Table of Contents

1 Installation **1**

Mini-HoTT is a basic Agda[5] library which contains basic definitions and results in Univalent type theory[6]. There is no guarantee whatsoever of any kind. At the moment, this library suffers of many changes without any warning.

- Website documentation: https://mini-hott.readthedocs.io/

Quick start

1 Installation

The only prerequisite is to have installed the latest version of Agda[7] and a text editor with Agda support, e.g., Emacs or Atom. Then, you can install the library as usual after cloning the sources by running the following command.

```
$ git clone http://github.com/jonaprieto/mini-hott
```

For newcomers, the easiest way to install a library is using agda-pkg[8] You can run the following commands to install it:

```
$ pip3 install agda-pkg
$ apkg init
$ apkg install --github jonaprieto/mini-hott
```

After installing the sources, just include in your code at the top the following line:

```
open import MiniHoTT
```

1.1 Quick start

This documentation is generated automatically. The sub-folder `src` in the repository of this project contains the Agda sources.

1.1.1 Code conventions

Definitions and theorems are typed with unicode characters. Lemmas and theorems are shown as rule inferences as much as possible. Level universes are explicitly given.

1.1.2 Proof relevancy

To be consistent with univalent type theory, we tell Agda to not use *Axiom K* for type-checking by using the option `without-K` on the top of the files.

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

https://travis-ci.org/jonaprieto/mini-hott
https://GitHub.com/jonaprieto/mini-hott/issues/
https://lbesson.mit-license.org/
https://GitHub.com/jonaprieto/mini-hott/tags/
http://github.com/agda/agda
http://homotopytypetheory.org/
http://github.com/agda/agda
http://github.com/agda/agda-pkg

In addition, without the Axiom K, Agda's `Set` is not a good name for universes in HoTT. So, we rename `Set` to `Type`. Our type judgments then will include the universe level as one explicit argument. Also, we want to have judgmental equalities for each usage of (=) so we use the following option.

```
{-# OPTIONS --exact-split #-}
```

```
open import Agda.Primitive
  using ( Level ; lsuc; lzero; _⊔_ ) public
```

Note that `l ⊔ q` is the maximum of two hierarchy levels `l` and `q` and we use this later on to define types in full generality.

```
Type
  : (ℓ : Level)
  → Set (lsuc ℓ)

Type ℓ = Set ℓ
```

```
Type₀
  : Type (lsuc lzero)

Type₀ = Type lzero
```

```
Type₁
  : Type (lsuc (lsuc lzero))

Type₁ = Type (lsuc lzero)
```

The following type is to lift a type to a higher universe.

```
record
  ↑ {a : Level} ℓ (A : Type a)
    : Type (a ⊔ ℓ)
  where
  constructor Lift
  field
    lower : A

open ↑ public
```

1.2 Everything in Mini HoTT

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

module MiniHoTT where

  open import Intro public

  open import BasicTypes public
  open import HLevelTypes public

  open import BasicFunctions public

  open import AlgebraOnPaths public

  open import Transport public
  open import TransportLemmas public

  open import AlgebraOnDependentPaths public
```

(continues on next page)

(continued from previous page)

```
open import ProductIdentities public
open import CoproductIdentities public

open import FibreType public

open import EquivalenceType public

open import DependentAlgebra public

open import HomotopyType public
open import HomotopyLemmas public

open import FunExtAxiom public
open import FunExtTransport public
open import FunExtTransportDependent public

open import DecidableEquality public

open import HalfAdjointType public

open import QuasiinverseType public
open import QuasiinverseLemmas public


open import SigmaEquivalence public
open import SigmaPreserves public

open import PiPreserves public

open import UnivalenceAxiom public

open import HLevelLemmas public

open import HedbergLemmas public

open import UnivalenceIdEquiv public
open import UnivalenceLemmas public

open import EquivalenceReasoning public
open import UnivalenceTransport public

open import Rewriting public
open import CircleType public
open import IntervalType public
open import SuspensionType public
open import TruncationType public
open import SetTruncationType public

open import TypesofMorphisms public

open import NaturalType public
open import IntegerType public


open import QuotientType public
open import RelationType public

open import MonoidType public
open import GroupType public
```

(continues on next page)

(continued from previous page)

```
  open import BasicEquivalences public

  open import Connectedness public

  open import FundamentalGroupType public
```

```
{-# OPTIONS --without-K --exact-split #-}
open import Intro public
```

1.3 Types

1.3.1 Empty type

A type without points is the *empty* type (also called falsehood).

** Declaration **

```
data
  𝟘 (ℓ : Level)
    : Type ℓ
  where
```

** Additionally syntax **

```
⊥     = 𝟘
Empty = 𝟘
```

** Elimination principle **

Ex falso quodlibet:

```
exfalso
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}
  ------------
  → (⊥ ℓ₂ → A)

exfalso ()
```

** Additionally syntax **

```
Empty-elim = exfalso
⊥-elim     = exfalso
𝟘-elim     = exfalso
```

We abbreviate functions with codomain the empty type by using the prefix symbol of the negation.

```
¬ : ∀ {ℓ : Level} → Type ℓ → Type ℓ
¬ {ℓ} A = (A → ⊥ ℓ)
```

1.3.2 Unit type

Consequently, we now consider the type with one point (also called *unit* type). This type is defined in Agda as a record, this because of the η-rule definitionally we get.

** Declaration **

```
record
  𝟙 (ℓ : Level)
    : Type ℓ
  where
  constructor unit
```

** Additionally syntax **

For this type:

```
Unit = 𝟙
⊤    = 𝟙
```

For the data constructor:

```
pattern ⋆ = unit
pattern ∗ = unit
```

** Elimination principle **

```
unit-elim
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₂}
  → (a : A)
  ------------
  → (𝟙 ℓ₁  → A)
unit-elim a ∗ = a
```

1.3.3 Two-point type

** Declaration **

```
data
  ⊭ (ℓ : Level)
    : Type (lsuc ℓ)
  where
  0₂ : ⊭ ℓ
  𝟙₂ : ⊭ ℓ
```

** Additionally syntax **

```
Bool = ⊭ lzero
```

Constructors synonyms:

```
false : ⊭ lzero
false = 0₂

true : ⊭ lzero
true  = 1₂

ff = false
tt = true
```

1.3.4 Natural numbers

** Declaration **

```
data
  ℕ : Type lzero
  where
  zero : ℕ
  succ : ℕ → ℕ

{-# BUILTIN NATURAL ℕ #-}
```

** Additionally syntax **

```
Nat = ℕ
```

** An order relation **

```
module ℕ-ordering (ℓ : Level) where
  _<_ : ℕ → ℕ → Type ℓ
  zero   < zero   = ⊥ _
  zero   < succ b = ⊤ _
  succ _ < zero   = ⊥ _
  succ a < succ b = a < b
```

```
_>_ : ℕ → ℕ → Type ℓ
a > b = b < a
```

1.3.5 $\sum$-types

Dependent sum type is a type of pairs where the second term may depend on the first.

** Declaration **

```
record
  ∑ {ℓ₁ ℓ₂ : Level}
   (A : Type ℓ₁) (B : A → Type ℓ₂)
   ------------------------------
   : Type (ℓ₁ ⊔ ℓ₂)
  where
  constructor _,_
  field
```

(continues on next page)

(continued from previous page)

```
    π₁ : A
    π₂ : B π₁

infixr 60 _,_
open ∑ public

{-# BUILTIN SIGMA ∑ #-}
```

**** Additionally syntax ****

```
Σ = ∑ -- \Sigma and \sum

syntax ∑ A (λ a → B) = ∑[ a : A ] B

Σ-s0 : ∀ {ℓ₁ ℓ₂} (A : Type ℓ₁) → (A → Type ℓ₂) → Type (ℓ₁ ⊔ ℓ₂)
Σ-s0 A = Σ _
syntax Σ-s0 A (λ x → B) = Σ[ x : A ] B

