an infinity-groupoid hence enables a natural description of homotopy theory.

The first-order logic system has a model of set theory, which takes the membership relation as its predicate, and set theory can be interpreted in the category of infinity-groupoids as a full subcategory of groupoids without interesting higher structures. Therefore, HoTT, a syntax for infinity-groupoids, can have multiple meanings - it can be interpreted logically, set theoretically, and homotopically. A type in HoTT, its primitive object, is a proposition, or a set, or a space(its fundamental infinity groupoid). A term of a type is a proof of a proposition, or an element of a set, or a point in a space.

HoTT is also a formal system as one of type theories. In type theories, a constructed term of a type contains all information of what inference rules the term is constructed

through. Therefore, a term serves as a label recording which inference rules were applied and the order of those applications. Hence, HoTT can be viewed as a formal system forcing any witness of a mathematical statement to be formally checked for validity.

It may seem somewhat inefficient to use a system forcing one to provide formal proof in developing complex mathematical theories. However, as HoTT's framework is considerably close to our intuition from algebraic topology, it is possible to conduct sufficiently 'informal' reasoning activities through HoTT, which is no different from the fact that we do not pursue complete formalism in the ZFC system. HoTT has the additional advantage that complete formalization can be achieved with a considerable (but feasible) amount of labor, with the help of a proof assistant. Moreover, it is known that HoTT is a syntax for a general theory of mathematics with higher structures beyond the classical homotopy theory of topological spaces.

This paper merely shows how some parts of the contents in textbooks [9] and [5] can be formalized in Agda. Also, many notations for the Agda code are brought from [2]. In Chapter 2, we specify the dependent type theory, which forms the basis of HoTT. Chapter 3 defines basic notions such as homotopy, equivalence, and h-levels. All components that constitute HoTT are fully specified in Chapter 4 by declaring the univalence axiom and some higher inductive types. Finally, we prove that the fundamental group of S^1 is equal to $\mathbb{Z}$ in Chapter 5.

Chapter 2

Dependent Type Theory

Homotopy type theory consists of dependent type theory, univalence axiom, and higher inductive types. In this chapter, we define the dependent type system.

2.1 Basic terminologies and structural rules

A formal system consists of criteria for verifying whether a given formula(character string) is well-formed and rules for deforming a well-formed formula to another well-formed formula. The system should be defined as one where any well-formedness of a formula and well-derivedness of a sequence of deformations could be checked using a machine with perfect reading, erasing, substituting, and parsing functionalities. We start with the criteria for the well-formedness of a formula.

We first take the following primitive formulas of 4 kinds:

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

which are called **judgements**. We respectively pronounce them by "under the context Γ, A is a type, A and A' are judgementally equal types, a is a term of a type A, a and a' are judgementally equal terms of a type A". In the above forms, Γ represents a finite tuple of the forms $(x_i : A_i)_{i<n}$, where (x_i) are distinctive free variables, with an additional sequence of $(n-1)$-judgements which is that for each $i < n$, $(x_j : A_j)_{j<i} \vdash A_i$ type. We call such Γ a **context**, and denote it as $\vdash \Gamma$ ctxt. We regard the empty string as a well-formed context. A context is well-formed if all judgements $(x_j : A_j)_{j<i} \vdash A_i$ type are derivable from empty hypothesis. A judgement $\Gamma \vdash \mathcal{J}$ is well-formed if the context is

well-formed where $\mathcal{J}$ is one of the 4 kinds of judgements. We regard the left part of $\vdash$ as an assumption and the right part as a conclusion.

We have jointly defined judgment and contexts. Now we define **structural rules** which become a basis for various type theories.

Rules for judgemental equality:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash A \equiv A \text{ type}} \quad \frac{\Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash B \equiv A \text{ type}} \quad \frac{\Gamma \vdash A \equiv B \text{ type} \quad \Gamma \vdash B \equiv C \text{ type}}{\Gamma \vdash A \equiv C \text{ type}}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \quad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A} \quad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}$$

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

Variable, **weakening**, and **substitution** rules:

$$\frac{\vdash \Gamma, x : A, \Delta \text{ctxt}}{\Gamma, x : A, \Delta \vdash x : A}\text{V} \quad \frac{\Gamma \vdash A \text{ type} \quad \Gamma, \Delta \vdash \mathcal{J}}{\Gamma, x : A, \Delta \vdash \mathcal{J}}\text{W} \quad \frac{\Gamma \vdash a : A \quad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, \Delta[a/x] \vdash \mathcal{J}[a/x]}\text{S}$$

$$\frac{\Gamma \vdash a \equiv a' : A \quad \Gamma, x : A, \Delta \vdash B \text{ type}}{\Gamma, \Delta[a/x] \vdash B[a/x] \equiv B[a'/x] \text{ type}}\text{S-cong-type} \quad \frac{\Gamma \vdash a \equiv a' : A \quad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \equiv b[a'/x] : B[a/x]}\text{S-cong-term}$$

2.2 Type constructors

The remaining rules of the dependent type theory are called **logical rules** which are defined in this section.

Those rules share the following framework:

- Formation rule - declares a certain type under a suitable context
- Introduction rule - declares how one can construct a term for the new type
- Elimination rule - Shows how to use a term of the type, or construct a section over the type
- Computation rule - Gives a judgemental equality relating introduction rule and elimination rule

In each rule, we have a corresponding congruence rule stating that the rule is compatible with a judgemental equality of each argument. For example, we have the following rule for the Π-formation rule:

$$\frac{\Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma,\ x : A \vdash B(x) \equiv B'(x) \text{ type}}{\Gamma \vdash \Pi_{x:A} B(x) \equiv \Pi_{x:A'} B'(x)}\ \Pi\text{-form-cong}$$

Although not written down, a corresponding congruence rule is assumed for each rule.

As mentioned in Chapter 1, our type theory can be interpreted in various layers. It can be seen as the syntax for logic, set theory, and homotopy theory. We present how each type constructor can be interpreted at each layer, and from now on, we develop the content by implementing each type constructor and its related fundamental constructions in the proof assistant Agda.

Agda is an interactive proof assistant, and information on installation and basic usage can be easily found on the Internet. We only need some commands for making holes and C-c C-l, C-c C-c, C-c C-, C-c C-=, and C-c C-space. We start our Agda document with flags –without-K and –rewriting. Related Agda files can be found on [11].

In Agda, we have universes of types and they have a hierarchy structure primitively defined in Agda. The universe of types of Level ℓ is written as `type ℓ`. The type of all universe levels is written as `Level`. There is the least element `ℓ0` and it has two operations `lsuc`, taking the successor, and `_⊔_`, taking the least upper bound of two input arguments. For any level `ℓ`, `type ℓ` is a term of `type (lsuc ℓ)`. We assume that for any given type A, there uniquely exists a level `ℓ` such that A : `type ℓ`. Moreover, we assume that for any type family $x : A \vdash B(x)$ type, there uniquely exists a level `ℓ'` such that $x : A \vdash B(x)$: `type ℓ'`, or at least we are only interested in such type families. We denote the minimal level type universe `type ℓ0` as $\mathcal{U}$ and call it the base universe. We will introduce the univalence axiom only on this base universe.

These assumptions can be introduced to our Agda file as below:

```
open import Agda.Primitive public
  renaming ( Set to type
           ; lzero to ℓ0
           ; Setω to typeω
           )
  using    (Level ; lsuc ; _⊔_)

𝒰 : type (lsuc ℓ0)
𝒰 = type ℓ0
```

All type constructors in Agda will be defined in a way that if the given type (or type family) arguments are of level $\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_n$, the resulting type resides in type $(\boldsymbol{\ell}_1 \sqcup \ldots \sqcup \boldsymbol{\ell}_n)$. That is, we can additionally assume that for any judgement of the form

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash B(x_1, \ldots, x_n) \text{ type}$$

where A_i : `type` $\boldsymbol{\ell}_i$, the type family B resides in the universe `type` $\boldsymbol{\ell}_1 \sqcup \ldots \sqcup \boldsymbol{\ell}_n$.

It would be inefficient if one had to specify the type level each time a type is introduced. We can avoid this annoyance by using the variable command:

```
variable
  ℓ ℓ' ℓ'' ℓ''' : Level
```

Now we don't have to write "`ℓ : Level`". The levels are automatically regarded as implicit arguments in each construction.

In HoTT textbooks like [9] and [5], universes are assumed to be cumulative, that is, for any level $\boldsymbol{\ell}_1$ and $\boldsymbol{\ell}_2$ with $\boldsymbol{\ell}_1 \leq \boldsymbol{\ell}_2$, `A : type` $\boldsymbol{\ell}_1$ implies `A : type` $\boldsymbol{\ell}_2$. However, to seek full generality of our Agda files, we don't assume this condition. If one wants to assume the cumulativity of universes, one should manually define some record types since it is not primitively supported by Agda. Without the cumulativity, it can be possible to get hindered when we are trying to define some type families. However, for problems in this paper, we can find a roundabout way not using the cumulativity.

For a more rigorous description of type universes, see [5].

2.2.1 Dependent product Π

For a given judgement $\Gamma,\ x : A \vdash B(x)$ type , we call B a **type family over** A **under the context** Γ. For a judgement $\Gamma,\ x : A \vdash b(x) : B(x)$, we call b a **section of** B. The **dependent product** type constructor Π is defined as rules for constructing a space of sections and its evaluation.

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \Pi_{x:A} B(x) \text{ type}}\Pi\text{-form} \qquad \frac{\Gamma, x : A \vdash B(x) \text{ type} \quad \Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.b(x) : \Pi_{x:A} B(x)}\Pi\text{-intro}$$

$$\frac{\Gamma \vdash f : \Pi_{x:A} B(x) \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B(a)}\Pi\text{-elim}$$

$$\frac{\Gamma, x : A \vdash B(x) \text{ type} \quad \Gamma, x : A \vdash b(x) : B(x) \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x : A.b(x))(a) \equiv b(a) : B(a)}\Pi\text{-comp-}\beta$$

$$\frac{\Gamma \vdash f : \Pi_{x:A}B(x)}{\Gamma \vdash \lambda x.f(x) \equiv f : \Pi_{x:A}B(x)}\Pi\text{-comp-}\eta$$

For the special case when B is non-dependent over A, that is, if we have two judgements $\Gamma \vdash A$ type and $\Gamma \vdash B$ type, we write $A \to B$ for $\Pi_{x:A}B(x)$.

One can interpret type theory logically, regarding the type family B as a predicate over the domain A. In this interpretation, the dependent product type can be interpreted as a proposition $\forall x : A.B(x)$, and a section of it is taken as a proof of it. For a non-dependent case, one can also interpret the rules in propositional logic by considering each type as a proposition and their term as a proof of it. Then $A \to B$ is the logical implication; the elimination rule says that if A holds then B holds.

One can also interpret type theory in set theory or homotopy theory. If we regard each type as a set(space), then each $B(x)$ is a set indexed by $x \in A$ (fiber over $x \in A$). Then the type $\Pi_{x:A}B(x)$ is interpreted as the space of sections. For a non-dependent case, $A \to B$ is the function space.

Now we observe how the Π-constructor can be represented in Agda. In fact, dependent product type is a built-in constituent of Agda, hence we don't even need to define it. A dependent product type $\Pi_{x:A}\mathcal{P}(x)$ is represented as $(x : A) \to \mathcal{P}\ x$. Note that for a given type family $\mathcal{P}$, its base type A can be inferred from it. In such a case, we can set the inferable argument implicit in Agda as follows:

```
Π :{A : type ℓ} → (A → type ℓ') → type (ℓ ⊔ ℓ')
Π {A = A} 𝒫 = (x : A) → 𝒫 x
```

An argument presented inside brace { } is regarded as implicit. Hence we can just write as Π $\mathcal{P}$. We can also make the base type explicit and use it to match the notation with the Π-type constructor in our theory:

```
-Π : (A : type ℓ) (𝒫 : A → type ℓ') → type (ℓ ⊔ ℓ')
-Π A 𝒫 = Π 𝒫
syntax -Π A (λ x → 𝒫x) = Π x : A , 𝒫x
```

-Π takes the base type of $\mathcal{P}$ explicitly (which we will never use again) and we define a new syntax through -Π, which coincides with the notation of our dependent type theory.

Note that the usual character ":" on our keyboard is the reserved character of Agda, which is prohibited to be written except for declaring a free variable or the start-line of construction. The character "`:`" in the expression "Π `x : A`" is the one that can be written by the Agda input method "\ : 4".

Identity function and function composition can be simply constructed in Agda as follows:

```
id : {X : type ℓ} → X → X
id x = x


_∘_ : {X : type ℓ} {Y : type ℓ'} {𝒫 : (y : Y) → type ℓ''}
    → ((y : Y) → 𝒫 y) → (f : X → Y) → ((x : X) → 𝒫 (f x))
g ∘ f = λ x → g (f x)
infixl 70 _∘_
```

The expression `infixl 70` sets the priority of the operator and declares that $\circ$ is left associative.

For the first and last time, we present a formal derivation of function composition to show how much labor can be waivered by the proof assistant. First, let $\mathcal{H}$ be a hypothesis consisting of the following three judgements:

$$\Gamma \vdash X \text{ type} \qquad \Gamma \vdash Y \text{ type} \qquad \Gamma,\ y : Y \vdash \mathcal{P}(y) \text{ type}$$

then we get the following derivation, which we label as Lemma,

$$\frac{\mathcal{H}}{\Gamma,\ g : \Pi_{y:Y}\mathcal{P}(y),\ f : X \to Y,\ x : X \text{ ctxt}}\ \text{Lemma}$$

from the below proof tree:

$$\dfrac{\dfrac{\dfrac{\Gamma,\ y : Y \vdash \mathcal{P}(y) \text{ type}}{\Gamma \vdash \Pi_{y:Y}\mathcal{P}(y) \text{ type}}\ \Pi\text{-form} \qquad \dfrac{\dfrac{\Gamma \vdash X \text{ type} \qquad \Gamma \vdash Y \text{ type}}{\Gamma \vdash X \to Y \text{ type}}\ \Pi\text{-form} \qquad \Gamma \vdash X \text{ type}}{\Gamma,\ f : X \to Y \vdash X \text{ type}}\ \text{W}}{\Gamma,\ g : \Pi_{y:Y}\mathcal{P}(y),\ f : X \to Y \vdash X \text{ type}}\ \text{W}}{\Gamma,\ g : \Pi_{y:Y}\mathcal{P}(y),\ f : X \to Y,\ x : X \text{ ctxt}}$$

Now the formal construction of function composition is given as:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{H}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\vdash g:\Pi_{y:Y}\mathcal{P}(y)}\ \text{(a)} \quad \dfrac{\mathcal{H}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\vdash f(x):Y}\ \text{(b)}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\vdash g(f(x)):\mathcal{P}(f(x))}\ \Pi\text{-elim}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y\vdash \lambda x.g(f(x)):\Pi_{x:X}\mathcal{P}(f(x))}\ \Pi\text{-intro}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y)\vdash \lambda f.\lambda x.g(f(x)):\Pi_{f:X\to Y}\Pi_{x:X}\mathcal{P}(f(x))}\ \Pi\text{-intro}}{\Gamma\vdash \lambda g.\lambda f.\lambda x.g(f(x)):\Pi_{g:\Pi_{y:Y}\mathcal{P}(y)}\Pi_{f:X\to Y}\Pi_{x:X}\mathcal{P}(f(x))}\ \Pi\text{-intro}$$

where derivations (a) and (b) are given respectively as:

$$\dfrac{\dfrac{\mathcal{H}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\ \text{ctxt}}\ \text{Lemma}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\vdash g:\Pi_{y:Y}\mathcal{P}(y)}\ \text{V}$$

$$\dfrac{\dfrac{\dfrac{\mathcal{H}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\ \text{ctxt}}\ \text{Lemma}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\vdash f:X\to Y}\ \text{V} \quad \dfrac{\dfrac{\mathcal{H}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\ \text{ctxt}}\ \text{Lemma}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\vdash x:X}\ \text{V}}{\Gamma,\ g:\Pi_{y:Y}\mathcal{P}(y),\ f:X\to Y,\ x:X\vdash f(x):Y}\ \Pi\text{-elim}$$

The above formal construction shows that seeking complete formalism by human hands is indeed virtually impossible. Even the very fundamental construction has considerably lengthy derivation. We can think that proof assistants automatically apply some inference rules described in this chapter.