Σ-s1 : ∀ {ℓ₁ ℓ₂} {A : Type ℓ₁} → (A → Type ℓ₂) → Type (ℓ₁ ⊔ ℓ₂)
Σ-s1 = Σ _
syntax Σ-s1 (λ x → B) = ∑[ x ] B

Σ-s2 : ∀ {ℓ₁ ℓ₂} {A : Type ℓ₁} → (A → Type ℓ₂) → Type (ℓ₁ ⊔ ℓ₂)
Σ-s2 = Σ _
syntax Σ-s2 (λ x → B) = Σ[ x ] B
```

Constructor synonyms:

```
proj₁ = π₁
proj₂ = π₂

pr₁   = π₁
pr₂   = π₂

fst   = π₁
snd   = π₂

#     =  π₁
```

We use the built-in Σ-type in Agda, thus, we "pattern match" instead of declaring a elimination principle for it.

1.3.6 Π-types

In dependent type theories, the notion of a function is extended by the notion of a *dependent* function. These are those functions where the codomain may dependent on values of its domain.

**** Declaration ****

```
Π
  : ∀ {ℓ₁ ℓ₂ : Level}
  → (A : Type ℓ₁) (B : A → Type ℓ₂)
  ---------------------------------
  → Type (ℓ₁ ⊔ ℓ₂)

Π A B = (x : A) → B x
```

**** Additionally syntax ****

```
-- \prod vs \Pi
Π = ∏

syntax ∏ A (λ a → B) = ∏[ a : A ] B

∏-s0 : ∀ {ℓ₁ ℓ₂} (A : Type ℓ₁) → (A → Type ℓ₂) → Type (ℓ₁ ⊔ ℓ₂)
∏-s0 A = ∏ _
syntax ∏-s0 A (λ x → B) = Π[ x : A ] B

∏-s1 : ∀ {ℓ₁ ℓ₂} {A : Type ℓ₁} → (A → Type ℓ₂) → Type (ℓ₁ ⊔ ℓ₂)
∏-s1 = ∏ _
syntax ∏-s1 (λ x → B) = ∏[ x ] B

∏-s2 : ∀ {ℓ₁ ℓ₂} {A : Type ℓ₁} → (A → Type ℓ₂) → Type (ℓ₁ ⊔ ℓ₂)
∏-s2 = ∏ _
syntax ∏-s2 (λ x → B) = Π[ x ] B
```

1.3.7 Products

A particular case of Σ-types is the type of *products*. A product of two types A and B is the collection of pairs between an element of type A with one of type B. However, there is no relation between those two.

**** Declaration ****

```
_×_
  : ∀ {ℓ₁ ℓ₂ : Level}
  → (A : Type ℓ₁) (B : Type ℓ₂)
  ----------------------------
  → Type (ℓ₁ ⊔ ℓ₂)

A × B = ∑ A (λ _ → B)

infixl  39 _×_
```

1.3.8 Coproducts

A coproduct between types A and B (also called sum types) is a type of their *disjoint union*, i.e., this type is formed by tagging which elements comes from the type A and B. The tags are the constructor for this type, named here as `inr` or `inl`, that stands for right and left injection, respectively.

**** Declaration ****

```
data
  _+_ {ℓ₁ ℓ₂ : Level} (A : Type ℓ₁)(B : Type ℓ₂)
    : Type (ℓ₁ ⊔ ℓ₂)
  where
  inl : A → A + B
  inr : B → A + B

infixr 31 _+_
```

**** Elimination principle ****

```
+-elim
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}
  → {A : Type ℓ₁}{B : Type ℓ₂} {C : Type ℓ₃}
  → (A → C) → (B → C)
  -------------------
  → (A + B) → C

+-elim A→C _  (inl x) = A→C x
+-elim _  B→C (inr x) = B→C x
```

**** Additionally syntax ****

```
cases = +-elim

syntax cases f g = ⟨ f + g ⟩
```

1.3.9 Finite sets

Among the diffrent way, one can define *finite* types, We opt to use two version, the first version is a $\sum$-type while the second one is a sum type. Each definition offers its own advantages and drawbacks. The former is much clear while the latter is more practical.

A *finite type* of $n : \mathsf{N}$ elements is of type Fin_n. This type is the collection of natural numbers strictly less than n. We will prove later on that, indeed, these finite types are sets, and any finite type is equivalent to some n-finite type.

**** Declaration ****

```
Fin : ∀ {ℓ : Level} → ℕ → Type ℓ
Fin {ℓ} n = Σ ℕ (λ m → m < n)
  where open ℕ-ordering ℓ
```

**** Additionally syntax ****

```
syntax Fin n = ⟦ n ⟧
```

**** The function bound-of ****

```
bound-of : ∀ {ℓ : Level} {n : ℕ} → Fin {ℓ} n → ℕ
bound-of {n = n} _ = n
```

Another definition for finite sets we use is the following.

**** Alternative Declaration ****

```
module _ {ℓ : Level} where

  ⟦_⟧₂ : ℕ → Type ℓ
  ⟦_⟧₂ zero      = 0 _
  ⟦_⟧₂ (succ n)  = 𝟙 ℓ + ⟦ n ⟧₂
```

** Alternative fin-succ **

```
⟦⟧₂-succ
  : {n : ℕ}
  → ⟦ n ⟧₂ → ⟦ succ n ⟧₂

⟦⟧₂-succ {succ n} (inl x) = inr (inl unit)
⟦⟧₂-succ {succ n} (inr x) = inr (⟦⟧₂-succ x)
```

** Alternative fin-pred **

```
⟦⟧₂-pred
  : ∀ (n : ℕ)
  → ⟦ n ⟧₂ → ⟦ n ⟧₂

⟦⟧₂-pred (succ n) (inl x) = inl x
⟦⟧₂-pred (succ n) (inr x) = inr (⟦⟧₂-pred n x)
```

1.3.10 Equalities

In HoTT, we have a different interpretation of type theory in which the set-theoretical notion of *sets* for *types* is replaced by the topological notion of *spaces*.

The (homogeneous) equality type also called identity type is considered a primary type (included in the theory by default). We denote the identity type between $a, b : A$ as $a =_A b$ (also denoted by $\mathsf{Id}_A(a, b)$ or $a \rightsquigarrow b$. For the identity type, there is only one constructor, one way to inhabit such types. This is the reflexivity path (also called idp or refl).

** Declariton **

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

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

** Additionally syntax **

```
Eq   = _==_
Id   = _==_
Path = _==_
_⇝_  = _==_   -- type this '\r~'
_≡_  = _==_
```

(continues on next page)

(continued from previous page)

```
infix 30 _==_ _⇝_ _≡_

_≢_ : ∀ {ℓ : Level}{A : Type ℓ}(x y : A) → Type ℓ
x ≢ y = ¬ (x == y)
```

**** Reflexivity path of a given point ****

```
refl
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (a : A)
  ---------
  → a == a

refl  a = idp
```

**** Symmetry of a path ****

```
sym
  : ∀ {ℓ : Level} {A : Type ℓ} {x y : A}
  → x == y
  --------
  → y == x

sym idp = idp

syntax sym p =  p
```

To work with identity types, the induction principle is the J-eliminator.

Paulin-Mohring J rule

**** Path-induction v1 ****

```
J
  : ∀ {ℓ : Level} {A : Type ℓ} {a : A} {ℓ₂ : Level}
  → (B : (a' : A) (p : a == a') → Type ℓ₂)
  → (B a idp)
  ----------------------------------------
  → ({a' : A} (p : a == a') → B a' p)

J _ b idp = b
```

1.4 Other custom types

1.4.1 Implications

```
data
  _⇒_ {ℓ₁ ℓ₂ : Level}
    (A : Type ℓ₁) (B : Type ℓ₂)
    ---------------------------
    : Type (ℓ₁ ⊔ ℓ₂)
  where
  fun : (A → B) → A ⇒ B
```

1.4.2 Bi-implications

```
_⇔_
  : ∀ {ℓ₁ ℓ₂}
  → Type ℓ₁ → Type ℓ₂
  -------------------
  → Type (ℓ₁ ⊔ ℓ₂)

A ⇔ B = (A → B) × (B → A)
```

More syntax:

```
_↔_ = _⇔_

infix 30 _↔_ _⇔_
```

1.4.3 Decidable type

```
data
  Dec {ℓ : Level}(P : Type ℓ)
    : Type ℓ
  where
  yes : ( p : P) → Dec P
  no  : (¬p : P → ⊥ ℓ) → Dec P
```

```
⌊_⌋ : ∀ {ℓ : Level}{P : Type ℓ} → Dec P → ⊭ ℓ
⌊ yes _ ⌋ = 𝟙₂
⌊ no  _ ⌋ = 𝟘₂
```

```
Decidable
  : ∀ {ℓ₁ ℓ₂ ℓ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (A → B → Type ℓ)
  → Type (ℓ₁ ⊔ ℓ₂ ⊔ ℓ)

Decidable _∼_ = ∀ x y → Dec (x ∼ y)
```

1.4.4 Heterogeneous equality

```
data
  ≡≡ {ℓ : Level} (A : Type ℓ)
    : (B : Type ℓ)
    → (α : A == B) (a : A) (b : B)
    → Type (lsuc ℓ)
  where
  idp : {a : A} → ≡≡ A A idp a a
```

```
{-# OPTIONS --without-K --exact-split #-}
open import BasicTypes public
```

1.5 Basic functions

1.5.1 Path functions

Composition of paths

```
_·_
  : ∀ {ℓ : Level} {A : Type ℓ}  {x y z : A}
  → (p : x == y)
  → (q : y == z)
  --------------
  → x == z

_·_ idp q = q

infixl 50 _·_
```

Inverse of paths

```
_⁻¹
  : ∀ {ℓ : Level} {A : Type ℓ}  {a b : A}
  → a == b
  --------
  → b == a

idp ⁻¹ = idp
```

More syntax: for inverse path

```
inv = _⁻¹
!_  = inv

infixl 60 _⁻¹ !_
```

1.5.2 Identity functions

The identity function with implicit type.

```
id
  : ∀ {ℓ : Level} {A : Type ℓ}
  ----------------
  → A → A

id = λ x → x

identity = id
```

The identity function on a type `A` is `idf A`.

```
id-on
  : ∀ {ℓ : Level} (A : Type ℓ)
  ---------------
  → (A → A)

id-on A = λ x → x
```

1.5.3 Constant functions

Constant function at some point `b` is `cst b`

```
constant-function
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (b : B)
  ---------
  → (A → B)

constant-function b = λ _ → b
```

1.5.4 Reasoning with negation

```
¬¬ ¬¬¬
  : ∀ {ℓ : Level}
  → Type ℓ
  → Type ℓ

¬¬ A  = ¬(¬ A)
¬¬¬ A = ¬(¬¬ A)
```

```
neg¬
  : Bool
  → Bool

neg¬ tt = ff
```

```
contrapositive
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}{A : Type ℓ₁}{B : Type ℓ₂}
  → (A → B)
  → ((B → ⊥ ℓ₃) → (A → ⊥ ℓ₃))

contrapositive f v a = v (f a)
```

1.5.5 Composition

A more sophisticated composition function that can handle dependent functions.

```
_∘_
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂} {C : (a : A) → (B a → Type ℓ₃)}
  → (g : {a : A} → ∏ (B a) (C a))
  → (f : ∏ A B)
  ------------------------------
  → ∏ A (λ a → C a (f a))

g ∘ f = λ x → g (f x)

infixr 80 _∘_
```

Synonym for composition (diagrammatic version)

```
_:>_
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂} {C : (a : A) → (B a → Type ℓ₃)}
  → (f : Π A B)
  → (g : {a : A} → Π (B a) (C a))
  ------------------------------
  → Π A (λ a → C a (f a))

f :> g = g ∘ f

_;_ = _:>_
```

(continues on next page)

(continued from previous page)

```
infixr 90 _:>_
infixr 90 _;_
```

```
domain
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{B : Type ℓ₂}
  → (A → B)
  → Type ℓ₁

domain {A = A} _ = A
```

```
codomain
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{B : Type ℓ₂}
  → (A → B)
  → Type ℓ₂

codomain {B = B} _ = B
```

```
type-of
  : ∀ {ℓ : Level}{X : Type ℓ}
  → X → Type ℓ

type-of {X = X} _ = X
```

```
level-of
  : ∀ {ℓ : Level}
  → (A : Type ℓ)
  → Level

level-of {ℓ} A = ℓ
```

Associativity of composition

- Left associativity

```
∘-lassoc
  : ∀ {ℓ : Level} {A B C D : Type ℓ}
  → (h : C → D) → (g : B → C) → (f : A → B)
  -----------------------------------------
  → (h ∘ (g ∘ f)) == ((h ∘ g) ∘ f)

∘-lassoc h g f = idp {a = (λ x → h (g (f x)))}
```

- Right associativity

```
∘-rassoc
  : ∀ {ℓ : Level} {A B C D : Type ℓ}
  → (h : C → D) → (g : B → C) → (f : A → B)
  -----------------------------------------
  → ((h ∘ g) ∘ f) == (h ∘ (g ∘ f))

∘-rassoc h g f = sym (∘-lassoc h g f)
```

When using diagramatic composition we use the equivalent lemmas to the above ones:

- Left associativity of (:>)

```
:>-lassoc
  : ∀ {ℓ : Level} {A B C D : Type ℓ}
  → (f : A → B) → (g : B → C) → (h : C → D)
  -----------------------------------------
  → (f :> (g :> h)) == ((f :> g) :> h)

:>-lassoc f g h = idp
```

- Right associativity of (:>)

```
:>-rassoc
  : ∀ {ℓ : Level} {A B C D : Type ℓ}
  → (f : A → B) → (g : B → C) → (h : C → D)
  -----------------------------------------
  → ((f :> g) :>  h) == (f :> (g :> h))

:>-rassoc f g h = sym (:>-lassoc f g h)
```

1.5.6 Application

```
_←_ : ∀ {ℓ₁ ℓ₂ : Level} (A : Type ℓ₁) (B : Type ℓ₂) → Type (ℓ₁ ⊔ ℓ₂)
B ← A = A → B
```

```
_$_
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}
  → (∀ (x : A) → B x)
  -------------
  → (∀ (x : A) → B x)

f $ x = f x

infixr 0 _$_
```

1.5.7 Natural number operations

```
plus : ℕ → ℕ → ℕ
plus zero     y = y
plus (succ x) y = succ (plus x y)
```

```
infixl 60 _+ₙ_
_+ₙ_ : ℕ → ℕ → ℕ
_+ₙ_ = plus
```

```
max : ℕ → ℕ → ℕ
max 0        n = n
max (succ n) 0 = succ n
max (succ n) (succ m) = succ (max n m)
```

```
min : ℕ → ℕ → ℕ
min 0        n = 0
min (succ n) 0 = 0
min (succ n) (succ m) = succ (min n m)
```

Now, we prove some lemmas about natural number addition are the following. Notice in the proofs, the extensive usage of rewriting, because at this point we have not showed that the equality type is a congruent relation.