2.2.2 Dependent sum Σ

The **dependent sum** type constructor Σ is defined as the following rules and corresponding congruence rules:

$$\dfrac{\Gamma\vdash A\ \text{type} \qquad \Gamma,\ x:A\vdash B(x)\ \text{type}}{\Gamma\vdash \Sigma_{x:A}B(x)\ \text{type}}\ \Sigma\text{-form}$$

$$\dfrac{\Gamma\vdash A\ \text{type} \qquad \Gamma,\ x:A\vdash B(x)\ \text{type}}{\Gamma,\ x:A,\ y:B(x)\vdash \text{pair}(x,y):\Sigma_{x:A}B(x)}\ \Sigma\text{-intro}$$

$$\dfrac{\Gamma,\ z:\Sigma_{x:A}B(x)\vdash \mathcal{P}(z)\ \text{type} \qquad \Gamma,\ x:A,\ y:B(x)\vdash f(x,y):\mathcal{P}(\text{pair}(x,y))}{\Gamma,\ z:\Sigma_{x:A}B(x)\vdash \Sigma\text{elim}(f,z):\mathcal{P}(z)}\ \Sigma\text{-elim}$$

$$\dfrac{\Gamma,\ z:\Sigma_{x:A}B(x)\vdash \mathcal{P}(z)\ \text{type} \qquad \Gamma,\ x:A,\ y:B(x)\vdash f(x,y):\mathcal{P}(\text{pair}(x,y))}{\Gamma,\ x:A,\ y:B(x)\vdash \Sigma\text{elim}(f,\text{pair}(x,y))\equiv f(x,y):\mathcal{P}(\text{pair}(x,y))}\ \Sigma\text{-comp}$$

For the case where B is not dependent over A, we write $A\times B$ for $\Sigma_{x:A}B(x)$.

When we interpret types A and B as propositions, then the above rules suggest that we can regard the type $A\times B$ as $A\wedge B$, the logical conjunction of two propositions. When we interpret the type family B as a predicate over a set A, we can think Σ as the existential quantifier $\exists$. If we interpret each type as a set, Σ corresponds to the disjoint union. In homotopical interpretation, Σ corresponds to taking the total space of a fibration.

As in the case of Π, we define corresponding two Agda functions respectively taking the base type implicitly, and explicitly with a similar notation to our type theory.

```
record Σ {A : type ℓ} (𝒫 : A → type ℓ') : type (ℓ ⊔ ℓ') where
    constructor
        _,_
    field
        pr₁ : A
        pr₂ : 𝒫 pr₁
open Σ public
infixr 50 _,_


-Σ :  (A : type ℓ) (𝒫 : A → type ℓ') → type (ℓ ⊔ ℓ')
-Σ A 𝒫 = Σ 𝒫
syntax -Σ A (λ x → 𝒫x) = Σ x : A , 𝒫x



_×_ : type ℓ → type ℓ' → type (ℓ ⊔ ℓ')
A × B = Σ x : A , B
infixr 30 _×_
```

2.2.3 Binary disjoint sum +

The **disjoint sum** type constructor + is defined as the following rules and corresponding congruence rules:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A + B \text{ type}} \text{ +-form}$$

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma,\ x : A \vdash \text{inl}(x) : A + B} \text{ +-intro-l}$$

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma,\ y : B \vdash \text{inr}(y) : A + B} \text{ +-intro-r}$$

$$\frac{\Gamma,\ z : A + B \vdash \mathcal{P}(z) \text{ type} \qquad \Gamma, x : A \vdash l(x) : \mathcal{P}(\text{inl}(x)) \qquad \Gamma, y : B \vdash r(y) : \mathcal{P}(\text{inr}(y))}{\Gamma,\ z : A + B \vdash +\text{elim}(l, r, z) : \mathcal{P}(z)} \text{ +-elim}$$

$$\frac{\Gamma,\ z : A + B \vdash \mathcal{P}(z)\ \text{type} \qquad \Gamma, x : A \vdash l(x) : \mathcal{P}(\text{inl}(x)) \qquad \Gamma, y : B \vdash r(y) : \mathcal{P}(\text{inr}(y))}{\Gamma,\ x : A \vdash +\text{elim}(l, r, \text{inl}(x)) \equiv l(x) : \mathcal{P}(\text{inl}(x))}\ +\text{-comp-l}$$

$$\frac{\Gamma,\ z : A + B \vdash \mathcal{P}(z)\ \text{type} \qquad \Gamma, x : A \vdash l(x) : \mathcal{P}(\text{inl}(x)) \qquad \Gamma, y : B \vdash r(y) : \mathcal{P}(\text{inr}(y))}{\Gamma,\ y : B \vdash +\text{elim}(l, r, \text{inr}(y)) \equiv r(y) : \mathcal{P}(\text{inr}(y))}\ +\text{-comp-r}$$

We formulate this type constructor as the following code:

```
data _+_ (A : type ℓ) (B : type ℓ') : type (ℓ ⊔ ℓ') where
    inl : A → A + B
    inr : B → A + B
infixr 20 _+_
```

Note that + corresponds to the coproduct of two objects in a category.

2.2.4 Empty type $\mathbb{0}$

The **empty type** $\mathbb{0}$ takes the role of the logical false in our theory. It resides in the base universe $\mathcal{U}$ and becomes the initial object among types. The rules for the empty type are given as follows, accompanied by corresponding congruence rules:

$$\frac{\vdash \Gamma\ \text{ctxt}}{\Gamma \vdash \mathbb{0}\ \text{type}}\ \mathbb{0}\text{-form}$$

$$\frac{\Gamma,\ x : \mathbb{0} \vdash \mathcal{P}(x)\ \text{type}}{\Gamma, x : \mathbb{0} \vdash \mathbb{0}\text{elim}(x) : \mathcal{P}(x)}\ \mathbb{0}\text{-elim}$$

There are no introduction and computation rules. Note that if there is a term in $\mathbb{0}$ under a context Γ, it would explode the system under the context Γ, making all types under Γ be inhabited by $\mathbb{0}$-elim, meaning that any mathematical statement is true with assumption Γ. We call a type theory with the above empty type rules **inconsistent** if it is possible to construct a term in $\mathbb{0}$ from the empty context(i.e. it is a useless system). If not, we call the theory **consistent**.

The empty type is implemented in Agda by the code:

```
data 𝟘 : 𝒰 where
```

It has no constructor.

The $\mathbb{0}$-elimination rule is automatically applied in Agda:

```
𝟘elim : {𝒫 : 𝟘 → type ℓ} → (x : 𝟘) → 𝒫 x
𝟘elim ()
```

We define the negation ¬ of types:

```
¬_ : type ℓ → type ℓ
¬ X = X → 𝟘
```

2.2.5 Unit type $\mathbb{1}$

The **unit type** $\mathbb{1}$ is a type generated by a single term. It resides in the base universe and becomes the terminal object among types. The rules for the unit type are given as follows, accompanied by corresponding congruence rules:

$$\frac{\vdash \Gamma \text{ ctxt}}{\Gamma \vdash \mathbb{1} \text{ type}}\ \mathbb{1}\text{-form}$$

$$\frac{\vdash \Gamma \text{ ctxt}}{\Gamma \vdash \star : \mathbb{1}}\ \mathbb{1}\text{-intro}$$

$$\frac{\Gamma,\ x : \mathbb{1} \vdash \mathcal{P}(x) \text{ type} \qquad \Gamma \vdash c : \mathcal{P}(\star)}{\Gamma, x : \mathbb{1} \vdash \mathbb{1}\text{elim}(c, x) : \mathcal{P}(x)}\ \mathbb{1}\text{-elim}$$

$$\frac{\Gamma,\ x : \mathbb{1} \vdash \mathcal{P}(x) \text{ type} \qquad \Gamma \vdash c : \mathcal{P}(\star)}{\Gamma, x : \mathbb{1} \vdash \mathbb{1}\text{elim}(c, \star) \equiv c : \mathcal{P}(x)}\ \mathbb{1}\text{-comp}$$

We formulate the unit type in Agda as:

```
data 𝟙 : 𝒰 where
    ⋆ : 𝟙
```

With the constructor +, we can build a type "with only 2 elements":

```
𝟚 : 𝒰
𝟚 = 𝟙 + 𝟙
```

We will use this type to inductively define spheres in Chapter 4. It has two terms "inl ⋆" and "inr ⋆" and it's sufficient to give values over them when constructing a section over $\mathbb{2}$ by elimination rules of + and $\mathbb{1}$. We introduce new notations for these terms:

```
pattern ₁ = inl ⋆
pattern ₂ = inr ⋆
```

The pattern command goes beyond just appending judgemental equality to one's preferred symbol. It allows pattern matching on its left-hand side symbol.

For any given type A, we can construct the constant function from A to $\mathbb{1}$:

```
const⋆ : {A : type ℓ} → A → 𝟙
const⋆ x = ⋆
```

2.2.6 Inductive types

Our theory has a rule allowing declarations of certain 'initial algebras' called the **W-type** constructor. However, we won't need the general description of W-types. We rather give an example of a W-type, the type of natural numbers $\mathbb{N}$ residing in the base universe, generated by two constructors, each respectively has its arity $\mathbb{0}$ (corresponds to the constant 0) and $\mathbb{1}$ (corresponds to the endo-map suc: $\mathbb{N} \to \mathbb{N}$), which can be category-theoretically interpreted as the initial object among algebras of the endofunctor $X \mapsto 1+X$:

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

$$\frac{\vdash \Gamma \text{ ctxt}}{\Gamma \vdash 0 : \mathbb{N}}\ \mathbb{N}\text{-intro-0}$$

$$\frac{\vdash \Gamma \text{ ctxt}}{\Gamma \vdash \text{suc} : \mathbb{N} \to \mathbb{N}}\ \mathbb{N}\text{-intro-suc}$$

$$\frac{\Gamma,\ n : \mathbb{N} \vdash \mathcal{P}(n) \text{ type} \qquad \Gamma \vdash c_0 : \mathcal{P}(0) \qquad \Gamma,\ n : \mathbb{N} \vdash c_s : \mathcal{P}(n) \to \mathcal{P}(\text{suc}(n))}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{elim}(c_0, c_s, n) : \mathcal{P}(n)}\ \mathbb{N}\text{-elim}$$

$$\frac{\Gamma,\ n : \mathbb{N} \vdash \mathcal{P}(n) \text{ type} \qquad \Gamma \vdash c_0 : \mathcal{P}(0) \qquad \Gamma, n : \mathbb{N} \vdash c_s : \mathcal{P}(n) \to \mathcal{P}(\text{suc}(n))}{\Gamma \vdash \mathbb{N}\text{elim}(c_0, c_s, 0) \equiv c_0 : \mathcal{P}(0)}\ \mathbb{N}\text{-comp-0}$$

$$\frac{\Gamma,\ n : \mathbb{N} \vdash \mathcal{P}(n) \text{ type} \qquad \Gamma \vdash c_0 : \mathcal{P}(0) \qquad \Gamma, n : \mathbb{N} \vdash c_s : \mathcal{P}(n) \to \mathcal{P}(\text{suc}(n))}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{elim}(c_0, c_s, \text{suc}(n)) \equiv c_s(n, \mathbb{N}\text{elim}(c_0, c_s, n)) : \mathcal{P}(\text{suc}(n))}\ \mathbb{N}\text{-comp-suc}$$

The above rules can be implemented in Agda as:

```
data ℕ : 𝒰 where
  zero : ℕ
  suc : ℕ → ℕ
{-# BUILTIN NATURAL ℕ #-}
```

BUILTIN NATURAL command allows us to write the usual decimal presentation for terms of $\mathbb{N}$. For a general explanation of W-types, see [9] Chapter 5.

2.2.7 Identity types _=_

Note that rules producing new judgemental equalities defined so far come from the rule stating that judgemental equalities are equivalence relations or from congruence and computational rules for each type constructor. These rules are so restrictive to do our envisioned mathematics and we inductively define a new concept of identification much more flexible than judgemental equality. The below-declared type families are called **identity types**, **path types**, or **propositional equalities**.

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

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma,\ x : A \vdash \text{refl}_A(x) : x =_A x} \text{ =-intro}$$

$$\frac{\Gamma,\ x, y : A,\ p : x =_A y \vdash \mathcal{P}(x, y, p) \text{ type} \qquad \Gamma,\ x : A \vdash c(x) : \mathcal{P}(x, x, \text{refl}_A(x))}{\Gamma,\ x, y : A,\ p : x =_A y \vdash\!= \text{elim}(c, x, y, p) : \mathcal{P}(x, y, p)} \text{ =-elim}$$

$$\frac{\Gamma,\ x, y : A,\ p : x =_A y \vdash \mathcal{P}(x, y, p) \text{ type} \qquad \Gamma,\ x : A \vdash c(x) : \mathcal{P}(x, x, \text{refl}_A(x))}{\Gamma,\ x, y : A,\ p : x =_A y \vdash\!= \text{elim}(c, x, x, \text{refl}_A(x)) \equiv c(x) : \mathcal{P}(x, x, \text{refl}_A(x))} \text{ =-comp}$$

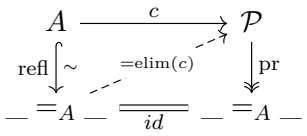
When we interpret the type A as a set, we expect the type $x =_A y$ to be merely a proposition on 'whether two elements x and y are equal or not', completely determined by whether there is a term of it or not. That is, there would be no distinct two terms in the identity type. Then the type family $\_ =_A \_ : A \to A \to \mathcal{U}$ can be interpreted as a predicate over the product set $A \times A$, where its value is 'true'(that is, there exists the term refl) on the diagonal and 'false' elsewhere. Hence $\mathcal{P}$ is a family of sets 'partially' indexed over $A \times A$, where it is well-defined only on the diagonal. With these interpretations the above rules all make sense.

Recall that in homotopical interpretation, a type family $x : B \vdash E(x)$ type is regarded as a fibration $E \xrightarrow{\text{pr}} B$. More naturally, without confining the type $x =_A y$ ought to be trivial, we can interpret the type A as a space, and the identity type $\_ =_A \_$ as the path object A^I with the factorization of the diagonal map:

$$A \xrightarrow[\text{refl}]{\sim} \_ =_A \_ \xrightarrow{\text{pr}} A \times A, \qquad A \xrightarrow{\Delta} A \times A$$

Now refl_A becomes the trivial cofibration assigning the constant path in the loop space $\Omega(A, x)$ for each point of A. The elimination and computation rules still make sense since

the required section and commutativeness are the solution to the lifting problem assured by the lifting axiom of the model category, depicted below (for the concept of model category, refer to [3]):



For a more detailed explanation of the above interpretation, see [1].

Note that $x =_A y$ is also a type, which allows to construct a new path type when given the two paths $p, q : x =_A y$. The type $p =_{x=_A y} q$ is interpreted as the space of all path-homotopies from p to q. Iterating this procedure will yield an ∞-groupoid structure on the given type A, as with the case of a fundamental ∞-groupoid on a given space. We call a term of a type of form $x =_A y$ as a **1-path** of A, $p =_{x=_A y} q$ as a **2-path** of A, and correspondingly a **n-path** of A.

The above type constructor is presented by the following Agda code:

```
data _=_ {A : type ℓ} : A → A → type ℓ where
    refl : (x : A) → x = x
infix 0 _=_
```

The character " = " is not the usual equality sign on the keyboard "=" and is not primitively supported by the Agda input method. One can append this symbol to the input method by following the instructions on this page [6].

1-groupoid structure on types

Now we examine how the groupoid structure for 1-paths can be constructed. We already have `refl`, expected to be the neutral elements for our groupoid. We define the operation as follows:

```
_•_ : {X : type ℓ} {x y z : X} → x = y → y = z → x = z
refl _ • q = q
infixl 30 _•_
```

For given 1-paths $p : x = y$ and $q : y = z$ in a type A, we have to construct their 'concatenated path' in $x = z$. By the $=$-elimination rule, it suffices to give a path for the case $p \equiv \text{refl}(x)$ with $x \equiv y$. For that case, we already have a path $q : x = z$ since $y \equiv x$,

hence define $\mathrm{refl}(x) \bullet q :\equiv q$. Now we get a general way to concatenate two given 1-paths p and q sharing suitable end-points, where the concatenated path $p \bullet q$ reduces to q when p is judgementally equal to $\mathrm{refl}(x)$.

On the above Agda code, we observe that the $=$-elimination rule is automatically handled inside of Agda. Such automatic application of elimination rules of inductive types in proof assistants is called **pattern matching**.

From the code above, we can observe another treatment of the underbar character "_" in Agda. We already saw that "_" can be used to define an infix operator. In the above code, it was differently used to implicitly write the argument x. When the constraints are sufficient to determine what argument should be filled, one can use "_" in that place instead of writing the argument explicitly. Agda will automatically infer the argument in that place. This allows us to write Agda code more neatly.

The inverse path operator can also be constructed by the $=$-elimination rule:

```
_⁻¹ : {A : type ℓ} {x y : A} → x = y → y = x
refl _ ⁻¹ = refl _
infix 40 _⁻¹
sym = _⁻¹
```

For convenience, we parallelly use the notation $p^{-1} \equiv: \mathtt{sym}(p)$.