```
plus-lunit
  :  (n : ℕ)
  ----------------
  → zero +ₙ n == n

plus-lunit n = refl n
```

```
plus-runit
  : (n : ℕ)
  ----------------
  → n +ₙ zero == n

plus-runit zero     = refl zero
plus-runit (succ n) rewrite (plus-runit n) = idp
```

```
plus-succ
  :  (n m : ℕ)
  ---------------------------------
  → succ (n +ₙ m) == (n +ₙ (succ m))

plus-succ zero     m = refl (succ m)
plus-succ (succ n) m rewrite (plus-succ n m) = idp
```

```
plus-succ-rs
  : (n m o p : ℕ)
  →         n +ₙ m == o +ₙ p
  -------------------------------
  → n +ₙ (succ m) == o +ₙ (succ p)

plus-succ-rs 0 m 0 p α rewrite α = idp
plus-succ-rs 0 m (succ o) p α rewrite α | plus-succ o p = idp
plus-succ-rs (succ n) m 0 p α rewrite α | ! (plus-succ n m) | α = idp
plus-succ-rs (succ n) m (succ o) p α rewrite ! α | ! (plus-succ n m) | plus-succ o p | α = idp
-- other solution, just : ! (plus-succ n m) · ap succ α · (plus-succ o p)
```

Commutativity

```
plus-comm
  : (n m : ℕ)
  -----------------
  → n +ₙ m == m +ₙ n

plus-comm zero     m = inv (plus-runit m)
plus-comm (succ n) m rewrite (plus-comm n m)  = plus-succ m n
```

Associativity

```
plus-assoc
  : (n m p : ℕ)
  --------------------------------
  → n +ₙ (m +ₙ p) == (n +ₙ m) +ₙ p

plus-assoc zero     m p = refl (m +ₙ p)
plus-assoc (succ n) m p  rewrite (plus-assoc n m p) = idp
```

1.5.8 Coproduct manipulation

Functions handy to manipulate coproducts:

```
+-map
  : ∀ {i j k l : Level} {A : Type i} {B : Type j} {A' : Type k} {B' : Type l}
  → (A → A')
  → (B → B')
  → A + B → A' + B'

+-map f g = cases (f :> inl) (g :> inr)

syntax +-map f g = ⟨ f ⊕ g ⟩
```

```
parallell
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂} {C : (a : A) → (B a → Type ℓ₃)}
  → (f : (a : A) → B a)
  → ((a : A) → C a (f a))
  -------------------------
  → (a : A) → ∑ (B a) (C a)

parallell f g a = (f a , g a)
```

```
syntax parallell f g = ⟨ f × g ⟩
```

1.5.9 Curryfication

```
curry
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}  {C : Σ A B → Type ℓ₃}
  → ((s : ∑ A B) → C s)
  -------------------------------
  → ((x : A)(y : B x) → C (x , y))

curry f x y = f (x , y)
```

1.5.10 Uncurryfication

```
unCurry
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂} {C : Type ℓ₃}
  → (D : (a : A) → B a → C)
  ------------------------
  → (p : ∑ A B) → C

unCurry D p = D (proj₁ p) (proj₂ p)
```

```
uncurry
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂} {C : (a : A) → (B a → Type ℓ₃)}
  → (f : (a : A) (b : B a) → C a b)
  ---------------------------------
  → (p : ∑ A B) → C (π₁ p) (π₂ p)

uncurry f (x , y) = f x y
```

1.5.11 Finite iteration of a function

For any endo-function in A, $f : A \to A$, the following function iterates n times f

$$f^{n+1}(x) = f(f^n(x))$$

```
infixl 50 _^_
_^_
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (f : A → A) → (n : ℕ)
  -----------------------
  → (A → A)

f ^ 0 = id
f ^ succ n = λ x → f ((f ^ n) x)
```

```
app-comm
  : ∀ {ℓ : Level}{A : Type ℓ}
  → (f : A → A) → (n : ℕ)
  → (x : A)
  --------------------------------
  → (f ((f ^ n) x) ≡ ((f ^ n) (f x)))

app-comm f 0 x = idp
app-comm f (succ n) x rewrite app-comm f n x = idp
```

```
app-comm₂
  : ∀ {ℓ : Level}{A : Type ℓ}
  → (f : A → A)
  → (n k : ℕ)
  → (x : A)
  -----------------------------------------
  → ((f ^ (n +ₙ k)) x) ≡ (f ^ n) ((f ^ k) x)

app-comm₂ f 0 0 x = idp
app-comm₂ f 0 (succ k) x = idp
app-comm₂ f (succ n) 0 x rewrite plus-runit n  = idp
app-comm₂ f (succ n) (succ k) x rewrite app-comm₂ f n (succ k) x = idp
```

```
postulate
  app-comm₃
    : ∀ {ℓ : Level}{A : Type ℓ}
    → (f : A → A)
    → (k n : ℕ)
    → (x : A)
    -----------------------------------------
    → (f ^ k) ((f ^ n) x) ≡ (f ^ n) ((f ^ k) x)

-- app-comm₃ f 0 0 x = idp
-- app-comm₃ f 0 (succ k) x = idp
-- app-comm₃ f (succ n) 0 x rewrite plus-runit n  = idp
-- app-comm₃ f (succ n) (succ k) x rewrite app-comm₃ f n (succ k) x = {!idp!}
```

1.5.12 Coproducts functions

```
inr-is-injective
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{B : Type ℓ₂} {b₁ b₂ : B}
  → inr {A = A}{B} b₁ ≡ inr b₂
  ---------------------------
  → b₁ ≡ b₂

inr-is-injective idp = idp
```

```
inl-is-injective
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{B : Type ℓ₂} {a₁ a₂ : A}
```

(continues on next page)

(continued from previous page)

```
  → inl {A = A}{B} a₁ ≡ inl a₂
  ---------------------------
  → a₁ ≡ a₂

inl-is-injective idp = idp
```

1.5.13 Equational reasoning

Equational reasoning is a way to write readable chains of equalities like in the following proof.

```
t : a ≡ e
t =
  begin
    a ≡⟨ p ⟩
    b ≡⟨ q ⟩
    c ≡⟨ r ⟩
    d ≡⟨ s ⟩
    e
  ∎
```

where `p` is a path from `a` to `b`, `q` is a path from `b` to `c`, and so on.

```
module
  EquationalReasoning {ℓ : Level} {A : Type ℓ}
  where
```

Definitional equalness.

```
_==⟨⟩_
  : ∀ (x {y} : A)
  ----------------
  → x == y → x == y

_ ==⟨⟩ p = p

_==⟨idp⟩_  = _==⟨⟩_
_==⟨refl⟩_ = _==⟨⟩_
_≡⟨⟩_      = _==⟨⟩_

infixr 2 _==⟨⟩_ _==⟨idp⟩_ _==⟨refl⟩_ _≡⟨⟩_
```

Chain:

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

_ ==⟨ thm ⟩ q = thm · q
```

Synomyms:

```
_≡⟨_⟩_ = _==⟨_⟩_

infixr 2 _==⟨_⟩_ _≡⟨_⟩_
```

Q.E.D:

```
_■
  : (x : A)
  → x == x

_■ = λ x → idp

infix 3 _■
```

The begining of a proof:

```
begin_
  : {x y : A}
  → x == y
  → x == y

begin_ p = p

infix 1 begin_
```

```
open EquationalReasoning public
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import ProductIdentities
open import EquivalenceType
open import HomotopyType
```

1.6 Decidable equality

A type has decidable equality if any two of its elements are equal or different. This would be a particular instance of the Law of Excluded Middle that holds even if we do not assume Excluded Middle.

```
module DecidableEquality where
```

```
decEq
  : ∀ {ℓ : Level} → (A : Type ℓ) → Type ℓ

decEq A = (a b : A) → (a == b) + ¬ (a == b)
```

and a more convenient name for this:

```
_is-decidable
  : ∀ {ℓ : Level} → (A : Type ℓ) → Type ℓ
A is-decidable = decEq A
```

The product of types with decidable equality is a type with decidable equality.

```
decEqProd
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → decEq A → decEq B
  -------------------
  → decEq (A × B)

decEqProd da db (a1 , b1) (a2 , b2)
 with (da a1 a2) | (db b1 b2)
... | inl aeq | inl beq = inl (prodByComponents (aeq , beq))
... | inl _   | inr bnq = inr λ b → bnq (ap π₂ b)
... | inr anq | _       = inr λ b → anq (ap π₁ b)
```

This surely can be extend to other types.

```
{-# OPTIONS --without-K --exact-split --rewriting #-}
open import BasicTypes public
open import BasicFunctions public
```

1.7 Algebra of paths

```
ap
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B) {a₁ a₂ : A}
  → a₁ == a₂
  --------------
  → f a₁ == f a₂

ap f idp = idp
```

More syntax:

```
cong  = ap
app-≡ = ap

syntax app-≡ f p = f [[ p ]]
```

Now, we can define a convenient syntax sugar for `ap` in equational reasoning.

```
infixl 40 ap
syntax ap f p = p |in-ctx f
```

Let's suppose we have a lemma:

```
lemma : a ≡ b
lemma = _
```

used in an equational reasoning like:

```
t : a ≡ e
t = f a ≡⟨ ap f lemma ⟩
    f b
    ∎
```

Then, we can now put the lemma in front:

```
t : a == e
t = f a =⟨ lemma |in-ctx f ⟩
    f b
    ∎
```

Lastly, we can also define actions on two paths:

```
ap²
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {C : Type ℓ₃}  {a₁ a₂ : A} {b₁ b₂ : B}
  → (f : A → B → C)
  → (a₁ == a₂) → (b₁ == b₂)
  --------------------------
  → f a₁ b₁  == f a₂ b₂

ap² f idp idp = idp
```

```
ap-·
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {a b c : A}
```

(continues on next page)

(continued from previous page)

```
  → (f : A → B) → (p : a == b) → (q : b == c)
  -------------------------------------------
  → ap f (p · q) == ap f p · ap f q

ap-· f idp q = refl (ap f q)
```

```
ap-inv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}  {a b : A}
  → (f : A → B) → (p : a == b)
  ----------------------------
  → ap f (p ⁻¹) == (ap f p) ⁻¹

ap-inv f idp = idp
```

More syntax:

```
ap-! = ap-inv
```

```
ap-comp
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {C : Type ℓ₃} {a b : A}
  → (f : A → B)
  → (g : B → C)
  → (p : a == b)
  -------------------------------
  → ap g (ap f p) == ap (g ∘ f) p

ap-comp f g idp = idp
```

```
ap-id
  : ∀ {ℓ : Level} {A : Type ℓ} {a b : A}
  → (p : a == b)
  --------------
  → ap id p == p

ap-id idp = idp
```

```
ap-const
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}  {a a' : A} {b : B}
  → (p : a == a')
  -----------------------
  → ap (λ _ → b) p == idp

ap-const {b = b} idp = refl (refl b)
```

1.8 Properties on the groupoid

Some properties on the groupoid structure of equalities

```
·-runit
  : ∀ {ℓ : Level} {A : Type ℓ}  {a a' : A}
  → (p : a == a')
  --------------
  → p == p · idp

·-runit idp = idp
```

For convenience, we add the following rewriting rule.

```
open import Rewriting

postulate
  runit
    : ∀ {ℓ : Level} {A : Type ℓ}  {a a' : A}
    → {p : a == a'}
    --------------
    → p · idp ↦ p

{-# REWRITE runit #-}
```

```
·-lunit
  : ∀ {ℓ : Level} {A : Type ℓ}  {a a' : A}
  → (p : a == a')
  --------------
  → p == idp · p

·-lunit idp = idp
```

```
·-linv
  : ∀ {ℓ : Level} {A : Type ℓ}  {a a' : A}
  → (p : a == a')
  ----------------
  → ! p · p == idp

·-linv idp = idp

≡-inverse-left = ·-linv
```

```
·-rinv
  : ∀ {ℓ : Level} {A : Type ℓ}  {a a' : A}
  → (p : a == a')
  ----------------
  → p · ! p == idp

·-rinv idp = idp

≡-inverse-right  = ·-rinv
```

```
involution
  : ∀ {ℓ : Level} {A : Type ℓ}  {a a' : A}
  → (p : a == a')
  ---------------
  → ! (! p) == p

involution idp = idp
```

```
·-assoc
  : ∀ {ℓ : Level} {A : Type ℓ} {a b c d : A}
  → (p : a == b) → (q : b == c) → (r : c == d)
  --------------------------------------------
  → p · q · r == p · (q · r)

·-assoc idp q r = idp
```

```
·-cancellation
  : ∀ {ℓ : Level} {A : Type ℓ} {a : A}
  → (p : a == a) → (q : a == a)
  → p · q == p
  -----------------------------------------
```

(continues on next page)

(continued from previous page)

```
  → q == refl a

·-cancellation {a = a} p q α =
    begin
      q                ==⟨ ap (_· q) (! (·-linv p)) ⟩
      (! p · p) · q   ==⟨ (·-assoc (! p) p q) ⟩
      ! p · (p · q)   ==⟨ ap (! p ·_) α ⟩
      ! p · p          ==⟨ ·-linv p ⟩
      refl a
    ∎
```

Moving a term from one side to the other is a common task, so let's define some handy functions for that.

```
·-left-to-right-l
  : ∀ {ℓ : Level} {A : Type ℓ} {a b c : A} {p : a == b} {q : b == c} {r : a == c}
  → p · q == r
  ------------------
  →     q == ! p · r

·-left-to-right-l {a = a}{b = b}{c = c} {p} {q} {r} α =
  begin
    q
      ==⟨ ·-lunit q ⟩
    refl b · q
      ==⟨ ap (_· q) (! (·-linv p)) ⟩
    (! p · p) · q
      ==⟨ ·-assoc (! p) p q ⟩
    ! p · (p · q)
      ==⟨ ap (! p ·_) α ⟩
    ! p · r
  ∎
```

```
·-left-to-right-r
  : ∀ {ℓ : Level} {A : Type ℓ} {a b c : A} {p : a == b} {q : b == c} {r : a == c}
  → p · q == r
  -------------------
  →      p == r · ! q

·-left-to-right-r {a = a}{b = b}{c = c} {p} {q} {r} α =
  begin
    p
      ==⟨ ·-runit p ⟩
    p · refl b
      ==⟨ ap (p ·_) (! (·-rinv q)) ⟩
    p · (q · ! q)
      ==⟨ ! (·-assoc p q (! q)) ⟩
    (p · q) · ! q
      ==⟨ ap (_· ! q) α ⟩
    r · ! q
  ∎
```

```
·-right-to-left-r
  : ∀ {ℓ : Level} {A : Type ℓ} {a b c : A} {p : a == c} {q : a == b} {r : b == c}
  →       p == q · r
  -------------------
  → p · ! r == q

·-right-to-left-r {a = a}{b = b}{c = c} {p} {q} {r} α =
  begin
```