Recall that the fundamental groupoid structure in algebraic topology was given by quotients up to homotopy, that is, by constructing the suitable homotopies between some paths. Similarly, groupoid rules for types don't hold strictly; that is, the related equations hold by propositional equalities rather than judgemental ones. (In this sense, we should say types are "weak" ∞-groupoids in principle). We construct corresponding 2-paths, that is, the homotopies between 1-paths, by pattern matching on the path type arguments as above:

```
•-refl-l : {A : type ℓ} {x y : A}
         → (p : x = y) → refl x • p = p
•-refl-l (refl _) = refl _

•-refl-r : {A : type ℓ} {x y : A}
         → (p : x = y ) → p • refl y = p
```

```
•-refl-r (refl _) = refl _

•-sym-l : {A : type ℓ} {x y : A}
        → (p : x = y ) → p ⁻¹ • p = refl y
•-sym-l (refl _) = refl _

•-sym-r : {A : type ℓ} {x y : A}
        → (p : x = y ) → p • p ⁻¹ = refl x
•-sym-r (refl _) = refl _

•-assoc : {A : type ℓ} {x y z w : A}
        → (p : x = y) (q : y = z) (r : z = w)
        → (p • q) • r = p • (q • r)
•-assoc (refl _) q r = refl _

sym²~id : {A : type ℓ} {x y : A}
        → (p : x = y) → sym (sym p) = p
sym²~id (refl _) = refl _

sym• : {A : type ℓ} {x y z : A}
     → (p : x = y) (q : y = z)
     → sym (p • q) = sym q • sym p
sym• (refl _) q = sym (•-refl-r (q ⁻¹))
```

Functions are functors between groupoids

Any function $f : X \to Y$ acts on paths in the domain type. We denote the action as ap, which stands for "apply." It preserves refl, the operation $\bullet$, and inverse up to homotopy, therefore we consider a function between types as a groupoid morphism up to homotopy. The constructions are trivially followed by pattern matching on path arguments:

```
ap : {X : type ℓ} {Y : type ℓ'} {x x' : X}
   → (f : X → Y) → (x = x') → f x = f x'
ap f (refl _) = refl _

ap-refl : {X Y : type ℓ} {x : X}
        → (f : X → Y) → ap f (refl x) = refl (f x)
ap-refl f = refl _

ap-• : {X : type ℓ} {Y : type ℓ'} {x y z : X}
     → (f : X → Y) (p : x = y) (q : y = z)
     → ap f (p • q) = ap f p • ap f q
ap-• f (refl _) q = refl _

ap-sym : {X : type ℓ} {Y : type ℓ'} {x y : X}
       → (f : X → Y) (p : x = y)
       → ap f (p ⁻¹) = (ap f p) ⁻¹
ap-sym f (refl _) = refl _
```

The identity function $\mathtt{id} : X \to X$ yields (up to homotopy) identity groupoid morphism, and the action of functions on arrows is compatible with function compositions.

```
ap-id : {X : type ℓ} {x y : X}
      → (p : x = y) → ap id p = p
ap-id (refl x) = refl _

ap-∘ : {X : type ℓ} {Y : type ℓ'} {Z : type ℓ''} {x y : X}
     → (g : Y → Z) (f : X → Y) (p : x = y)
     → ap (g ∘ f) p = (ap g (ap f p))
ap-∘ g f (refl x) = refl _
```

Here we define some gadgets for Agda code. These will be useful to formally deal with a chain of path concatenations in Agda.

```
_=⟨_⟩_ : {A : type ℓ} {y z : A} (x : A) → x = y → y = z → x = z
```

```
x =⟨ p ⟩ q = p • q
infixr 0 _=⟨_⟩_

_■ : {A : type ℓ} (x : A) → x = x
x ■ = refl x
infix 1 _■
```

Note that the operator _=⟨_⟩_ is declared right-associative and ■ has higher priority. For example, the expression "x =⟨p⟩ y =⟨q⟩ z =⟨r⟩ w ■" will denote the path "p•(q•(r•refl (w)))". This clever construction allows us to code the chain of propositional equalities as accustomed form. Below is one example, which is also a useful gadget for Agda coding.

```
ap₂ : {X : type ℓ} {Y : type ℓ'} {Z : type ℓ''} {x x' : X} {y y' : Y}
    → (F : X → Y → Z) → x = x' → y = y' → F x y = F x' y'
ap₂ {x = x} {x' = x'} {y = y} {y' = y'} F p q
  = F x y  =⟨ ap _ p ⟩
    F x' y =⟨ ap _ q ⟩
    F x' y' ■
```

In the above code, the first underbar is the function λ t → F t y in the type X → Z, where Agda has automatically inferred it.

The action of 1-paths on total spaces

As a path in a base space of a covering space has the natural action that sends points in the fiber at the path's starting point to the fiber at the path's target point, we can construct the same action in our type theory, called transport:

```
tr : {A : type ℓ} {x y : A}
   → (𝒫 : A → type ℓ') → x = y → 𝒫 x → 𝒫 y
tr 𝒫 (refl x) = id
```

This can be thought of as the type theory version of Leibniz's indiscernibility of identicals in classical logic. If x and y are equal propositionally, $\mathcal{P}\ x$ holds if and only if $\mathcal{P}\ y$ holds.

It is tempting to make the type family argument $\mathcal{P}$ implicit and write the action of a base path on its fibration with a right action notation as:

```
-tr : {A : type ℓ} {x y : A} {𝒫 : A → type ℓ'}
    → x = y → 𝒫 x → 𝒫 y
-tr (refl x) = id
syntax -tr p z = z ↶ p
```

However, in most cases, Agda can not infer what the type family is. Hence we won't use the above notation in our Agda files.

Transport is well-behaved with respect to $\bullet$ and $\circ$ up to homotopy as expected:

```
tr-• : {X : type ℓ} {x y z : X}
     → (𝒫 : X → type ℓ') (p : x = y) (q : y = z)
     → tr 𝒫 (p • q) = tr 𝒫 q ∘ tr 𝒫 p
tr-• 𝒫 (refl _) q = refl _

tr-∘ : {X : type ℓ} {Y : type ℓ'} {x x' : X}
     → (𝒫 : Y → type ℓ'') (f : X → Y) (p : x = x')
     → tr (𝒫 ∘ f) p = tr 𝒫 (ap f p)
tr-∘ 𝒫 f (refl _) = refl _
```

For a constant type family $\mathcal{P} :\equiv \lambda(x : X).Y$, the fibration will be trivial and regarded as the product $X \times Y$, and will have no interesting torsion:

```
tr-const : {X : type ℓ} {Y : type ℓ'} {x x' : X}
         → (p : x = x') (y : Y)
         → tr (λ - → Y) p y = y
tr-const (refl _) y = refl y
```

Let a type family $\mathcal{P}$ over A with $a, b : A$ and $x : \mathcal{P}(a)$, $y : \mathcal{P}(b)$ be given. $\mathcal{P}(a)$ and $\mathcal{P}(b)$ might not be the same type, hence we can't think of path type $x = y$ generally. The right notion will be the path type in the total space $(a, x) =_{\Sigma\mathcal{P}} (b, y)$. However, we have another way to relate x and y using transport; For any path $p : a =_A b$, $\mathrm{tr}^{\mathcal{P}}_p x$ resides in the type $\mathcal{P}(b)$ so we can construct the path type $\mathrm{tr}^{\mathcal{P}}_p x =_{\mathcal{P}(b)} y$. Indeed, we can establish an equivalence between $(a, x) =_{\Sigma\mathcal{P}} (b, y)$ and $\Sigma_{p:a=_A b}(x \curvearrowleft p) =_{\mathcal{P}(b)} y$ (see [5]

9.3 Characterizing the identity types of Σ-types). Through this isomorphism, we identify each path in $\mathrm{tr}_p^{\mathcal{P}} x =_{\mathcal{P}(b)} y$ with a path in the total space $(a, x) =_{\Sigma\mathcal{P}} (b, y)$. We introduce a new notation representing this identification:

```
pathover : {A : type ℓ} (𝒫 : A → type ℓ') {a b : A}
         → (x : 𝒫 a) (y : 𝒫 b) (p : a = b) → type ℓ'
pathover 𝒫 x y p = tr 𝒫 p x = y

syntax pathover 𝒫 x y p = x =↑ y [ p ]over 𝒫
```

There are some applications of transport which will be used later:

```
tr-fibmap : {X : type ℓ} {x y : X}
          → (𝒫 : X → type ℓ') (𝒬 : X → type ℓ'') (p : x = y) (f : 𝒫 x → 𝒬 x)
          → tr (λ - → 𝒫 - → 𝒬 -) p f = (tr 𝒬 p) ∘ f ∘ (tr 𝒫 (p ⁻¹))
tr-fibmap 𝒫 𝒬 (refl _) f = refl f

tr-path-lfix : {X : type ℓ} {x y : X}
             → (s : X) (p : x = y)
             → (r : s = x) → tr (λ - → s = -) p r = r • p
tr-path-lfix s (refl _) r = sym (•-refl-r r)

tr-path-rfix : {X : type ℓ} {x y : X}
             → (t : X) (p : x = y)
             → (r : x = t) → tr (λ - → - = t) p r = (p ⁻¹) • r
tr-path-rfix t (refl _) r = refl r

tr-path-btwmaps : {X : type ℓ} {Y : type ℓ'} {x y : X}
                → (f g : X → Y) (p : x = y) (γ : f x = g x)
                → tr (λ - → f - = g -) p γ = (ap f p) ⁻¹ • γ • (ap g p)
tr-path-btwmaps f g (refl _) γ = sym (•-refl-r γ)
```

where fibmap stands for "fiberwise map", lfix for "left fixed", rfix for "right fixed", and btwmaps for "between maps".

Finally, we observe the action of a dependent function on paths in a base type. For a given path $p : x = y$ in a type A with a type family $\mathcal{P}$ over A and a section $f : \Pi\mathcal{P}$, it is natural to expect there will be an induced path in total space $(x, f(x)) =_{\Sigma\mathcal{P}} (y, f(x))$. Recall that this type was identified with the type $\Sigma_{q:x=y}(\mathrm{tr}_q^{\mathcal{P}} f(x)) = f(y)$.

```
apd : {X : type ℓ} {𝒫 : X → type ℓ'} {x y : X}
    → (f : Π 𝒫) (p : x = y) → tr 𝒫 p (f x) = f y
apd f (refl _) = refl (f _)
```

We regard the above path in $\mathcal{P}(y)$ as an image of the path p under f in the total space of the fibration $\mathcal{P}$.

When the type family $\mathcal{P}$ is a constant family, `apd` can be reduced to `ap` through `tr-const`:

```
apd-const : {X : type ℓ} {Y : type ℓ'} {x y : X}
          → (f : X → Y) (p : x = y)
          → apd f p = (tr-const p (f x)) • (ap f p)
apd-const f (refl _) = refl _
```

Chapter 3

Basic Constructions

In this chapter, we construct fundamental concepts to analyze structures emerging from our type theory, namely, homotopy between functions, equivalence of types, and h-level.

3.1 Homotopy between functions

For given two sections $f, g : \Pi_{x:X}\mathcal{P}(x)$, a term H of the type $\Pi_{x:X} f(x) = g(x)$ is interpreted as a universal(continuous) way of giving a path from $f(x)$ to $g(x)$. (Recall that any term of Π-type is regarded as a continuous section in homotopy interpretation). Hence, it can be considered a function $H : I \times X \to \mathcal{P}$. We call such H a **homotopy from** f **to** g and formalize in Agda as:

```
_∼_ : {X : type ℓ} {𝒫 : X → type ℓ'}
    → Π 𝒫 → Π 𝒫 → type (ℓ ⊔ ℓ')
_∼_ {X = X} f g = (x : X) → f x = g x
infix 0 _∼_
```

We can easily show that $\sim$ is an equivalence relation:

```
ht-refl : {A : type ℓ} {𝒫 : A → type ℓ'}
        → (f : Π 𝒫) → f ∼ f
ht-refl f = λ - → refl _


_ʰⁱ :{A : type ℓ} {𝒫 : A → type ℓ'} {f g : Π 𝒫}
    → f ∼ g → g ∼ f
```

```
H ^hi = λ x → sym (H x)

ht-sym = _^hi
```

```
_•h_ : {A : type ℓ} {P : A → type ℓ'} {f g h : Π P}
     → f ∼ g → g ∼ h → f ∼ h
H •h K = λ x → H x • K x

ht• = _•h_

infixl 20 _•h_
```

Recall that a function between types becomes (up to homotopy) a groupoid morphism. A homotopy H from f to g becomes (up to homotopy) a natural isomorphism. That is, for any path $p : x = y$, we have a 2-path filling the below square:

$$\begin{array}{ccc} f(x) & \xrightarrow{ap_f p} & f(y) \\ {\scriptstyle H(x)}\downarrow & \swarrow\!\!\swarrow & \downarrow{\scriptstyle H(y)} \\ g(x) & \xrightarrow[ap_g p]{} & g(y) \end{array}$$

The construction is followed by =-elimination:

```
ht-nat : {A : type ℓ} {B : type ℓ'} {f g : A → B} {x y : A}
       → (H : f ∼ g) (p : x = y) → (H x) • (ap g p) = (ap f p) • (H y)
ht-nat {x = x} H (refl x) = •-refl-r (H x)
```

It is useful to stipulate the bottom side and upper side of the above diagram:

```
ht-nat-u : {A : type ℓ} {B : type ℓ'} {f g : A → B} {x y : A}
       → (H : f ∼ g) (p : x = y) → (H x) • (ap g p) • (H y) ^ -1 = ap f p
ht-nat-u {f = f} {g = g} {x = x} H (refl x)
  = H x • refl (g x) • H x ^ -1 =⟨ ap (λ - → - • H x ^ -1) (•-refl-r (H x)) ⟩
    H x • H x ^ -1 =⟨ •-sym-r (H x) ⟩
    refl _ ■
```

```
ht-nat-d : {A : type ℓ} {B : type ℓ'} {f g : A → B} {x y : A}
       → (H : f ∼ g) (p : x = y) → (H x) ^ -1 • (ap f p) • (H y) = ap g p
ht-nat-d {x = x} H (refl x)
  = ap (λ - → - • H x) (•-refl-r (H x ^ -1)) • (•-sym-l (H x))
```

We define new operations called left and right whiskering, corresponding to composites of a functor and a natural transformation:

```
_∘l_ : {A : type ℓ} {B : type ℓ'} {C : type ℓ''} {f g : A → B}
     → (h : B → C) → f ∼ g → h ∘ f ∼ h ∘ g
h ∘l H = λ - → ap h (H -)
infix 70 _∘l_

_∘r_ : {A : type ℓ} {B : type ℓ'} {C : type ℓ''} {g h : B → C}
     → g ∼ h → (f : A → B) → g ∘ f ∼ h ∘ f
H ∘r f = λ - → H (f -)
infix 70 _∘r_
```

Finally, we define syntax sugars similar to the already defined one for paths:

```
_∼⟨_⟩_ : {A : type ℓ} {P : A → type ℓ'} {g h : Π P}
       → (f : Π P) → f ∼ g → g ∼ h → f ∼ h
f ∼⟨ H ⟩ K = H •h K
infixr 0 _∼⟨_⟩_

_□ : {A : type ℓ} {P : A → type ℓ'} (f : Π P) → f ∼ f
f □ = ht-refl f
infix 1 _□
```

3.2 Equivalence

Let a function $f : A \to B$ be given. A function $g : B \to A$ is called a **section of** f if there is a homotopy $H : f \circ g \sim \mathrm{id}$, and a **retraction of** f if there is a homotopy $K : g \circ f \sim \mathrm{id}$.

```
has-sec : {A : type ℓ} {B : type ℓ'} (f : A → B) → type (ℓ ⊔ ℓ')
has-sec f = Σ (λ g → f ∘ g ∼ id)
```

```
has-ret : {A : type ℓ} {B : type ℓ'} (f : A → B) → type (ℓ ⊔ ℓ')
has-ret f = Σ (λ g → g ∘ f ∼ id)
```

When there are functions $s : A \to B$, $r : B \to A$ and a homotopy witnessing that r is a retraction of s, that is, a homotopy $r \circ s \sim \mathrm{id}$, we say the type A **is a retract of** B and denote it by $A \lhd B$. We can formulate this concept using record command, rather than as an iterated Σ type. We can attach suitable labels through the record command so that we don't have to remember the order of data:

```
record _◁_ (A : type ℓ) (B : type ℓ') : type (ℓ ⊔ ℓ') where
  constructor ◁pf
  field
    ret : B → A
    retpf : has-sec (ret)
infix 0 _◁_
```

A retraction preserves the h-level, which is the concept that measures the dimension in which the higher structure of a given type becomes trivialized. This will be proved in the next section.

Now we define invertibility and equivalence. For a function $f : A \to B$, we say f is **invertible** when there is a single function $g : B \to A$ with two homotopies each witnessing $f \circ g \sim \mathrm{id}$ and $g \circ f \sim \mathrm{id}$.

```
record ivtbl {A : type ℓ} {B : type ℓ'} (f : A → B) : type (ℓ ⊔ ℓ') where
  constructor Ivtbl
  field
    inv : B → A
    inv-s : f ∘ inv ∼ id
    inv-r : inv ∘ f ∼ id
```

Two types A and B are **isomorphic** if there is an invertible function $f : A \to B$. We denote it by $A \cong B$.

```
record _≅_ (A : type ℓ) (B : type ℓ') : type (ℓ ⊔ ℓ') where
  constructor ≅pf
```

```
  field
    ivt : A → B
    ivtpf : ivtbl ivt
infix  0 _≅_
```

We similarly define equivalence. A function $f : A \to B$ is **equivalence** if f has both section and retraction. That is, there are two functions $r, s : B \to A$ and homotopies $H : f \circ s \sim \mathrm{id}$ and $K : r \circ f \sim \mathrm{id}$.

```
record  equiv  {A : type ℓ} {B : type ℓ'} (f : A → B) : type (ℓ ⊔ ℓ') where
  constructor Equiv
  field
    sec : B → A
    sec-h : f ∘ sec ∼ id
    ret : B → A
    ret-h : ret ∘ f ∼ id
```

Two types A and B are **equivalent** if there is an equivalence $f : A \to B$. We denote it by $A \simeq B$.

```
record  _≃_  (A : type ℓ) (B : type ℓ') : type (ℓ ⊔ ℓ') where
  constructor ≃pf
  field
    eqv : A → B
    eqvpf : equiv eqv
infix  0 _≃_
```

Equivalence and invertibility are logically equal. One direction is obvious; invertibility implies equivalence:

```
ivtbl-equiv :  {A : type ℓ} {B : type ℓ'} {f : A → B}
            → (I : ivtbl f) → equiv f
ivtbl-equiv {f} (Ivtbl inv inv-s inv-r) = Equiv inv inv-s inv inv-r
```

Conversely, an equivalence datum can be refined to an invertibility proof. First, we need to show that for equivalence data of a function f, its section and retraction are homotopic. Let the data is given as (s, S, r, R) with $S : f \circ s \sim \mathrm{id}$ and $R : r \circ f \sim \mathrm{id}$.

Whisker R by s at right to get a homotopy of $r \circ f \circ s \sim s$. Whisker S by r at left to get a homotopy of $r \circ f \circ s \sim r$. Concatenate them to get $s \sim r$.

```
equiv-sec∼ret :  {A : type ℓ} {B : type ℓ'} {f : A → B}
              → (E : equiv f) →  equiv.sec(E) ∼ equiv.ret(E)
equiv-sec∼ret {f} (Equiv s S r R) = ((R ∘r s) ^hi)  •h  (r ∘l S)
```

Using the above homotopy, we can constitute a term of `ivtbl` f with its inverse as s:

```
equiv-ivtbl : {A : type ℓ} {B : type ℓ'} {f : A → B}
            → (E : equiv f) → ivtbl f
equiv-ivtbl {f = f} (Equiv s S r R) = Ivtbl s S ((H ∘r f) •h R)
  where
  H = equiv-sec∼ret (Equiv s S r R)
```

Of course, the constructed inverse pair (f, s) shows that s is also an equivalence:

```
equiv-inv : {A : type ℓ} {B : type ℓ'} {f : A → B}
          → (E : equiv f) → (B → A)
equiv-inv E = ivtbl.inv (equiv-ivtbl E)


equiv-inv-equiv : {A : type ℓ} {B : type ℓ'} {f : A → B}
                → (E : equiv f) → equiv (equiv-inv E)
equiv-inv-equiv {f = f} E = ivtbl-equiv (Ivtbl f H (equiv.sec-h (E)))
  where
  H = equiv.sec(E) ∘ f   ∼⟨ equiv-sec∼ret(E) ∘r f ⟩
      equiv.ret(E) ∘ f   ∼⟨ equiv.ret-h E ⟩
      id  □
```

We can use the above logical equivalence to convert between $\cong$ and $\simeq$:

```
≅-≃ : {A : type ℓ} {B : type ℓ'} → A ≅ B → A ≃ B
≅-≃ {A} {B} (≅pf ivt (Ivtbl inv inv-s inv-r))
  = ≃pf ivt (Equiv inv inv-s inv inv-r)


≃-≅ : {A : type ℓ} {B : type ℓ'} → A ≃ B → A ≅ B
≃-≅ {A} {B} (≃pf eqv  eqvpf) = ≅pf eqv (equiv-ivtbl eqvpf)
```

It is useful to stipulate the extraction of component function and inverse function of a given equivalence datum:

```
≃-eqv : {A : type ℓ} {B : type ℓ'} → A ≃ B → (A → B)
≃-eqv E = _≃_.eqv(E)

≃-inv :  {A : type ℓ} {B : type ℓ'} → A ≃ B → (B → A)
≃-inv E = ivtbl.inv (equiv-ivtbl (_≃_.eqvpf(E)))
```

Even though invertibility and equivalence seem similar and are logically equivalent, they have a crucial difference. For any function $f : A \to B$, the type `equiv` f is a proposition, meaning that any two terms of it are propositionally equal. However, one can construct a function f such that the type `ivtbl` f is not a proposition, having two terms where the path type between them is empty. See [9] 4.1.3. We think that the type of equivalence data has more residual space allowing the type to shrink to a point if it is inhabited.

Recall that a homotopy between functions is a natural isomorphism between groupoids up to homotopy. Therefore isomorphism and equivalence of types correspond to the notion of equivalence of categories if we regard types as 1-groupoids up to 2-paths. Now one may doubt whether these concepts preserve higher structures above dimension 1 since they were constructed only using 1-paths. Surprisingly, the answer is affirmative. If $f : A \to B$ is an equivalence, then for any $x, y : A$, the apply function `ap` $_f : x = y \to f(x) = f(y)$ becomes an equivalence. This implies that an equivalence between two types induces an equivalence between higher structures with suitable boundaries. The proof is in [9] Theorem 2.11.1. The idea is to use the homotopy square several times.

3.3 h-level

In this section, we define h-level and prove that $\mathbb{Z}$ is a set, that is, $\mathbb{Z}$ has h-level 2. h-level is the concept that measures the dimension in which the higher structure of a given type becomes trivialized. It is inductively defined and starts with the contractibleness.

Contractibleness of a type A consists of a term c of A, which will be called the center

of contraction, and a uniform assigning of paths; for each $x : A$, a path in $c = x$. A term of this type is interpreted as a homotopy from the constant function (at c) to the identity function id_A, appealing to the usual definition of contractible space in topology.

```
contr : type ℓ → type ℓ
contr A = Σ c : A , (Π x : A , (c = x))


center : {A : type ℓ} → contr A → A
center C = pr₁ C
```

We can easily show that the unit type is contractible:

```
𝟙-contr : contr 𝟙
𝟙-contr = ⋆ , λ {⋆ → refl ⋆}
```

For a contraction, we should construct the section in $\Pi_{x:\mathbb{1}} \star = x$. By $\mathbb{1}$-elim, it suffices to give a path when $x \equiv \star$, we give it as refl $\star$. The above code shows how we can use inline pattern matching; by enclosing the bounding area of λ by braces { }, we can use pattern matching inside braces. Of course, one can use the where clause to construct the section rather than using inline pattern matching.

If a type is contractible, any path type above it is also contractible:

```
contr-closed-above : {A : type ℓ} → (contr A) → ((x y : A) → contr (x = y))
contr-closed-above (c , H) x y
  = ((H x) ⁻¹ • H y) , λ { (refl x) → •-sym-l (H x)}
```

Let `(c , H) : contr A`. Then we set the center as the path drop by `c` using contraction `H`. For the contraction, we should construct the section $\Pi_{p:x=y}$`(H x) ⁻¹ • H y = p`. Note that `x` and `y` are free, that is, they are variables in our context. Hence we can use pattern matching on p, represented by the inline pattern matching in the above code. It suffices to give a path when `y` $\equiv$ `x`, and `p` $\equiv$ `refl x`. We give `•-sym-l (H x) : (H x) ⁻¹ • H x = refl x`.

Now we define h-level as follows:

```
_has-hlv_ : type ℓ → ℕ → type ℓ
A has-hlv 0 = contr A
A has-hlv suc n = (x y : A) → (x = y) has-hlv n
```

In fact, it is customary to start from -2 for h-level. We define as above to use the built-in decimal notation of natural numbers in Agda.

We call a type of h-level 1 a proposition. A type is a proposition if and only if any path type over it is contractible. Hence a proposition is empty or contractible.

```
prop : type ℓ → type ℓ
prop A = A has-hlv 1
```

The empty type is trivially a proposition; the variable in context completes the proof by $\mathbb{0}$-elim.

```
𝟘-prop : prop 𝟘
𝟘-prop = λ x ()
```

We call a type of h-level 2 a set. A type is a set if and only if any 2-path space is contractible.

```
set : type ℓ → type ℓ
set A = A has-hlv 2
```

This condition is equivalent to the statement that there is a uniform way of giving a 2-path over any parallel two 1-paths. We denote the latter condition as `set'`:

```
set' : type ℓ → type ℓ
set' A = (x y : A) → (p q : x = y) → p = q
```

One direction is direct. We just take the center from `(sA x y p q) : contr (p = q)`:

```
set→set' : {A : type ℓ} → set A → set' A
set→set' sA x y p q = pr₁ (sA x y p q)
```

For the converse, note that the given term `(sA' : set' A)` gives the contractibleness proof for the type `x = y` using the variable `p` in the context. This completes the proof with the above derivation `contr-closed-above`:

```
set'→set : {A : type ℓ} → set' A → set A
set'→set sA' x y p q = contr-closed-above  below-contr p q
  where
  below-contr : contr (x = y)
  below-contr = p , (sA' x y p)
```

We will need the derivation `set→ set'` in the Chapter 5.

Now we prove some properties of h-level. First, the h-level is closed above; if a type is of h-level k, then it also is of h-level $k+1$.

```
hlv-closed-above : (n : ℕ) {A : type ℓ} → (A has-hlv n) → (A has-hlv (suc n))
hlv-closed-above 0 H = contr-closed-above H
hlv-closed-above  (suc n) H x y = IH {T = (x = y)} (H x y)
  where
  IH : {T : type ℓ} → (T has-hlv n) → (T has-hlv (suc n))
  IH = hlv-closed-above n
```

We use the pattern matching on the argument from $\mathbb{N}$ for the first time. We can use the `n`-th term when constructing the `(suc n)`-th term. We denote that by `IH`, standing for induction hypothesis. For `(suc n)`-th term, we have to construct the term of `(x=y has-hlv (suc n))`. By the `IH`, it suffices to give a term for `(x = y has-hlv n)`, which can be given as `(H x y)`.

The above property directly proves that $\mathbb{0}$ and $\mathbb{1}$ are set:

```
𝟙-prop : prop 𝟙
𝟙-prop = hlv-closed-above 0 𝟙-contr

𝟙-is-set : set 𝟙
𝟙-is-set = hlv-closed-above 1 𝟙-prop

𝟘-is-set : set 𝟘
𝟘-is-set = hlv-closed-above 1 𝟘-prop
```

Now we show that retraction preserves h-level. For the 0-level case, we should show that a retract of a contractible type is contractible:

```
contr-closed-ret : {A : type ℓ} {B : type ℓ'} → A ◁ B → contr B → contr A
contr-closed-ret {A = A} (◁pf r (s , H)) (b , C) = r b , K
  where
  K : (x : A) → r b = x
  K x = r b =⟨ ap r (C (s x)) ⟩
```

```
      r (s x) =⟨ H x ⟩
      x ■
```

We also need that a retraction induces a retraction of path types:

```
◁-closed-above : {A : type ℓ} {B : type ℓ'}
              → ((◁pf r (s , H)) : A ◁ B) → (x y : A) → (x = y) ◁ (s x = s y)
◁-closed-above (◁pf r (s , H)) x y =
  ◁pf ((λ - → (H x) ⁻¹ • - • (H y)) ∘ ap r)
      ((ap s) , K)
    where
    K : (p : x = y) → H x ⁻¹ • ap r (ap s p) • H y = p
    K (refl x) = (ap (λ - → - • H x) (•-refl-r (H x ⁻¹))) • (•-sym-l (H x))
```

The above homotopy `K` is constructed through the pattern matching on path argument.

We now can prove that h-level is stable under retraction:

```
hlv-closed-ret : (n : ℕ) {A : type ℓ} {B : type ℓ'}
              → A ◁ B → (B has-hlv n) → (A has-hlv n)
hlv-closed-ret 0 R hB = contr-closed-ret R hB
hlv-closed-ret (suc n) {A = A} {B = B} R hB  x y
  = IH (◁-closed-above R x y) (hB (s x) (s y))
  where
  IH = hlv-closed-ret n
  s : A → B
  s = pr₁ (_◁_.retpf R)
```

We combined facts that `(x = y)` is a retract of `(s x = s y)` and `(hB (s x) (s y))` gives the proof of `(s x = s y)` has h-level `n`.

We have defined the general properties of h-level. Our next goal is to define the type of integers and prove that it is a set. We define $\mathbb{Z}$ as an inductive type and construct the successor and predecessor operations.

```
data ℤ : 𝒰 where
 pos : ℕ → ℤ
 negsuc : ℕ → ℤ
```

```
succℤ : ℤ → ℤ
succℤ (pos x) = pos (suc x)
succℤ (negsuc 0) = pos 0
succℤ (negsuc (suc x)) = negsuc x

predℤ : ℤ → ℤ
predℤ (pos 0) = negsuc 0
predℤ (pos (suc x)) = pos x
predℤ (negsuc x) = negsuc (suc x)
```

The successor and predecessor constitute isomorphism:

```
succ-sec : predℤ ∘ succℤ ∼ id
succ-sec (pos x) = refl _
succ-sec (negsuc 0) = refl _
succ-sec (negsuc (suc x)) = refl _

succ-ret : succℤ ∘ predℤ ∼ id
succ-ret (pos 0) = refl _
succ-ret (pos (suc x)) = refl _
succ-ret (negsuc x) = refl _

succ-≅ : ℤ ≅ ℤ
succ-≅ = ≅pf succℤ (Ivtbl predℤ succ-ret succ-sec)

succ-≃ : ℤ ≃ ℤ
succ-≃ = ≅-≃ succ-≅
```

Of course, we could have defined the integer as $\mathbb{N} + \mathbb{N}$ with corresponding operations on it. Indeed, we can establish the following equivalence:

```
ℤ→ : ℤ → ℕ + ℕ
ℤ→ (pos x) = inr x
ℤ→ (negsuc x) = inl x
```

```
ℤ← : ℕ + ℕ → ℤ
ℤ← (inl x) = negsuc x
ℤ← (inr x) = pos x

ℤ≅ℕ+ℕ : ℤ ≅ ℕ + ℕ
ℤ≅ℕ+ℕ = ≅pf ℤ→ (Ivtbl ℤ← H K)
  where
  H : (x : ℕ + ℕ) → ℤ→ (ℤ← x) = x
  H (inl x) = refl (inl x)
  H (inr x) = refl (inr x)
  K : (x : ℤ) → ℤ← (ℤ→ x) = x
  K (pos x) = refl (pos x)
  K (negsuc x) = refl (negsuc x)
```

By the above equivalence, to prove $\mathbb{Z}$ is a set, it suffices to prove that $\mathbb{N}$ is a set and sets are closed under $+$. We can prove that $\mathbb{N}$ is a set by defining an auxiliary 'binary predicate' over $\mathbb{N}$. We define a binary type family $\mathbb{N}\mathtt{eq} : \mathbb{N} \to \mathbb{N} \to \mathcal{U}$ by pattern matching, where each `(ℕeq m n)` is a proposition. We establish an equivalence between `(ℕeq m n)` and `(m = n)`. Then we are done since it suffices to prove that every path type in $\mathbb{N}$ is a proposition.

```
ℕeq : ℕ → ℕ → 𝒰
ℕeq 0 0 = 𝟙
ℕeq 0 (suc n) = 𝟘
ℕeq (suc m) 0 = 𝟘
ℕeq (suc m) (suc n) = ℕeq m n

ℕeq-prop : (m n : ℕ) → prop (ℕeq m n)
ℕeq-prop zero zero = 𝟙-prop
ℕeq-prop zero (suc n) = 𝟘-prop
ℕeq-prop (suc m) zero = 𝟘-prop
ℕeq-prop (suc m) (suc n) = ℕeq-prop m n
```

```
ℕeq← : (m n : ℕ) → m = n → ℕeq m n
ℕeq← 0 0 (refl _) = ⋆
ℕeq← (suc m) (suc m) (refl _) = ℕeq← m m (refl m)

ℕeq→ : (m n : ℕ) → ℕeq m n → m = n
ℕeq→ 0 0 ⋆ = refl 0
ℕeq→ 0 (suc n) = λ ()
ℕeq→ (suc m) 0 = λ ()
ℕeq→ (suc m) (suc n) = (ap suc) ∘ IH
  where
  IH : ℕeq m n → m = n
  IH = ℕeq→ m n

ℕeq←suc : (m n : ℕ) → (ℕeq← (suc m) (suc n)) ∘ (ap suc) ∼ ℕeq← m n
ℕeq←suc m m (refl m) = refl (ℕeq← m m (refl m))


ℕeq≅ : (m n : ℕ) → (m = n) ≅ ℕeq m n
ℕeq≅ = λ m n → ≅pf (ℕeq← m n) (Ivtbl (ℕeq→ m n) (S m n) (R m n))
  where
  S : (m n : ℕ) → (ℕeq← m n) ∘ (ℕeq→ m n) ∼ id
  S 0 0 ⋆ = refl ⋆
  S 0 (suc n) = λ ()
  S (suc m) 0 = λ ()
  S (suc m) (suc n) = ((ℕeq←suc m n) ∘ᵣ (ℕeq→ m n)) •ₕ (S m n)

  R : (m n : ℕ) → (ℕeq→ m n) ∘ (ℕeq← m n) ∼ id
  R 0 0 (refl 0) = refl (refl 0)
  R (suc m) (suc m) (refl (suc m)) = ap (ap suc) (R m m (refl m))
```

A useful tool extracting the retraction data from an isomorphism data:

```
≅-◁ : {A : type ℓ} {B : type ℓ'} → A ≅ B → A ◁ B
```

```
≅-◁ (≅pf ivt (Ivtbl inv inv-s inv-r)) = ◁pf inv (ivt , inv-r)
```

Now we prove that $\mathbb{N}$ is set.

```
ℕ-is-set : set ℕ
ℕ-is-set m n = hlv-closed-ret 1 RET (ℕeq-prop m n)
  where
  RET : (m = n) ◁ ℕeq m n
  RET = ≅-◁ (ℕeq≅ m n)
```

What remains is to prove that h-level is closed under the disjoint sum $+$. We first observe that the disjoint sum is really "disjoint". For any `A + B`, we can build a type family over the sum by pattern matching, namely, $\mathbb{1}$ on `A` and $\mathbb{0}$ on `B`. Now by transport, any path from `(inl a)` to `(inr b)` will produce a term of $\mathbb{0}$.

```
+-disjoint : {A : type ℓ} {B : type ℓ'}
           → (a : A) (b : B) → ¬ (inl a = inr b)
+-disjoint {A = A} {B = B} a b p = tr 𝒫 p ⋆
  where
  𝒫 : A + B → 𝒰
  𝒫 (inl x) = 𝟙
  𝒫 (inr y) = 𝟘
```

Now we may try to imitate the proof of `ℕ-is-set`. Construct a suitable binary type family over `A + B`, and establish a fiberwise isomorphism. However, without the cumulativity of universes, we can't construct a suitable type family. For example, the following Agda code won't type check

```
+eq : {A : type ℓ} {B : type ℓ'} → (x y : A + B) → type (ℓ ⊔ ℓ')
+eq (inl a) (inl a') = a = a'
+eq (inl a) (inr b) = 𝟘
+eq (inr b) (inl a) = 𝟘
+eq (inr b) (inr b') = b = b'
```

since each type on the right-hand side of each line is of universe level `ℓ`, `ℓ0`, `ℓ0`, and `ℓ'`. Without the cumulativity, they won't reside in `type (ℓ ⊔ ℓ')`. To formulate the cumulativity in Agda, we have to append a type constructor generating the copy of each

type from a lower-level universe to a higher-level universe. We will not conduct this work but take a detour.