(continues on next page)

(continued from previous page)

```
    p · ! r
      ==⟨ ap (_· ! r) α ⟩
    (q · r) · ! r
      ==⟨ ·-assoc q r (! r) ⟩
    q · (r · ! r)
      ==⟨ ap (q ·_) (·-rinv r) ⟩
    q · refl b
      ==⟨ ! (·-runit q) ⟩
    q
    ■
```

```
·-right-to-left-l
  : ∀ {ℓ : Level} {A : Type ℓ} {a b c : A} {p : a == c} {q : a == b} {r : b == c}
  →        p == q · r
  ------------------
  → ! q · p == r

·-right-to-left-l {a = a}{b = b}{c = c} {p} {q} {r} α =
  begin
    ! q · p
      ==⟨ ap (! q ·_) α ⟩
    ! q · (q · r)
      ==⟨ ! (·-assoc (! q) q r) ⟩
    ! q · q · r
      ==⟨ ap (_· r) (·-linv q) ⟩
    refl b · r
      ==⟨ ! (·-lunit r) ⟩
    r
  ■
```

Finally, when we invert a path composition this is what we got.

```
!-·
  : ∀ {ℓ : Level} {A : Type ℓ} {a b : A}
  → (p : a == b)
  → (q : b == a)
  --------------------------
  → ! (p · q) == ! q · ! p

!-· idp q = ·-runit (! q)
```

```
{-# OPTIONS --without-K --exact-split --rewriting #-}
open import BasicTypes public
open import BasicFunctions public
open import AlgebraOnPaths public
open import Transport public
open import TransportLemmas public
```

1.9 Algebra of Pathovers

Some of the following lemmas are based on `HoTT-Agda/core/lib/PathOver.agda`.

```
!ᵈ
  : ∀ {ℓ₁ ℓ₂}{A : Type ℓ₁} {B : A → Type ℓ₂} {x y : A}
  → {p : x == y} {u : B x} {v : B y}
  → u == v [ B ↓ p ]
  ---------------------
  → v == u [ B ↓ (! p)]
```

(continues on next page)

(continued from previous page)

```
!ᵈ {p = idp} = !_
```

```
module _  {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : Type ℓ₂} where

  ↓-cst-in : {x y : A} {p : x == y} {u v : B}
    → u == v
    → u == v [ (λ _ → B) ↓ p ]
  ↓-cst-in {p = idp} q = q

  ↓-cst-out : {x y : A} {p : x == y} {u v : B}
    → u == v [ (λ _ → B) ↓ p ]
    → u == v
  ↓-cst-out {p = idp} q = q

module _  {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂} where
  _·2ᵈ_
    :  {f g h : Π A B}
    → {a a' : A} {p : a == a'} {q : f a == g a} {q' : f a' == g a'}
    → {r : g a == h a} {r' : g a' == h a'}
    → (q == q'           [ (λ a → f a == g a) ↓ p ])
    → (r == r'           [ (λ a → g a == h a) ↓ p ])
    ------------------------------------------------
    → (q · r == q' · r' [ (λ a → f a == h a) ↓ p ])

  _·2ᵈ_ {p = idp} idp idp = idp -- α · β
```

```
apd-·
  : (f : ∏ A B) → {a₁ a₂ a₃ : A}
  → (α : a₁ ≡ a₂) → (β : a₂ ≡ a₃)
  ------------------------------------------------------------------
  → apd f (α · β) ≡ pathover-comp {p = α} (apd f α) (apd f β)

apd-· C idp idp = idp
```

```
apd-!
  : (f : ∏ A B) → {a₁ a₂ : A}
  → (α : a₁ ≡ a₂)
  -----------------------------------------------------------------------------
  → apd f (α ⁻¹) ≡  ! (move-transport {α = α} (apd f α))

apd-! C idp = idp
```

```
  ·d-lunit
    : ∀ {x y : A} {p : x == y}
    → {u : B x} {v : B y}
    → (r : u == v [ B ↓ p ])
    ------------------------
    → pathover-comp {q = p} idp r == r
  ·d-lunit {p = idp} idp = idp

-- ◁idp : {x y : A} {p : x == y}
--       {u : B x} {v : B y} (q : u == v [ B ↓ p ])
--       → q ◁ idp == q
--   ◁idp {p = idp} idp = idp
```

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

open import BasicTypes public
```

(continues on next page)

(continued from previous page)

```
open import AlgebraOnPaths public
```

1.10 Transport

```
transport
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}
  → (C : A → Type ℓ₂) {a₁ a₂ : A}
  → (p : a₁ == a₂)
  --------------------------------
  → (C a₁ → C a₂)

transport C idp = (λ x → x)
```

More syntax:

```
tr     = transport
tr₁    = transport
transp = transport
```

1.10.1 Coercion

```
coe
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → A == B
  ---------
  → (A → B)

coe p a = transport (λ X → X) p a
```

and its inverse:

```
!coe
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → A == B
  ---------
  → (B → A)

!coe p a = transport (λ X → X) (! p) a
```

1.10.2 Pathovers

Let be `A : Type`, `a₁, a₂ : A`, `C : A → Type`, `c₁ : C a₁` and `c₂ : C a₂`. Using the same notation from {% cite hottbook %}, one of the definitions for the Pathover type is as the shorthand for the path between the transport along a path α `: a₁ = a₂` of the point `c₁ : C a₁` and the point `c₂` in the fiber `C a₂`. That is, a pathover is a term that inhabit the type `transport C` α `c₁ = c₂` also denoted by `PathOver C c₁` α `c₂`.

```
PathOver
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}
  → (B : A → Type ℓ₂) {a₁ a₂ : A}
  → (b₁ : B a₁) → (α : a₁ == a₂) → (b₂ : B a₂)
  ---------------------------------------------
  → Type ℓ₂

PathOver B b₁ α b₂ = tr B α b₁ == b₂
```

```
infix 30 PathOver

syntax PathOver B b₁ α b₂  = b₁ == b₂ [ B ↓ α ]
```

Another notation:

```
≡Over = PathOver
```

```
infix 30 ≡Over
syntax ≡Over B b α b' = b ≡ b' [ B / α ]
```

1.10.3 Compositions of Pathovers

```
infixl 80 _·d_
_·d_
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂}
  → {a₁ a₂ a₃ : A} {p : a₁ ≡ a₂} {q : a₂ ≡ a₃}
  → {b₁ : B a₁}{b₂ : B a₂} {b₃ : B a₃}
  → (b₁ ≡ b₂ [ B / p ])
  → (b₂ ≡ b₃ [ B / q ])
  ------------------------
  → b₁ ≡ b₃ [ B / (p · q) ]

_·d_ {p = idp} {q = idp} idp idp = idp -- α β = α · β
```

```
pathover-comp = _·d_
_●d_          = _·d_
```

```
tr₁-≡
  : ∀ {ℓ : Level} {A : Type ℓ} {a₀ a₁ a₂ : A}
  → (α : a₁ ≡ a₂)
  → (ϵ : a₀ ≡ a₁)
  → (δ : a₀ ≡ a₂)
  → (ϵ ≡ δ [ (λ a' → a₀ ≡ a') / α ])
  ---------------------------------
  → α ≡ ! ϵ · δ

tr₁-≡ idp .idp idp idp = idp
```

1.10.4 Transport along pathovers

```
tr₂
  :  ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂}
  → (C : (a : A) → (B a → Type ℓ₃))
  → {a₁ a₂ : A}{b₁ : B a₁}{b₂ : B a₂}
  → (p : a₁ == a₂)
  → (q : b₁ == b₂ [ B ↓ p ])
  → C a₁ b₁
  ----------------------
  → C a₂ b₂

tr₂ C idp idp = id
```

Gylterud's tr_2-commute:

```
tr₂-commute
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂}
```

(continues on next page)

(continued from previous page)

```
  → (C : (a : A) → (B a → Type ℓ₃))
  → (D : (a : A) → (B a → Type ℓ₃))
  → (f : ∀ a b → C a b → D a b)
  → ∀ {a a' b b'}
  → (p : a ≡ a')
  → (q : b ≡ b' [ B / p ])
  ---------------------------------------------------
  → ∀ c → tr₂ D p q (f a b c) ≡ f a' b' (tr₂ C p q c)

tr₂-commute C D f idp idp c = idp
```

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

open import Transport public
```

1.11 Transport Lemmas

```
lift
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {a₁ a₂ : A} {C : A → Type ℓ₂}
  → (α : a₁ == a₂)
  → (u : C a₁)
  ----------------------------
  → (a₁ , u) == (a₂ , tr C α u)

lift {a₁ = a₁} idp u = refl (a₁ , u)
```

```
transport-const
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {a₁  a₂ : A} {B : Type ℓ₂}
  → (p : a₁ == a₂)
  → (b : B)
  ----------------------
  → tr (λ _ → B) p b == b

transport-const idp b = refl b
```

::

lift₂ : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {C : A → A → Type ℓ₂} → {a₁ a₂ : A} → (α₁ : a₁ ≡ a₂) → {b₁ b₂ : A} → (α₂ : b₁ ≡ b₂) → (u : C a₁ b₁) ———————————— → Path {A = ∑[x] ∑[y] C x y}
(a₁ , b₁ , u) (a₂ , b₂ , tr₂ C α₁ (transport-const α₁ b₁ · α₂) u)
lift₂ idp idp u = idp

:: {- lift₃
: ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} → {C : A → A → Type ℓ₂} → {D : (x y : A) → C x y → Type ℓ₃} → {a₁ a₂ : A} → (α₁ : a₁ ≡ a₂) → {b₁ b₂ : A} → (α₂ : b₁ ≡ b₂) → {u : C a₁ b₁} → {v : C a₂ b₂} → ? ———————————— → Path {A = ∑[x] ∑[y] C x y}
(a₁ , b₁ , u) (a₂ , b₂ , v)
lift₃ idp idp u = ? -}

```
transport²
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{P : A → Type ℓ₂}
  → {x y : A} {p q : x ≡ y}
  → (r : p ≡ q)
  → (u : P x)
  ------------------------------
  → (tr P p u) ≡ (tr P q u)
```

(continues on next page)

(continued from previous page)

```
transport² idp u = idp
```

```
transport-inv-l
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{P : A → Type ℓ₂} {a a' : A}
  → (p : a == a')
  → (b : P a')
  --------------------------
  → tr P p (tr P (! p) b) == b

transport-inv-l idp b = idp
```

```
transport-inv-r
  :  ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{P : A → Type ℓ₂}  {a a' : A}
  → (p : a == a')
  → (b : P a)
  ------------------------------------------
  → tr P (! p) (tr P p b) == b

transport-inv-r idp _ = idp
```

More syntax:

```
tr-inverse = transport-inv-r
```

```
transport-concat-r
  : ∀ {ℓ : Level} {A : Type ℓ} {a : A} {x y : A}
  → (p : x == y)
  → (q : a == x)
  -------------------------------
  → tr (λ x → a == x) p q == q · p

transport-concat-r idp q = ·-runit q
```

```
transport-concat-l
  : ∀ {ℓ : Level} {A : Type ℓ} {a : A} {x y : A}
  → (p : x == y)
  → (q : x == a)
  --------------------------------
  → tr (λ x → x == a) p q == (! p) · q

transport-concat-l idp q = idp
```

```
move-transport
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{B : A → Type ℓ₂}
  → {a₁ a₂ : A}
  → {α : a₁ ≡ a₂}
  → {b₁ : B a₁}{b₂ : B a₂}
  → (tr B α b₁ ≡ b₂)
  ---------------------
  → (b₁ ≡ tr B (! α) b₂)

move-transport {α = idp} idp = idp
```

```
transport-concat
  : ∀ {ℓ : Level} {A : Type ℓ} {x y : A}
  → (p : x == y)
  → (q : x == x)
  -------------------------------------
```

(continues on next page)

(continued from previous page)

```
  → tr (λ x → x == x) p q == (! p · q) · p

transport-concat idp q = ·-runit q
```

```
transport-eq-fun
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f g : A → B) {x y : A}
  → (p : x == y)
  → (q : f x == g x)
  -------------------------------------------------------
  → tr (λ z → f z == g z) p q == ! (ap f p) · q · (ap g p)

transport-eq-fun f g idp q = ·-runit q
```

```
transport-comp
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {a b c : A} {P : A → Type ℓ₂}
  → (p : a == b)
  → (q : b == c)
  ---------------------------------------
  → ((tr P q) ∘ (tr P p)) == tr P (p · q)

transport-comp {P = P} idp q = refl (transport P q)
```

```
transport-comp-h
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {a b c : A} {P : A → Type ℓ₂}
  → (p : a == b)
  → (q : b == c)
  → (x : P a)
  -------------------------------------------
  → ((tr P q) ∘ (tr P p)) x == tr P (p · q) x

transport-comp-h {P = P} idp q x = refl (transport P q x)
```

```
transport-eq-fun-l
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}  {b : B}
  → (f : A → B) {x y : A}
  → (p :   x == y)            → (q : f x == b)
  -------------------------------------------
  → tr (λ z → f z == b) p q == ! (ap f p) · q

transport-eq-fun-l {b = b} f p q =
  begin
    transport (λ z → f z == b) p q   ==⟨ transport-eq-fun f (λ _ → b) p q ⟩
    ! (ap f p) · q · ap (λ _ → b) p  ==⟨ ap (! (ap f p) · q ·_) (ap-const p) ⟩
    ! (ap f p) · q · idp             ==⟨ ! (·-runit _) ⟩
    ! (ap f p) · q
  ∎
```

```
transport-eq-fun-r
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {b : B}
  → (g : A → B) {x y : A}
  → (p : x == y)
  → (q : b == g x)
  -----------------------------------------
  → tr (λ z → b == g z) p q == q · (ap g p)

transport-eq-fun-r {b = b} g p q =
  begin
    transport (λ z → b == g z) p q    ==⟨ transport-eq-fun (λ _ → b) g p q ⟩
```

(continues on next page)