We still want to construct an equivalence between `(inl a = inl a')` and `(a = a')` for every `a, a' : A`. To define an adequate function from left to right, we need a function `A + B → A`. Note that we have a term of `A` in our context. This allows us to construct such function which we might call pointed projection:

```
+-pointed-pr-l : {A : type ℓ} {B : type ℓ'} → A → A + B → A
+-pointed-pr-l a₀ (inl a) = a
+-pointed-pr-l a₀ (inr b) = a₀
```

From the above projection, we can construct a function between path types by path induction:

```
apinl^i : {A : type ℓ} {B : type ℓ'}
    → (a : A) → (x : A + B) → (inl a = x) → (a = (+-pointed-pr-l a) x)
apinl^i {A = A} {B = B} a (inl a) (refl (inl a)) = refl a
```

`(apinl^i a (inl a'))` gives a function `(inl a = inl a') → (a = a')`. We expect this function to be an inverse of `(ap inl)`. We can easily construct a homotopy `S` witnessing that `(ap inl)` is a section by path induction since `(apinl^i a (inl a') (refl (inl a))` reduces to `refl a`.

For a homotopy `(ap inl) ∘ (apinl^i a (inl a')) ∼ id`, note that we can't use path induction since the path argument has the non-free endpoint `(inl a')`. Hence we make the endpoint free and consider a suitable section over the context `(x : A + B), (q : inl a = x)`. Note that `(ap inl) ∘ (apinl^i a x) (q)` inhabits in the type `(inl a = inl (+-pointed-pr-l a x))`. If one can construct a path in `(x = inl (+-pointed-pr-l a x))`, where we denote it by `(aux-l a x q)`, we could try to construct a term of `(ap inl) ∘ (apinl^i a (inl a')) (q) = q • (aux-l a x q)`.

Note that we can construct such `(aux-l a x q)` in two different ways. One is to use pattern matching on `q`, which will have a reduction `(aux-l a (inl a) (refl (inl a))) ≡ refl (inl a)`. The other is to use pattern matching on `x`, which will have a reduction `(aux-l a (inl a') q) ≡ refl (inl a')`, for all `(a' : A)` and `q`. We observe that the latter construction has a much more strong reduction rule. If we take the latter construction, we can prove as follows:

```
aux-l : {A : type ℓ} {B : type ℓ'}
      → (a : A) (x : A + B) (q : inl a = x)
      → x = inl (+-pointed-pr-l a x)
aux-l a (inl a') q = refl (inl a')
aux-l a (inr b) q = 𝟘elim (+-disjoint a b q)

inl-emb :  {A : type ℓ} {B : type ℓ'}
        → (a a' : A) → (inl {A = A} {B = B} a = inl a') ≅ (a = a')
inl-emb {A = A} {B = B} a a'
  = ≅pf (apinl^i a (inl a'))
        (Ivtbl (ap inl)
               S
               λ q → (R (inl a') q) • (•-refl-r q))
  where
  S : apinl^i {A = A} {B = B} a (inl a') ∘ ap inl ∼ id
  S (refl .a) = refl (refl a)

  R : (x : A + B) (q : inl a = x)
    → ap (inl {A = A} {B = B}) (apinl^i a x q) = q • aux-l a x q
  R .(inl a) (refl .(inl a)) = refl _
```

In the code, `(R (inl a') q)` inhabits in `(ap inl) ∘ (apinl^i a x) (q) = q • refl (inl a')`

Notice that if we have used path induction rather than +elim, we could not find a way to construct a term of `(q • (aux-l a (inl a') q) = q)` for the path without free endpoints `q : inl a = inl a'`. This example shows that the method of construction matters significantly; our type system is proof-relevant.

We can dually do the same thing for inr:

```
+-pointed-pr-r : {A : type ℓ} {B : type ℓ'} → B → A + B → B
+-pointed-pr-r b₀ (inl a) = b₀
+-pointed-pr-r b₀ (inr b) = b

apinr^i : {A : type ℓ} {B : type ℓ'}
        → (b : B) → (x : A + B) → (inr b = x) → (b = (+-pointed-pr-r b) x)
```

```
apinr^i {A = A} {B = B} b (inr b) (refl (inr b)) = refl b


aux-r : {A : type ℓ} {B : type ℓ'}
      → (b : B) (x : A + B) (q : inr b = x) → x = inr (+-pointed-pr-r b x)
aux-r b (inr y) q = refl (inr y)


inr-emb :  {A : type ℓ} {B : type ℓ'}
        → (b b' : B) → (inr {A = A} {B = B} b = inr b') ≅ (b = b')
inr-emb {A = A} {B = B} b b'
 = ≅pf (apinr^i b (inr b')) (Ivtbl (ap inr) S λ q → R (inr b') q • •-refl-r q)
  where
  S : apinr^i {A = A} {B = B} b (inr b') ∘ ap inr ∼ id
  S (refl b) = refl (refl b)

  R : (x : A + B) (q : inr b = x)
    → ap (inr {A = A} {B = B}) (apinr^i b x q) = q • aux-r b x q
  R (inr b) (refl (inr b)) = refl _
```

We finally show that sets are closed under $+$, and $\mathbb{Z}$ is a set:

```
set-closed-+ : {A : type ℓ} {B : type ℓ'} → set A → set B → set (A + B)
set-closed-+ sA sB (inl a) (inl a') = hlv-closed-ret 1 (≅-◁ (inl-emb a a')) (sA a a')
set-closed-+ sA sB (inl a) (inr b) = λ x ()
set-closed-+ sA sB (inr b) (inl a) = λ x ()
set-closed-+ sA sB (inr b) (inr b') = hlv-closed-ret 1 (≅-◁ (inr-emb b b')) (sB b b')


ℤ-is-set : set ℤ
ℤ-is-set = hlv-closed-ret 2 (≅-◁ ℤ≅ℕ+ℕ) (set-closed-+ ℕ-is-set ℕ-is-set)
```

Chapter 4

Additional Rules in HoTT

We axiomatize function extensionality and univalence and declare some higher inductive types in this Chapter.

4.1 Function extensionality

Suppose we interpret our type theory in set theory. In that case, propositional equality is just a strict equality and a homotopy between two sections is the proof witnessing two sections are pointwise equal. Hence, introducing the rule witnessing $f \sim g \to f = g$ makes sense.

When we interpret type theory in classical homotopy theory, recall that a homotopy H between two sections $f, g : \Pi_{x:X}\mathcal{P}(x)$ is regarded as a homotopy $H : I \times X \to \mathcal{P}$ from f to g. If we assume that the interpretation is taken in a convenient category allowing exponential law, H will have a transpose $\tilde{H} : I \to \mathcal{P}^X$ yielding a path from f to g in the function space. Hence it is still plausible to assume that $f \sim g$ implies $f = g$.

We axiomatize this rule called **function extensionality**, appending more nontrivial paths in section spaces through homotopies. It is known that this is not provable from basic dependent type theory. We first construct a function $f = g \to f \sim g$:

```
happ : {X : type ℓ} {𝒫 : X → type ℓ'} {f g : Π 𝒫}
      → f = g → f ∼ g
happ p x = ap (λ - → - x) p
fext^i = happ
```

`happ` stands for homotopy apply. We also use the notation $\texttt{fext}^i$ for `happ`. We can

also define `happ` using the path induction but the above construction has a more general reduction rule. Now we declare that `happ` is equivalence:

```
postulate
  FEXT : {X : type ℓ} {𝒫 : X → type ℓ'} {f g : Π 𝒫}
       → equiv (happ {f = f} {g = g})
```

We give names for each component of FEXT for convenience:

```
FEXT-ivtbl : {X : type ℓ} {𝒫 : X → type ℓ'} {f g : Π 𝒫}
           → ivtbl (happ {f = f} {g = g})
FEXT-ivtbl = equiv-ivtbl (FEXT)

fext : {X : type ℓ} {𝒫 : X → type ℓ'} {f g : Π 𝒫}
     →  f ∼ g → f = g
fext = ivtbl.inv (FEXT-ivtbl)

fext-s : {X : type ℓ} {𝒫 : X → type ℓ'} {f g : Π 𝒫}
       → (happ {f = f} {g = g}) ∘ (fext {f = f} {g = g}) ∼ id
fext-s = ivtbl.inv-s (FEXT-ivtbl)

fext-r : {X : type ℓ} {𝒫 : X → type ℓ'} {f g : Π 𝒫}
       → (fext {f = f} {g = g}) ∘ (happ {f = f} {g = g}) ∼ id
fext-r {f} {g} = ivtbl.inv-r (FEXT-ivtbl)
```

We write `fext` for the inverse of `happ` $\equiv$ `fext`^{i}, `fext-s` for the homotopy witnessing `happ` $\circ$ `fext` $\sim$ `id`, and `fext-r` for the homotopy witnessing `fext` $\circ$ `happ` $\sim$ `id`.

In fact, function extensionality can be proven by using the univalence axiom. See [9] 4.9. We won't need a reduced form of `fext` through univalence, hence just axiomatized it.

4.2 Univalence axiom

As the function extensionality adds some paths in section spaces through homotopies, the univalence axiom is a rule adding some paths in a universe through equivalences. We assume the univalence only on the base universe $\mathcal{U} \equiv$ type $\ell 0$. It will be sufficient in

proving that the loop space of the circle is equivalent to $\mathbb{Z}$. We start from constructing a function $A =_{\text{type}\ \ell} B \to A \cong B$:

```
tr-id : {A B : type ℓ} → A = B → (A → B)
tr-id = tr id
```

In the code, `id` is the identity function `(type ℓ) →(type ℓ)`. Note that the type `A = B` is a path type of the universe type `type ℓ`, which itself resides in the universe `type (lsuc ℓ)`. Hence the section `tr-id` resides in the universe `type (lsuc ℓ)`. We can also construct a section by path induction, but the above derivation has a more convenient reduction rule. We expect `tr-id p`$^{-1}$ to be the inverse of `tr-id p`.

```
=-≅ : {A B : type ℓ} → A = B → A ≅ B
=-≅ p = (≅pf (tr-id p)
             (Ivtbl (tr-id (p ⁻¹)) H K))
  where
  H : tr-id p ∘ tr-id (p ⁻¹) ∼ id
  H = (tr (id) p ∘ tr (id) (p ⁻¹))   ∼⟨ happ (sym (tr-• (id) (sym p) p) ) ⟩
      tr (id) (p ⁻¹ • p)             ∼⟨ happ (ap (\ - → tr id -) (•-sym-l p)) ⟩
      id □

  K : tr (id) (p ⁻¹) ∘ tr (id) p ∼ id
  K = (tr (id) (p ⁻¹) ∘ tr (id) p) ∼⟨ happ (sym (tr-• (id) p (sym p)) ) ⟩
      tr (id) (p • p ⁻¹)     ∼⟨ happ (ap (\ - →  tr (id) -) (•-sym-r p)) ⟩
      id □

=-≃ : {A B : type ℓ} → A = B → A ≃ B
=-≃ = ≅-≃ ∘ =-≅
ua^i = =-≃
```

Now we postulate the univalence axiom, declaring that `=-≃` is an equivalence:

```
postulate
  UA : {A B : 𝒰} → equiv (=-≃ {A = A} {B = B})
```

One might expect that declaring `equiv (=-≅)` will result in the same system as ours.

However, such a modification makes the system inconsistent; that is, it will produce a term of $\mathbb{0}$. See [9] Exercise 4.6.

As in the case of `FEXT`, we give names for derivations of some projections of data from `UA`:

```
UA-ivtbl : {A B : 𝒰} → ivtbl (=-≃ {A = A} {B = B})
UA-ivtbl = equiv-ivtbl (UA)

ua : {A B : 𝒰} → A ≃ B → A = B
ua = ivtbl.inv (UA-ivtbl)

ua-s : {A B : 𝒰} → (ua^i {A = A} {B = B}) ∘ (ua {A = A} {B = B}) ∼ id
ua-s = ivtbl.inv-s (UA-ivtbl)

ua-r : {A B : 𝒰} → (ua {A = A} {B = B}) ∘ (ua^i {A = A} {B = B}) ∼ id
ua-r = ivtbl.inv-r (UA-ivtbl)
```

From the univalence, we have a path in the base universe for each type equivalence in the base universe. This will allow us to transport any construction on a type to its equivalent type through the path. Hence the univalence axiom provides the formalization of mathematical convention of identifying objects up to equivalence.

We need some properties related to the univalence axiom. Recall that for any path $p : X =_{\mathcal{U}} Y$ in the universe, the data `=-≃` (p) has the function `tr-id` p and its section and retraction `tr-id` p^{-1}. Hence we know that `(_≃_.eqv) ∘ ua^i` $\equiv$ `tr-id`. This gives the following:

```
tr-ua : {X Y : 𝒰} (E : X ≃ Y) → tr-id (ua E) = (_≃_.eqv E)
tr-ua = _≃_.eqv ∘ₗ (ua-s)
```

An equivalence data can easily be converted to the inverse direction:

```
≃sym : {X : type ℓ} {Y : type ℓ'} → X ≃ Y → Y ≃ X
≃sym (≃pf eqv eqvpf) = ≃pf (equiv-inv eqvpf) (equiv-inv-equiv eqvpf)
```

And is compatible with `=-≃`, which is directly proved by path induction:

```
=-≃-sym : {X Y : 𝒰} → (p : X = Y) → =-≃ (sym p) = ≃sym (=-≃ p)
=-≃-sym (refl _) = refl _
```

We prove that `ua` is compatible with `≃sym` and `sym`:

```
ua-sym : {X Y : 𝒰} → (E : X ≃ Y) → ua (≃sym E) = sym (ua E)
ua-sym E
 = ua (≃sym E)          =⟨ ap (λ - → ua (≃sym -)) (sym (ua-s E)) ⟩
   ua (≃sym (=-≃ (ua E)))   =⟨ ap ua (sym (=-≃-sym (ua E))) ⟩
   ua (=-≃ (sym (ua E)))    =⟨ ua-r (sym (ua E)) ⟩
   sym (ua E)  ■
```

4.3 Higher inductive types

We defined inductive types which are 'freely generated by some points' like $\mathbb{1}$, $+$, Σ, $\mathbb{N}$, and $\mathbb{Z}$. Now we declare the existence of some types which are 'freely generated by some points and higher paths'. Such a type is called a higher inductive type.

HoTT in Agda doesn't support the automated application of inference rules of higher inductive types, that is, there is no pattern-matching functionality for higher inductive types. Therefore we have to directly handle elimination rules by our hands and manually append some judgemental equalities corresponding to computation rules.

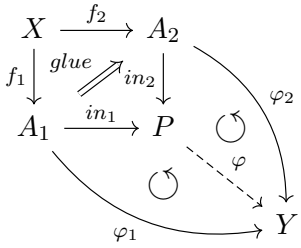
First write the following command in Agda, which will allow us to add some judgemental equalities.

```
{-# BUILTIN REWRITE _=_ #-}
```

4.3.1 Homotopy pushout

For any given prepushout diagram $A_1 \xleftarrow{f_1} X \xrightarrow{f_2} A_2$, we will declare the existence of an object P, called **homotopy pushout** of the given diagram, satisfying the following 'up to homotopy universal property': There are arrows $\mathtt{in}_k : A_k \to P$ for $k = 1, 2$ with a homotopy $\mathtt{glue} : \mathtt{in}_1 \circ f_1 \Rightarrow \mathtt{in}_2 \circ f_2$ such that for any two functions $\varphi_k : A_k \to Y$ with a homotopy $H : \varphi_1 \circ f_1 \Rightarrow \varphi_2 \circ f_2$, there exists $\varphi : P \to Y$ with two strictly commuting triangles $\varphi_k \equiv \varphi \circ \mathtt{in}_k$ and a homotopy witnessing $\varphi \circ_l \mathtt{glue} \sim H$, which is unique up to homotopy in a sense that for a given data $(\alpha, K_1, K_2, \mathtt{COH})$ where $\alpha : P \to Y$,

$K_1 : \varphi_1 \sim \alpha \circ \mathtt{in}_1$, $K_2 : \alpha \circ \mathtt{in}_2 \sim \varphi_2$ and $\mathtt{COH} : (K_1 \circ_r f_1) \bullet_h (\alpha \circ_l (\mathtt{glue})) \bullet_h (K_2 \circ_r f_2) \sim H$, we have $\alpha \sim \varphi$. COH stands for 'coherence'. The above setting can be depicted as below:



Now we formulate the above object for each prepushout diagram with the framework of type theory. Pushout formulation rule:

```
postulate
  pushout : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
          → (f₁ : X → A₁) (f₂ : X → A₂) → type (ℓ ⊔ ℓ' ⊔ ℓ'')
```

Pushout introduction rule:

```
postulate
  in₁ : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''} {f₁ : X → A₁} {f₂ : X → A₂}
      → A₁ → pushout f₁ f₂

  in₂ : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''} {f₁ : X → A₁} {f₂ : X → A₂}
      → A₂ → pushout f₁ f₂

  glue : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''} {f₁ : X → A₁} {f₂ : X → A₂}
      → (x : X) → in₁ {f₁ = f₁} {f₂ = f₂} (f₁ x) = in₂ (f₂ x)
```

Pushout elimination rule:

```
postulate
  pushout-elim : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
          → (f₁ : X → A₁) (f₂ : X → A₂)
          → (𝒫 : pushout f₁ f₂ → type ℓ''')
          → (φ₁ : Π (𝒫 ∘ in₁)) → (φ₂ : Π (𝒫 ∘ in₂))
          → (I : (x : X) → (φ₁ (f₁ x)) =↑ φ₂ (f₂ x) [ glue x ]over 𝒫 )
          → Π 𝒫
```

Pushout computation rule for point constructors:

```
postulate
  pushout-comp-1 : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
          → (f₁ : X → A₁) (f₂ : X → A₂)
          → (𝒫 : pushout f₁ f₂ → type ℓ''')
          → (φ₁ : Π (𝒫 ∘ in₁)) → (φ₂ : Π (𝒫 ∘ in₂))
          → (I : (x : X) → (φ₁ (f₁ x)) = ↑ φ₂ (f₂ x) [ glue x ]over 𝒫 )
          → (a1 : A₁)
          → (pushout-elim f₁ f₂ 𝒫 φ₁ φ₂ I) (in₁ a1) = (φ₁ a1)

  pushout-comp-2 : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
          → (f₁ : X → A₁) (f₂ : X → A₂)
          → (𝒫 : pushout f₁ f₂ → type ℓ''')
          → (φ₁ : Π (𝒫 ∘ in₁)) → (φ₂ : Π (𝒫 ∘ in₂))
          → (I : (x : X) → (φ₁ (f₁ x)) = ↑ φ₂ (f₂ x) [ glue x ]over 𝒫 )
          → (a2 : A₂)
          → (pushout-elim f₁ f₂ 𝒫 φ₁ φ₂ I) (in₂ a2) = (φ₂ a2)


{-# REWRITE pushout-comp-1 #-}
{-# REWRITE pushout-comp-2 #-}
```

By using the REWRITE command, we convert the above paths to judgemental equalities. Pushout computation rule for the path constructor:

```
postulate
  pushout-comp-glue : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
          → (f₁ : X → A₁) (f₂ : X → A₂)
          → (𝒫 : pushout f₁ f₂ → type ℓ''')
          → (φ₁ : Π (𝒫 ∘ in₁)) → (φ₂ : Π (𝒫 ∘ in₂))
          → (I : (x : X) → (φ₁ (f₁ x)) = ↑ φ₂ (f₂ x) [ glue x ]over 𝒫 )
          → (x : X) → apd (pushout-elim f₁ f₂ 𝒫 φ₁ φ₂ I) (glue x) = I x
```

It is convenient to stipulate the above rules for the non-dependent case, that is, the case where $\mathcal{P}$ is a constant type family. Recall that for the constant type family, `apd` can

be reduced to `tr-const` and `ap` through the derivation `apd-const`.

```
pushout-rec : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
           → (f₁ : X → A₁) (f₂ : X → A₂)
           → (Y : type ℓ''')
           → (φ₁ : A₁ → Y) → (φ₂ : A₂ → Y)
           → (I : (x : X) → (φ₁ (f₁ x)) = φ₂ (f₂ x) )
           → (pushout f₁ f₂) → Y
pushout-rec {X = X} f₁ f₂ Y φ₁ φ₂ I
  = pushout-elim f₁ f₂ (λ _ → Y) φ₁ φ₂
                 ((λ x → tr-const (glue x) (φ₁ (f₁ x))) •h I)

pushout-rec-comp-1 : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
           → (f₁ : X → A₁) (f₂ : X → A₂)
           → (Y : type ℓ''')
           → (φ₁ : A₁ → Y) → (φ₂ : A₂ → Y)
           → (I : (x : X) → (φ₁ (f₁ x)) = φ₂ (f₂ x) )
           → (a1 : A₁) → (pushout-rec f₁ f₂ Y φ₁ φ₂ I) (in₁ a1) = φ₁ a1
pushout-rec-comp-1 f₁ f₂ Y φ₁ φ₂ I a1 = refl _

pushout-rec-comp-2 : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
           → (f₁ : X → A₁) (f₂ : X → A₂)
           → (Y : type ℓ''')
           → (φ₁ : A₁ → Y) → (φ₂ : A₂ → Y)
           → (I : (x : X) → (φ₁ (f₁ x)) = φ₂ (f₂ x) )
           → (a2 : A₂) → (pushout-rec f₁ f₂ Y φ₁ φ₂ I) (in₂ a2) = φ₂ a2
pushout-rec-comp-2 f₁ f₂ Y φ₁ φ₂ I a2 = refl _

pushout-rec-comp-glue : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
           → (f₁ : X → A₁) (f₂ : X → A₂)
           → (Y : type ℓ''')
           → (φ₁ : A₁ → Y) → (φ₂ : A₂ → Y)
           → (I : (x : X) → (φ₁ (f₁ x)) = φ₂ (f₂ x) )
```

```
          → (x : X) → ap (pushout-rec f₁ f₂ Y φ₁ φ₂ I) (glue x) = I x
-- type checks by the above judgemental equality with (refl) !
pushout-rec-comp-glue {X = X} f₁ f₂ Y φ₁ φ₂ I x
  = ap s (glue x)  =⟨ ap (λ - → - • ap s (glue x)) (sym (•-sym-l q)) ⟩
    q ⁻¹ • q • ap s (glue x)   =⟨ •-assoc (q ⁻¹) q (ap s (glue x)) ⟩
    q ⁻¹ • (q • ap s (glue x)) =⟨ ap (λ - → q ⁻¹ • -)
                                     (sym (apd-const s (glue x))) ⟩
    q ⁻¹ • apd s (glue x) =⟨ ap (λ - → q ⁻¹ • -) (P x) ⟩
    q ⁻¹ • (q • (I x)) =⟨ sym (•-assoc (q ⁻¹) q (I x)) ⟩
    q ⁻¹ • q • I x =⟨ ap (λ - → - • I x) (•-sym-l q) ⟩
    I x ■
  where
  s = pushout-rec f₁ f₂ Y φ₁ φ₂ I
  P : (x₁ : X) →
      apd s (glue x₁) = ((λ t → tr-const (glue t) (φ₁ (f₁ t))) •ₕ I) x₁
  P = pushout-comp-glue f₁ f₂ (λ - → Y)
                        φ₁ φ₂ ((λ x → tr-const (glue x) (φ₁ (f₁ x))) •ₕ I)
  q = tr-const (glue x) (s (in₁ (f₁ x)))
```

Now we prove the 'up to homotopy universal property' of the homotopy pushout mentioned in the first part of this section. We start with defining the type of homotopies for a square:

```
square : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''} {Y : type ℓ'''}
         → (f₁ : X → A₁) (f₂ : X → A₂) (φ₁ : A₁ → Y) (φ₂ : A₂ → Y)
         → type (ℓ ⊔ ℓ''')
square f₁ f₂ φ₁ φ₂ = φ₁ ∘ f₁ ∼ φ₂ ∘ f₂
```

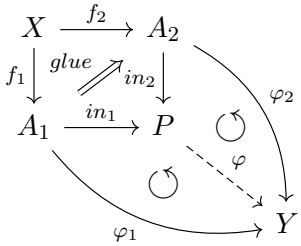
`glue` is the inhabitance for the square of pushout:

```
pushout-square : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''}
         → (f₁ : X → A₁) (f₂ : X → A₂)
         → square f₁ f₂ (in₁ {f₁ = f₁} {f₂ = f₂}) in₂
pushout-square f₁ f₂ = glue
```

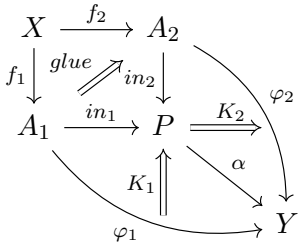
The existence part is just the recursion rule of pushout:

```
pushout-∃ : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''} {Y : type ℓ'''}
          → (f₁ : X → A₁) (f₂ : X → A₂) (φ₁ : A₁ → Y) (φ₂ : A₂ → Y)
          → square f₁ f₂ φ₁ φ₂
          → (pushout f₁ f₂) → Y
pushout-∃ {Y = Y} f₁ f₂ φ₁ φ₂ H = pushout-rec f₁ f₂ Y φ₁ φ₂ H
```

For the uniqueness part, let the data be given as below:



with a homotopy $H : \varphi_1 \circ f_1 \Rightarrow \varphi_2 \circ f_2$. Additionally, assume that a data $(\alpha, K_1, K_2, \texttt{COH})$ are given as below:



where $K_1 : \varphi_1 \sim \alpha \circ \texttt{in}_1$, $K_2 : \alpha \circ \texttt{in}_2 \sim \varphi_2$, and the coherence datum $\texttt{COH} : (K_1 \circ_r f_1) \bullet_h (\alpha \circ_l \texttt{glue}) \bullet_h (K_2 \circ_r f_2) \sim H$ stating that the concatenation of all $\Rightarrow$ shaped arrows in the above diagram with suitable whiskerings is homotopic to the given H.

We need to construct a section from P to the type family $\mathcal{P} :\equiv \lambda\ z \to \alpha(z) = \varphi(z)$. Recall that we have a strict commutativeness $\varphi_k \equiv \varphi \circ \texttt{in}_k$. For the section $\Pi\ (\mathcal{P} \circ \texttt{in}_1)$, we give $\texttt{ht-sym}\ K_1$. For the section $\Pi\ (\mathcal{P} \circ \texttt{in}_2)$, we give K_2. Now it remains to construct the `pathover` family over `glue`, that is, the section in $\Pi_{x:X}\ (\texttt{ht-sym}\ K_1)\ (f_1(x))) = \uparrow (K_2(f_2(x)))\ [\ \texttt{glue}\ x\ ]\texttt{over}\ \mathcal{P}$.

By the derivation `tr-path-btwmaps`, $\texttt{tr}^{\mathcal{P}}_{\texttt{glue}(x)}(K_1(f_1(x)))^{-1}$ reduces to $(\texttt{ap}_\alpha(\texttt{glue}(x)))^{-1} \bullet (K_1(f_1(x)))^{-1} \bullet \texttt{ap}_\varphi(\texttt{glue}(x))$.

But by `pushout-rec-comp-glue`, $\texttt{ap}_\varphi(\texttt{glue}(x))$ reduces to $H(x)$, and by the coherence datum `COH`, $H(x)$ reduces to $K_1(f_1(x)) \bullet \texttt{ap}_\alpha(\texttt{glue}(x)) \bullet K_2(f_2(x))$. Hence we can get a path from $\texttt{tr}^{\mathcal{P}}_{\texttt{glue}(x)}(K_1(f_1(x)))^{-1}$ to $(\texttt{ap}_\alpha(\texttt{glue}(x)))^{-1} \bullet (K_1(f_1(x)))^{-1} \bullet K_1(f_1(x)) \bullet \texttt{ap}_\alpha(\texttt{glue}(x)) \bullet K_2(f_2(x))$ which can be identified to $K_2(f_2(x))$. The above argument can be formalized as:

```
ASSOC' :  {X : type ℓ} {x1 x2 x3 x4 x5 x6 : X}
      → (p1 : x1 = x2) (p2 : x2 = x3) (p3 : x3 = x4) (p4 : x4 = x5) (p5 : x5 = x6)
      → p1 • p2 • (p3 • p4 • p5) = p1 • (p2 • p3 • p4) • p5
ASSOC' (refl _) (refl _) p3 p4 p5 = refl _


pushout-! : {X : type ℓ} {A₁ : type ℓ'} {A₂ : type ℓ''} {Y : type ℓ'''}
          → (f₁ : X → A₁) (f₂ : X → A₂) (φ₁ : A₁ → Y) (φ₂ : A₂ → Y)
          → (H : square f₁ f₂ φ₁ φ₂)
          → (α : (pushout f₁ f₂) → Y)
          → (K₁ : φ₁ ∼ α ∘ in₁) → (K₂ : α ∘ in₂ ∼ φ₂)
          → (K₁ ∘r f₁) •h (α ∘l (glue {f₁ = f₁} {f₂ = f₂})) •h (K₂ ∘r f₂) ∼ H
          → α ∼ (pushout-∃ f₁ f₂ φ₁ φ₂ H)
pushout-! {X = X} {Y = Y} f₁ f₂ φ₁ φ₂ H α K₁ K₂ COH
  = pushout-elim f₁ f₂ 𝒫
                 (ht-sym K₁)
                 K₂
                 G
  where
  φ = pushout-∃ f₁ f₂ φ₁ φ₂ H
  𝒫 = (λ z → α z = φ z)
  G : (x : X) → ((ht-sym K₁) (f₁ x)) = ↑ (K₂ (f₂ x)) [ glue x ]over 𝒫
  G x
   = tr (λ z → α z = φ z) (glue x) ((K₁ (f₁ x)) ⁻¹)
       =⟨ tr-path-btwmaps α φ (glue x) ((K₁ (f₁ x)) ⁻¹) ⟩
     ap α (glue x) ⁻¹ • K₁ (f₁ x) ⁻¹ • ap φ (glue x)
       =⟨ ap (λ - → ap α (glue x) ⁻¹ • K₁ (f₁ x) ⁻¹ • -) φ-comp ⟩
     ap α (glue x) ⁻¹ • K₁ (f₁ x) ⁻¹ • H x
       =⟨ ap (λ - → ap α (glue x) ⁻¹ • K₁ (f₁ x) ⁻¹ • -) (sym (COH x)) ⟩
     ap α (glue x) ⁻¹ • K₁ (f₁ x) ⁻¹ • (K₁ (f₁ x) • ap α (glue x) • K₂ (f₂ x))
       =⟨ ASSOC' (ap α (glue x) ⁻¹) (K₁ (f₁ x) ⁻¹) (K₁ (f₁ x)) (ap α (glue x)) (K₂ (f₂ x)) ⟩
     ap α (glue x) ⁻¹ • (K₁ (f₁ x) ⁻¹ • K₁ (f₁ x) • ap α (glue x)) • K₂ (f₂ x)
       =⟨ ap (λ - →  ap α (glue x) ⁻¹ • ( - • ap α (glue x)) • K₂ (f₂ x))
             (•-sym-l (K₁ (f₁ x))) ⟩
     ap α (glue x) ⁻¹ • ap α (glue x) • K₂ (f₂ x)
       =⟨ ap (λ - → - • K₂ (f₂ x)) (•-sym-l (ap α (glue x))) ⟩
```

```
 K₂ (f₂ x) ■
where
φ-comp : ap φ (glue x) = H x
φ-comp = pushout-rec-comp-glue f₁ f₂ Y φ₁ φ₂ H x
```

We have proved that homotopy pushout has the 'up to homotopy' universal property. From this, we can notice that homotopy pushout is unique up to type equivalence.

To discern the difference between pushout and homotopy pushout, we ignore the higher paths exceeding dimension 2. Then quotienting 1-paths up to 2-paths, we can regard each type as an ordinary groupoid (as a category). A function is a functor between groupoids and a homotopy between functions is a natural isomorphism between functors.

Now we notice that an ordinary colimit of a prepushout diagram of groupoids is hard to explicitly construct. Let a prepushout diagram of groupoids $A_1 \leftarrow X \rightarrow A_2$ be given. First note that there is a trivial functor indis : Set $\rightarrow$ Grpd, given by $S \mapsto$[a groupoid with the object set S and morphism set $S \times S$]. One can easily show that this becomes the right adjoint functor of the forgetful functor Grpd $\rightarrow$ Set, hence the forgetful functor preserves colimit. This forces the pushout groupoid's base set to simply be the usual set adjunction $\mathrm{ob}(A_1) \sqcup_{\mathrm{ob}(X)} \mathrm{ob}(A_2)$. However, a complex phenomenon emerges in arrows; a non-composable pair of arrows in A_1 or A_2 can be mapped to a pair of composable arrows by the inclusion functor. Hence one would need to handle the sequence of arrows in A_1 and A_2 which become composable through inclusion to formally define the morphism set of the pushout groupoid. We must be concerned about how the objects collapse through f_1 and f_2. This seems really hard to handle and compute.

However, if we consider the initial cocone of the prepushout diagram, the object is determined up to equivalence of category (rather than up to isomorphism), and one such groupoid can be constructed as the above declaration of higher inductive type. We disjointly sum the groupoids A_1 and A_2 and freely append one invertible arrow $f_1(x) \rightarrow f_2(x)$ for each $x \in X$ (which corresponds to `glue`(x) in the above HIT). This indeed becomes the homotopy initial cocone. This groupoid is much more manageable since we can directly use the information over A_1 and A_2.

Note that the above construction is reminiscent of the double mapping cylinder in topology; for each $x \in X$, we connect $f_1(x)$ and $f_2(x)$. If we consider each groupoid as a space(its fundamental groupoid), the above construction may correspond to taking

the fundamental groupoid of the double mapping cylinder and restricting it to the full subgroupoid generated by the base spaces. Indeed there seems to be an analogy between those concepts. We shortly take a digression into classical homotopy theory. The contents described below are taken from [8].

Let $\mathcal{T}$ be the convenient category of topological spaces (which is cartesian closed bi-complete full subcategory of Top). Let $\mathcal{I}$ be a small category. $\mathcal{T}$ has the model category structure where weak equivalences are homotopy equivalences and fibrations are arrows p satisfying homotopy lifting property with respect to all arrows to $\mathrm{dom}(p)$. Note that any space is a fibrant object.

Assume that $\mathcal{I}$ is a small category and let $\mathcal{T}^{\mathcal{I}}$ be a category of all diagrams from $\mathcal{I}$ to $\mathcal{T}$. It seems that we can give a model category structure on $\mathcal{T}^{\mathcal{I}}$ as follows: Fibrations are natural transformations that are object-wise fibration in $\mathcal{T}$. Weak equivalences are natural transformations that are object-wise homotopy equivalence in $\mathcal{T}$. Cofibrations are natural transformations that have the left lifting property with respect to all trivial fibrations. We can easily notice that any diagram is a fibrant diagram.

Using the functor - $\times I$, we can define a functor $\mathrm{Cyl} : \mathcal{T}^{\mathcal{I}} \to \mathcal{T}^{\mathcal{I}}$ where $\mathrm{Cyl}(F)(i) = F(i) \times I$. It seems to be a functorial choice of cylinder object of the above model category structure on $\mathcal{T}^{\mathcal{I}}$ with 'natural inclusions' $\mathrm{in}_0, \mathrm{in}_1 : \mathrm{id} \Rightarrow \mathrm{Cyl}$. This defines the left-homotopy; two natural transformations $\varphi_0, \varphi_1 : F \Rightarrow G$ are **diagram-homotopic** if there is a natural transformation $H : \mathrm{Cyl}(F) \Rightarrow G$ such that $H \circ \mathrm{in}_k = \varphi_k$, which is a big collection of homotopies $H_i : \varphi_0(i) \sim \varphi_1(i)$ compatible with induced arrows by functors $\mathrm{Cyl}(F)$ and G. It turns out that diagram-homotopy is compatible with colimit, that is, if a diagram-homotopy H is given as above, the induced maps $\mathrm{colim}\varphi_0, \mathrm{colim}\varphi_1 : \mathrm{colim}F \to \mathrm{colim}G$ are homotopic in $\mathcal{T}$. Hence two diagram-homotopy-equivalent diagrams yield homotopically equivalent colimits.

However, weak equivalences in $\mathcal{T}^{\mathcal{I}}$ are not. Consider the two prepushout diagram: $CS^0 \leftarrow S^0 \to CS^0$ and $* \leftarrow S^0 \to *$. There is an obvious weak equivalence between them, but their colimits are respectively S^1 and $*$.

Now it is natural to seek for a collection of diagrams where a **weak equivalence between them becomes the diagram-homotopy-equivalence**. Recall that every diagram is a fibrant object. Hence cofibrant diagrams work since weak equivalences coincide with the left-homotopy equivalences when related objects are fibrant and cofibrant. For

some important shapes $\mathcal{I}$, it is known that for any diagram F we can functorially find a cofibrant diagram $\bar{F}$ and weak equivalence $: \bar{F} \to F$, which is called "**cofibrant replacement**". Any two cofibrant replacements of a given diagram F are diagram-homotopy equivalent (hence yield homotopy-equivalent colimits). In classical homotopy theory, the construction of taking the cofibrant replacement of a diagram and then taking a colimit is called **(classical) homotopy colimit** and it is well-defined up to homotopy equivalence. Now we have a **homotopy invariant of weak-equivalence-class over $\mathcal{T}^{\mathcal{I}}$**.

We examine the case where $\mathcal{I}$ is the prepushout shape. For a diagram $A_1 \xleftarrow{f_1} X \xrightarrow{f_2} A_2$, we can take a cofibrant replacement $M_{f_1} \hookleftarrow X \hookrightarrow M_{f_2}$ where M_{f_k} is the mapping cylinder of f_k. Therefore we get the classical homotopy pushout as the "double mapping cylinder", analogous to the picture depicted from the homotopy pushout as HIT in our type theory.

There is one more analogy between homotopy pushouts in classical homotopy theory and our HIT. Recall that in classical homotopy theory, homotopy colimit is homotopy invariants of weak equivalence class over the diagram category. This phenomenon also holds for type theory. For given two prepushout diagrams of types and object-wise type equivalence natural transformation between them, we can construct a type equivalence between their homotopy pushout after dealing with some complex homotopies between functions.

It seems like taking homotopy pushout (in classical homotopy theory) and then taking the fundamental groupoid (now corresponds to types in our theory) corresponds to taking fundamental groupoid (first comes down to our types of HoTT) and then taking pushout "up to natural isomorphism". The correlation between those concepts seems to be explored in [7] with much more advanced language. We just take it for granted that there is up to homotopy invariants of weak equivalence in diagrams, which is more manipulable and computable at least for prepushout diagrams. Our theory declares that if our theory interprets in some model category, that should be closed under taking homotopy pushout.

4.3.2 Suspension, Spheres, and the circle

Suspension of a type X is defined as the homotopy pushout $\mathbb{1} \xleftarrow{\text{const}_\star} X \xrightarrow{\text{const}_\star} \mathbb{1}$. We denote it as $\mathbf{\Sigma}\, X$. **Spheres** are inductively defined using the suspension; `S^0`$:\equiv 2$ and `S^(n+1)` $:\equiv \mathbf{\Sigma}$ `S^n`.

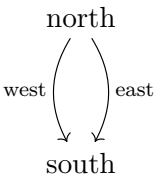
```
Σ : (X : type ℓ) → type ℓ
Σ X = pushout {X = X} const⋆ const⋆

S^_ : ℕ → 𝒰
S^ 0 = 2
S^ suc n = Σ (S^ n)
```

We introduce some notations for `S^1`:

```
north south : S^ 1
north = in₁ ⋆
south = in₂ ⋆

west east : north = south
west = glue 1
east = glue 2
```



How can `S^1` exactly be interpreted as the usual topological circle $\mathbb{S}^1$? Obviously, `S^1` is not the whole fundamental groupoid $\pi\mathbb{S}^1$. Recall that paths are primitive objects in HoTT. There are no 'intermediate' points in paths. It is identified as the full subgroupoid of $\pi\mathbb{S}^1$ generated by distinct two points of $\mathbb{S}^1$. Of course, we can think of a groupoid generated by a single point of $\mathbb{S}^1$, i.e. a fundamental group of $\mathbb{S}^1$. We can formulate such groupoid again by a declaration of a new higher inductive type, which we write as $\mathtt{S}^1$.

$\mathtt{S}^1$ formation and introduction rule:

```
postulate
  S¹ : 𝒰
  base : S¹
  loop : base = base
```

$\mathtt{S}^1$ elimination rule:

```
postulate
  S¹elim : (𝒫 : S¹ → type ℓ) (x : 𝒫 base) (ℓ : x = ↑ x  [ loop ]over 𝒫 )
        → (z : S¹) → 𝒫 z
```

$\mathbb{S}^1$ computation rule for the point constructor:

```
postulate
  S¹comp-base :  (𝒫 : S¹ → type ℓ) (x : 𝒫 base) (ℓ : x = ↑ x  [ loop ]over 𝒫)
              →  ((S¹elim 𝒫 x ℓ) base) = x
{-#  REWRITE S¹comp-base  #-}
```

$\mathbb{S}^1$ computation rule for the path constructor:

```
postulate
  S¹comp-loop :  (𝒫 : S¹ → type ℓ) (x : 𝒫 base) (ℓ : x = ↑ x [ loop ]over 𝒫)
              → apd (S¹elim 𝒫 x ℓ) loop = ℓ
```

It has a definite advantage over the inductively defined circle `S^1` since it has only one path constructor `loop`. Recall the hard work we had to get through when we constructed a section over a pushout. Giving a term for a `pathover` in a fibration seems to require some labor. For $\mathbb{S}^1$, we only need to construct one `pathover` lying over `loop`.

As expected, we can establish a type equivalence `S^ 1` $\cong \mathbb{S}^1$, but it will require some work. We start with stipulating the inference rules of $\mathbb{S}^1$ for the non-dependent case:

```
S¹rec : (A : type ℓ) (a : A) (p : a = a)
      → S¹ → A
S¹rec A a p = S¹elim (\ - → A) a (tr-const loop a • p)


S¹rec-comp-base : (A : type ℓ) (a : A) (p : a = a)
          → ((S¹rec A a p) base) = a
S¹rec-comp-base  A a p  = refl a   --judgementally holds by S¹comp-base
```

Recall again that we can reduce `apd` to `tr-const` and `ap` through the derivation `apd-const`.

```
S¹rec-comp-loop : (A : type ℓ) (a : A) (p : a = a)
          → ap (S¹rec A a p) loop = p
```

```
S¹rec-comp-loop  A a p
  = ap f loop    =⟨ ap (λ - → - • (ap f loop)) (sym (•-sym-l q)) ⟩
    (q ⁻¹ • q) • (ap f loop)    =⟨ •-assoc (q ⁻¹) q (ap f loop) ⟩
    q ⁻¹ • (q • ap f loop)    =⟨ ap (λ - → q ⁻¹ • -)
                                    (sym (apd-const f loop)) ⟩
    q ⁻¹ • (apd f loop)   =⟨ ap (λ - → q ⁻¹ • -)
                               (S¹comp-loop (λ - → A) a (q • p)) ⟩
    q ⁻¹ • (q • p)    =⟨ sym (•-assoc (q ⁻¹) q p) ⟩
    q ⁻¹ • q • p    =⟨ ap (λ - → - • p) (•-sym-l q) ⟩
    p ■
  where
  f = S¹rec A a p
  q = tr-const loop a


S¹rec-loop-sym :  (A : type ℓ) (a : A) (p : a = a)
      → ap (S¹rec A a p) (loop ⁻¹) = p ⁻¹
S¹rec-loop-sym A a p
  = ap f (sym loop)   =⟨ ap-sym f loop ⟩
    sym (ap f loop)   =⟨ ap (\ - → sym -) (S¹rec-comp-loop A a p) ⟩
    sym p   ■
  where
  f = S¹rec A a p
```

Now we construct the equivalence between two circles. First, define maps back and forth using recursion rules:

```
2toloopspace : 2 → base = base
2toloopspace ₁ = loop
2toloopspace ₂ = refl base


toS¹ : (S^ 1) → S¹
toS¹ = pushout-rec   const⋆ const⋆
        S¹
        (λ - → base)
```

```
        (λ - → base)
        2toloopspace

fromS¹ : S¹ → (S^ 1)
fromS¹ = S¹rec
        (S^ 1)
        north
        (west • (east ⁻¹))
```

toS¹ : (S^ 1) → S¹ is a map satisfying toS¹(north) ≡ toS¹(south) ≡ base and $\mathtt{ap}_{\mathtt{toS}^1}$ west = loop, $\mathtt{ap}_{\mathtt{toS}^1}$ east = refl base. fromS¹ : S¹ → (S^ 1) is a map satisfying fromS¹(base) ≡ north and $\mathtt{ap}_{\mathtt{fromS}^1}$ loop = west • (east ⁻¹).

We need to use the computation rules for path constructors several times so we stipulate them:

```
app-fromS¹-loop :  ap (fromS¹) loop = west • east ⁻¹
app-fromS¹-loop = S¹rec-comp-loop (S^ 1) north (west • (east ⁻¹))

app-toS¹ : (x : 2) → ap toS¹ (glue x) = 2toloopspace x
app-toS¹ = pushout-rec-comp-glue const⋆ const⋆ S¹
                                 (λ - → base) (λ - → base) 2toloopspace
app-toS¹-west : ap toS¹ west = loop
app-toS¹-west = app-toS¹ ₁
app-toS¹-east : ap toS¹ east = refl base
app-toS¹-east = app-toS¹ ₂
```

We finally get the equivalence:

```
tofrom : toS¹ ∘ fromS¹ ∼ id
tofrom = S¹elim
          𝒫
          loop
          looploop-over-loop
    where
    𝒫 = λ z → (toS¹ ∘ fromS¹) z = id z
    looploop-over-loop : loop = ↑ loop [ loop ]over 𝒫
```

```
  looploop-over-loop
    = tr 𝒫 loop loop
        =⟨ tr-path-btwmaps (toS¹ ∘ fromS¹) id loop loop ⟩
      (ap (toS¹ ∘ fromS¹) loop) ⁻¹ • loop • ap id loop
        =⟨ ap (λ - → ap (toS¹ ∘ fromS¹) loop ⁻¹ • loop • -) (ap-id loop) ⟩
      (ap (toS¹ ∘ fromS¹) loop) ⁻¹ • loop • loop
        =⟨ ap (λ - → (-) ⁻¹ • loop • loop) (ap-∘ toS¹ fromS¹ loop) ⟩
      (ap toS¹ (ap fromS¹ loop)) ⁻¹ • loop • loop
        =⟨ ap (λ - → (ap toS¹ -) ⁻¹ • loop • loop) app-fromS¹-loop ⟩
      (ap toS¹ (west • east ⁻¹)) ⁻¹ • loop • loop
        =⟨ ap (λ - → - • loop • loop) (sym (ap-sym toS¹ (west • east ⁻¹))) ⟩
      (ap toS¹ ((west • east ⁻¹) ⁻¹)) • loop • loop
        =⟨ ap (λ - → ap toS¹ - • loop • loop) (sym• west (sym east)) ⟩
      ap toS¹ (sym (sym east) • west ⁻¹) • loop • loop
        =⟨ ap (λ - → ap toS¹ (- • west ⁻¹) • loop • loop) (sym²~id east) ⟩
      ap toS¹ (east • west ⁻¹) • loop • loop
        =⟨ ap (λ - → - • loop • loop) (ap-• toS¹ east (west ⁻¹)) ⟩
      (ap toS¹ east) • (ap toS¹ (west ⁻¹)) • loop • loop
        =⟨ ap (λ - → - • loop • loop)
              (ap₂ (λ - ~ → - • ~) app-toS¹-east (ap-sym toS¹ west)) ⟩
      (ap toS¹ west) ⁻¹ • loop • loop
        =⟨ ap (λ - → (-) ⁻¹ • loop • loop) app-toS¹-west ⟩
      loop ⁻¹ • loop • loop
        =⟨ ap (λ - → - • loop) (•-sym-l loop) ⟩
      loop ■
```

```
fromto : fromS¹ ∘ toS¹ ∼ id
fromto = pushout-elim  const⋆ const⋆
           𝒫
           (λ {⋆ → refl north})
           (λ {⋆ → east})
           refltoeast-over-glue
  where
  𝒫 = (λ z → (fromS¹ ∘ toS¹) z = id z)
```

```
refltoeast-over-glue : (x : 2) → (refl north) = ↑ east [ glue x ]over 𝒫
refltoeast-over-glue ₁
  = tr 𝒫 west (refl north)
      =⟨ tr-path-btwmaps (fromS¹ ∘ toS¹) id west (refl north) ⟩
    (ap (fromS¹ ∘ toS¹) west)⁻¹ • (refl north) • (ap id west)
      =⟨ •-assoc ((ap (fromS¹ ∘ toS¹) west)⁻¹) (refl north) (ap id west) ⟩
    (ap (fromS¹ ∘ toS¹) west)⁻¹ • ap id west
      =⟨ ap (λ - → (ap (fromS¹ ∘ toS¹) west)⁻¹ • -) (ap-id west) ⟩
    (ap (fromS¹ ∘ toS¹) west)⁻¹ • west
      =⟨ ap (λ - → (-)⁻¹ • west) (ap-∘ fromS¹ toS¹ west) ⟩
    (ap fromS¹ (ap toS¹ west))⁻¹ • west
      =⟨ ap (λ - → (ap fromS¹ -) ⁻¹ • west) app-toS¹-west ⟩
    (ap fromS¹ loop)⁻¹ • west
      =⟨ ap (λ - → - ⁻¹ • west) app-fromS¹-loop ⟩
    (west • (east)⁻¹)⁻¹ • west
      =⟨ ap (λ - → - • west) (sym• west (east ⁻¹)) ⟩
    (east ⁻¹)⁻¹ • west ⁻¹ • west
      =⟨ ap (λ - → - • west ⁻¹ • west)
            (sym²~id east) • (•-assoc east (west ⁻¹) west) ⟩
    east • (west ⁻¹ • west)
      =⟨ ap (λ - → east • -) (•-sym-l west) • (•-refl-r east) ⟩
    east ■

refltoeast-over-glue ₂
  = tr 𝒫 east (refl north)
      =⟨ tr-path-btwmaps (fromS¹ ∘ toS¹) id east (refl north) ⟩
    (ap (fromS¹ ∘ toS¹) east)⁻¹ • refl north • ap id east
      =⟨ •-assoc ((ap (fromS¹ ∘ toS¹) east)⁻¹) (refl north) (ap id east) ⟩
    (ap (fromS¹ ∘ toS¹) east)⁻¹ • ap id east
      =⟨ ap (λ - → (ap (fromS¹ ∘ toS¹) east)⁻¹ • -) (ap-id east) ⟩
    (ap (fromS¹ ∘ toS¹) east)⁻¹ • east
      =⟨ ap (λ - → (-)⁻¹ • east) (ap-∘ fromS¹ toS¹ east) ⟩
    (ap fromS¹ (ap toS¹ east))⁻¹ • east
      =⟨ ap (λ - → (ap fromS¹ -)⁻¹ • east) app-toS¹-east ⟩
    east ■

S^1≅S¹ : S^ 1 ≅ S¹
```

```
S^1≅S¹ = ≅pf toS¹ (Ivtbl fromS¹ tofrom fromto)
```

Chapter 5

The Fundamental Group of The Circle

There is a type constructor which is also defined as a higher inductive type called k-truncation. k-truncation trivializes higher structures above the dimension $k-2$ by appending a k-path between any given two parallel $(k-1)$-paths. A k-truncated type becomes a type of h-level k. For the case $k=2$, we call the constructor a set truncation. I want to emphasize once again that our h-level is inductively defined starting from 0, not the conventional -2.

n-th homotopy group of a type X at $x : X$ is defined as the set truncation of n-dimensional loop space at x and denoted as $\pi^n(X, x)$. For example, $\pi^2(X, x)$ is the set truncation of $\mathtt{refl}_x =_{x=x} \mathtt{refl}_x$.

Rather than prove $\pi^1(\mathbb{S}^1, \mathtt{base}) \simeq \mathbb{Z}$, we actually prove stronger statement $(\mathtt{base} = \mathtt{base}) \cong \mathbb{Z}$. That is, the loop space already has no interesting higher structure before we truncate it to a set.

We denote the loop space `(base = base)` as $\Omega^1\mathbb{S}^1$.

```
Ω¹S¹ = base = base
```

By pattern matching, we can define a natural map $\mathbb{Z} \to \Omega^1\mathbb{S}^1$:

```
loop^_ : ℤ → Ω¹S¹
loop^ pos 0 = refl base
loop^ pos (suc n) = (loop^ (pos n)) • loop
loop^ negsuc 0 = loop ⁻¹
```

```
loop^ negsuc (suc n) = (loop^ (negsuc n)) • loop ⁻¹
```

This will be the one direction of a function constituting the equivalence (`base = base`) $\cong \mathbb{Z}$. However, there is no direct way to construct its inverse (`base = base`) $\to \mathbb{Z}$ such that $\texttt{loop}^n \mapsto n$. We try to define a suitable fibration over $\mathbb{S}^1$ whose fiber at `base` is $\mathbb{Z}$, faithfully recording the information of loop space as an action on it.

We define a covering space over $\mathbb{S}^1$ by $\mathbb{S}^1$-recursion, which has a fiber $\mathbb{Z}$ at `base`, anticipating to be the universal cover. According to the recursion rule. we need to give a path $\mathbb{Z} =_{\mathcal{U}} \mathbb{Z}$ over `loop` and we expect the action of `loop` over the fiber at $\mathbb{Z}$ to be the application of `succ`$\mathbb{Z}$. It seems that the univalence axiom is the only way to give such a path in the universe. We see here an interesting interplay between the univalence axiom and the higher inductive type. Higher inductive types require paths in the universe to construct a fibration over them, and those can be provided by the univalence axiom.

```
cover : S¹ → 𝒰
cover = S¹rec  𝒰  ℤ  (ua succ-≃)
```

By using `tr-ua : (E : X ≃ Y) → tr-id (ua E) = (_≃_.eqv E)`, we can prove that the action of `loop` on the fiber is what we have expected. Recall that transport is compatible with function composition and `cover` $\equiv \texttt{id}_{\mathcal{U}} \circ$ `cover`.

```
loopact=succ : tr cover loop = succℤ
loopact=succ
  = tr cover loop =⟨ tr-∘ id cover loop ⟩
    tr id (ap cover loop) =⟨ ap (λ - → tr id -)
                                (S¹rec-comp-loop 𝒰 ℤ (ua succ-≃)) ⟩
    tr id (ua succ-≃) =⟨ tr-ua succ-≃ ⟩
    succℤ ■


loop⁻¹act=pred : tr cover (loop ⁻¹) = predℤ
loop⁻¹act=pred
  = tr cover (loop ⁻¹) =⟨ tr-∘ id cover (loop ⁻¹) ⟩
  tr id (ap cover (loop ⁻¹)) =⟨ ap (λ - → tr id -) (ap-sym cover loop) ⟩
  tr id ((ap cover loop)⁻¹) =⟨ ap (λ - → tr id (- ⁻¹))
                                  (S¹rec-comp-loop 𝒰 ℤ (ua succ-≃)) ⟩
```

```
  tr id ((ua succ-≃)⁻¹) =⟨ ap (λ - → tr id -) (sym (ua-sym (succ-≃))) ⟩
  tr id (ua (≃sym succ-≃)) =⟨ tr-ua (≃sym succ-≃) ⟩
  predℤ ■
```

Now we can define the map (`base = base`) $\to \mathbb{Z}$ as the evaluation at `base` of the below section over $\mathbb{S}^1$, which is defined by the action of path on `pos 0 : ℤ`

```
encode : (z : S¹) → (base = z → cover z)
encode z p = tr cover p (pos 0)
```

We construct the fiberwise inverse of `encode` by $\mathbb{S}^1$-elimination rule. Its value at `base` should be `loop^_`. We need to give a `pathover` from `loop^_` to `loop^_` over `loop`. Using the gadget on transportation of a fiberwise map, transport of `loop^_` along the `loop` reduces to `tr (λ z → base = z) loop ∘ loop^_ ∘ tr cover (loop ⁻¹)`. Now the action of `loop ⁻¹` over `cover` reduces to `predℤ` and `tr (λ z → base = z) loop` reduces to a concatenating `loop` at right.

```
decode : (z : S¹) → (cover z → base = z)
decode = S¹elim
           𝒫
           loop^_
           loop^toloop^-over-loop
  where
  𝒫 = (λ z → cover z → base = z)
  loop^toloop^-over-loop : loop^_ = ↑ loop^_ [ loop ]over 𝒫
  loop^toloop^-over-loop
    = tr 𝒫 loop loop^_
        =⟨ tr-fibmap cover (_=_ base) loop loop^_ ⟩
      tr (_=_ base) loop ∘ loop^_ ∘ tr cover (loop ⁻¹)
        =⟨ ap (λ - → tr (_=_ base) loop ∘ loop^_ ∘ -) loop⁻¹act=pred ⟩
      tr (_=_ base) loop ∘ loop^_ ∘ predℤ
        =⟨ ap (λ - → - ∘ loop^_ ∘ predℤ) (fext (tr-path-lfix base loop)) ⟩
      (λ - → - • loop) ∘ loop^_ ∘ predℤ
        =⟨ fext H ⟩
      loop^_ ■
```

```
  where
  H : (λ - → - • loop) ∘ loop^_ ∘ predℤ   ∼   loop^_
  H (pos 0) = •-sym-l loop
  H (pos (suc n)) = refl _
  H (negsuc x)
   = (loop^ negsuc x) • loop ⁻¹ • loop
       =⟨ •-assoc (loop^ negsuc x) (loop ⁻¹) loop ⟩
     (loop^ negsuc x) • (loop ⁻¹ • loop)
       =⟨ ap (λ - → (loop^ negsuc x) • -) (•-sym-l loop) ⟩
     (loop^ negsuc x) • (refl _)
       =⟨ •-refl-r (loop^ negsuc x) ⟩
     (loop^ negsuc x) ■
```

It remains to prove that encode and decode constitute an inverse pair on each fiber of $\mathbb{S}^1$. First we show that for all `z : S¹`, `(encode z) ∘ (decode z) ∼ id`. Apply $\mathbb{S}^1$-elimination rule. The value at `base` can be easily constructed by induction on $\mathbb{Z}$. We write it as `htpy-base : (m : ℤ) → tr cover (loop^ m) (pos 0) = m`. Now we have to give a `pathover` from `htpy-base` to `htpy-base` over `loop`. By the function extensionality, it suffices to establish a homotopy between two sections `(n : ℤ) → (htpy-base ↶ loop) n = htpy-base n`. Note that each side is a path in $\mathbb{Z}$. Recall that $\mathbb{Z}$ is a set, stating that any 2-path space between parallel two 1-paths is trivial. This gives a required homotopy.

```
encode-decode : (z : S¹) → (encode z) ∘ (decode z) ∼ id
encode-decode = S¹elim
                  𝒫
                  htpy-base
                  (fext (λ n → set→set' ℤ-is-set _ _ _ _))
  where
  𝒫 = (λ z → encode z ∘ decode z ∼ id)
  htpy-base : (m : ℤ) → tr cover (loop^ m) (pos 0) = m
  htpy-base (pos 0) = refl (pos 0)
  htpy-base (pos (suc n))
      = tr cover ((loop^ pos n) • loop) (pos 0)
```

```
          =⟨ ap (λ - → - (pos 0)) (tr-• cover (loop^ pos n) loop) ⟩
        tr cover loop (tr cover (loop^ pos n) (pos 0))
          =⟨ ap (λ - → tr cover loop -) (htpy-base (pos n)) ⟩
        tr cover loop (pos n)
          =⟨ ap (λ - → - (pos n)) loopact=succ ⟩
        pos (suc n) ■
  htpy-base (negsuc 0)
    = tr cover (loop ⁻¹) (pos 0) =⟨ ap (λ - → - (pos 0)) loop⁻¹act=pred ⟩
      (negsuc 0) ■
  htpy-base (negsuc (suc n))
    = tr cover ((loop^ negsuc n) • loop ⁻¹) (pos 0)
          =⟨ ap (λ - → - (pos 0)) (tr-• cover (loop^ negsuc n) (loop ⁻¹)) ⟩
        tr cover (loop ⁻¹) (tr cover (loop^ negsuc n) (pos 0))
          =⟨ ap (λ - → tr cover (loop ⁻¹) -) (htpy-base (negsuc n)) ⟩
        tr cover (loop ⁻¹) (negsuc n)
          =⟨ ap (λ - → - (negsuc n)) loop⁻¹act=pred ⟩
        negsuc (suc n) ■
```

The other direction is easily proved by the path induction. `(encode base) (refl base)` reduces to `(pos 0)`, and `(decode base) (pos 0)` reduces to `loop^ (pos 0) ≡ refl base`.

```
decode-encode : (z : S¹) → (decode z) ∘ (encode z) ∼ id
decode-encode base (refl base) = refl (refl base)
```

We have constructed all the data required for the fiberwise equivalence:

```
S¹-fiberwise-eqv : (z : S¹) → (base = z) ≃ (cover z)
S¹-fiberwise-eqv z
  = ≅-≃ (≅pf (encode z)
          (Ivtbl (decode z) (encode-decode z) (decode-encode z)))
```

It can be shown that a fiberwise equivalence induces their total space equivalence. And it is easy to prove that for any type A and its term $a : A$, the total space $\Sigma_{x:A}(a = x)$ is contractible. Hence the above term `S¹-fiberwise-eqv` shows that the total space of `cover` is contractible as we anticipated. The `cover` is the universal cover.

Lastly, we get the desired:

```
Ω¹S¹≃ℤ : Ω¹S¹ ≃ ℤ
Ω¹S¹≃ℤ = S¹-fiberwise-eqv base
```

Chapter 6

Conclusion

We have observed how the inference rules of HoTT are given and used them to prove the usual fact in algebraic topology from scratch. We saw that the identity types can be interpreted as path spaces and related inference rules are strong enough to construct various tools for the homotopy theory. By the univalence axiom and higher inductive types, we could introduce objects with interesting higher structures and give them fibrations to analyze them.

When defining a new formal system, it is crucial to construct its semantics to justify it. Recall the fact that the slight modification of the univalence axiom (namely, postulate `equiv (=-≅)` instead of `equiv (=-≃)`) makes our system inconsistent. By finding a model of HoTT in 'classical mathematics' we can bring the authority and belief in the consistency of mathematics to our theory. This work can be found in the paper [4]. It seems a suitable goal for further study of HoTT, and the paper requires advanced knowledge of category theory, model category, and simplicial model.

Bibliography

[1] Steve Awodey. Type theory and homotopy, 2010.

[2] Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with agda, 2022.

[3] M. Hovey. *Model Categories*. Mathematical surveys and monographs. American Mathematical Society, 2007.

[4] Chris Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after voevodsky), 2018.

[5] Egbert Rijke. Introduction to homotopy type theory, 2022.

[6] Egbert Rijke, Fredrik Bakke, Raymond Baker, Jonathan Prieto-Cubides, and Vojtch tpaník. agda-unimath - identity types. https://unimath.github.io/agda-unimath/foundation-core.identity-types.html.

[7] Michael Shulman. Homotopy limits and colimits and enriched homotopy theory, 2009.

[8] J. Strom. *Modern Classical Homotopy Theory*. Graduate Studies in Mathematics. American Mathematical Society, 2023.

[9] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[10] Vladimir Voevodsky. The origins and motivations of univalent foundations - a personal mission to develop computer proof verification to avoid mathematical mistakes. https://www.ias.edu/ideas/2014/voevodsky-origins, 2014.

[11] Sangjin Yoon. agda-files-for-survey-on-homotopy-type-theory. https://github.com/septembraljourney/agda-files-for-survey-on-homotopy-type-theory, 2023.

요 약

호모토피 유형론(줄여서 HoTT)은 수학의 호모토피 이론과 컴퓨터과학의 유형론의 교차점에 있는 수학의 신생 분야이다. 유형론은 (ZFC 집합론이 기반하고 있는) 일차 논리 시스템과 같은 형식 체계의 일종으로서 러셀의 저서인 수학원리에서 발원하여 컴퓨터 과학자들과 논리학자들로부터 독자적으로 발전하여 현재에 이른다. 유형론은 이 분야의 형식 체계들이 공유하는 공통적인 형식이나 철학을 지칭하기도 하고, 그 각각의 개별적인 형식 체계를 지칭하기도 한다. HoTT는 그 개별적인 것들의 하나로서의 형식 체계이며, 그 자체로 호모토피적으로 해석 가능한 여러 현상들을 내포하고 있어 호모토피 이론과 관련된 수학을 위한 새로운 수학의 기초가 될 수 있다. 특히 유형론은 그것에 기반한 여러 증명보조기들이 있어 컴퓨터로의 이식이 매우 용이하다는 장점이 있고, 이를 통해 HoTT의 언어로 이미 여러 호모토피 이론의 증명들이 Agda와 Coq와 같은 증명보조기들에서 형식화되어있다. 요약하자면, HoTT는 컴퓨터로의 이식이 용이한, 호모토피 이론을 공리적으로 탐구할 수 있는 새로운 형식체계이다. 이 새로운 이론에 대한 비형식적 서술은 이미 좋은 교재들에서 소개되어 있으므로, 이 논문에서는 이 형식체계의 정의와 정리들이 어떻게 Agda 증명보조기에서 형식화될 수 있는지를 다룬다. 특히, S^1 의 기본군이 $\mathbb{Z}$ 임을 형식화하는 것을 목표로 한다.

주요어: 호모토피 유형론, 증명보조기, Agda

학 번: 2022-27298