(continued from previous page)

```
    ! (ap (λ _ → b) p) · q · ap g p   ==⟨ ·-assoc (! (ap (λ _ → b) p)) q (ap g p) ⟩
    ! (ap (λ _ → b) p) · (q · ap g p) ==⟨ ap (λ u → ! u · (q · ap g p)) (ap-const p) ⟩
    (q · ap g p)
  ∎
```

```
transport-inv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{P : A → Type ℓ₂} {a a' : A}
  → (p : a == a')
  → {a : P a'}
  --------------------------------------
  → tr (λ x → P x) p (tr P (! p) a) == a

transport-inv {P = P}  idp {a = a} =
  begin
    tr (λ v → P v) idp (tr P (! idp) a)
      ==⟨ idp ⟩
    tr P (! idp · idp) a
      ==⟨⟩
    tr P idp a
      ==⟨ idp ⟩
    a
  ∎
```

```
coe-inv-l
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → (p : A == B)
  → (b : B)
  --------------------------------------------
  → tr (λ v → v) p (tr (λ v → v) (! p) b) == b

coe-inv-l idp b = idp
```

```
coe-inv-r
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → (p : A == B)
  → (a : A)
  ---------------------------------------------
  → tr (λ v → v) (! p) (tr (λ v → v) p a) == a

coe-inv-r idp b = idp
```

```
transport-family
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁} {B : Type ℓ₂} {P : B → Type ℓ₃}
  → {f : A → B} → {x y : A}
  → (p : x == y)
  → (u : P (f x))
  -----------------------------------
  → tr (P ∘ f) p u == tr P (ap f p) u

transport-family idp u = idp
```

```
transport-family-id
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{P : A → Type ℓ₂}  → {x y : A}
  → (p : x == y)
  → (u : P x)
  ---------------------------------------------
  → transport (λ a → P a) p u == transport P p u

transport-family-id idp u = idp
```

```
transport-fun-coe
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → (α : A ≡ B)
  → (f : A → A)
  → (g : B → B)
  →     f == g [ (λ X → (X → X)) ↓ α ]
  ------------------------------------
  →  f :> coe α == (coe α) :> g

transport-fun-coe idp _ _ idp = idp
```

```
transport-fun
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁} {x y : X}
  → {A : X → Type ℓ₂} {B : X → Type ℓ₃}
  → (p : x ≡ y)
  → (f : A x → B x)
  ------------------------------------------
  → f ≡  ((λ a → tr B p (f (tr A (! p) a))))
      [ (λ x → A x → B x) / p ]

transport-fun idp f = idp
```

```
back-and-forth = transport-fun
```

```
transport-fun-h
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁}
  → {A : X → Type ℓ₂} {B : X → Type ℓ₃}
  → {x y : X}
  → (p : x == y) → (f : A x → B x)
  → (b : A y)
  --------------------------------
  → (tr (λ x → (A x → B x)) p f) b
  == tr B p (f (tr A (! p) b))

transport-fun-h idp f b = idp
```

More syntax:

```
back-and-forth-h = transport-fun-h
```

Now, when we transport dependent functions this is what we got:

```
transport-fun-dependent-h
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}{X : Type ℓ₁} {A : X → Type ℓ₂}
  → {B : (x : X) → (a : A x) → Type ℓ₃} {x y : X}
  → (p : x == y)
  → (f : (a : A x) → B x a)
  --------------------------------------------------------------------
  → (a' : A y)
  → (tr (λ x → (a : A x) → B x a) p f) a'
    == tr (λ w → B (π₁ w) (π₂ w)) (! lift (! p) a' ) (f (tr A (! p) a'))

transport-fun-dependent-h idp f a' = idp
```

More syntax:

```
dependent-back-and-forth-h = transport-fun-dependent-h
```

```
transport-fun-dependent
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}{X : Type ℓ₁} {A : X → Type ℓ₂}
```

(continues on next page)

(continued from previous page)

```
  → {B : (x : X) → (a : A x) → Type ℓ₃} {x y : X}
  → (p : x == y)
  → (f : (a : A x) → B x a)
  -------------------------------------------------------------------
  → (tr (λ x → (a : A x) → B x a) p f)
    == λ (a' : A y)
      → tr (λ w → B (π₁ w) (π₂ w)) (! lift (! p) a' ) (f (tr A (! p) a'))

transport-fun-dependent idp f = idp
```

More syntax:

```
dependent-back-and-forth = transport-fun-dependent
```

When using pathovers, we may need one of these identities:

```
apOver
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}{A A' : Type ℓ₁} {C : A → Type ℓ₂} {C' : A' → Type ℓ₃}
  → {a a' : A} {b : C a} {b' : C a'}
  → (f : A → A')
  → (g : {x : A} → C x → C' (f x))
  → (p : a == a')
  →      b == b' [ C ↓ p ]
  -------------------------------
  →    g b == g b' [ C' ↓ ap f p ]

apOver f g idp q = ap g q
```

1.12 Action on dependent paths

```
apd
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{P : A → Type ℓ₂}  {a a' : A}
  → (f : Π A P)
  → (p : a ≡ a')
  -------------------------
  → (f a) ≡ (f a') [ P / p ]

apd f idp = idp
```

More syntax:

```
fibre-app-≡ = apd
```

```
apd²
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{P : A → Type ℓ₂}
  → (f : Π A P)
  → {x y : A} {p q : x ≡ y}
  → (r : p ≡ q)
  --------------------------
  → apd f p  ≡ apd f q [ (λ x≡y → (f x) ≡ (f y) [ P / x≡y ]) / r ]

apd² f idp = idp
```

```
ap2d
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}  {C : Type ℓ₃}
  → (F : ∀ a → B a → C)
  → {a a' : A} {b : B a} {b' : B a'}
  → (p : a == a')
```

(continues on next page)

(continued from previous page)

```
  → (q : b == b' [ B ↓ p ])
  -------------------------
  →  F a b == F a' b'

ap2d F idp idp = idp
```

```
ap-idp
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → {a a' : A} → (p : a ≡ a')
  ----------------------------------------
  → ap f p == idp [ (λ a → f a ≡ f a') ↓ p ]

ap-idp f idp = idp
```

```
ap-idp'
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f g : A → B) → (σ : ∀ a → f a ≡ g a)
  → {a a' : A}     → (p : a' ≡ a)
  ------------------------------------------------------------
  → (! (σ a') · ap f p) · (σ a) == idp [ (\a' → g a' ≡ g a) ↓ p ]

ap-idp' f g σ {a = a} idp =
  begin
    σ a ⁻¹ · idp · σ a
      ≡⟨ ap (\p → p · σ a) (! (·-runit (σ a ⁻¹))) ⟩
     σ a ⁻¹ · σ a
      ≡⟨ ·-linv (σ a) ⟩
    idp
    ■
```

```
{-# OPTIONS --without-K --exact-split #-}
open import BasicTypes
open import BasicFunctions
open import Transport
open import TransportLemmas
```

1.13 Coproduct identities

```
module
  CoproductIdentities
  where
```

```
∑-≡
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} → (B : A → Type ℓ₂)
  → {ab ab' : ∑ A B}
  → (p : π₁ ab ≡ π₁ ab')
  → π₂ ab ≡ π₂ ab' [ B / p ]
  --------------------------
  → ab ≡ ab'

∑-≡ B idp idp = idp
```

```
π₁-≡
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} → (B : A → Type ℓ₂)
  → {ab ab' : ∑ A B}
  → ab ≡ ab'
```

(continues on next page)

(continued from previous page)

```
  ----------------
  → π₁ ab ≡ π₁ ab'

π₁-≡ B idp = idp
```

```
π₂-≡
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} → (B : A → Type ℓ₂)
  → {ab ab' : ∑ A B}
  → (p : ab ≡ ab')
  ----------------------------------------
  → (π₂ ab) ≡ (π₂ ab') [ B / (π₁-≡ B p) ]

π₂-≡ B idp = idp
```

```
∑-map
  :  ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂}
  → {A' : Type ℓ₃} {B' : A' → Type ℓ₄}
  → (f : A → A')
  → ((a : A) → B a → B' (f a))
  -----------------------------
  → ∑ A B → ∑ A' B'

∑-map f g p = (f (π₁ p) , g (π₁ p) (π₂ p))
```

```
∑-map-compose
  : ∀ {i₀ j₀ i₁ j₁ i₂ j₂}
  → {A₀ : Type i₀} {B₀ : A₀ → Type j₀}
  → {A₁ : Type i₁} {B₁ : A₁ → Type j₁}
  → {A₂ : Type i₂} {B₂ : A₂ → Type j₂}
  → (f₀ : A₀ → A₁) → (f₁ : (a : A₀) → B₀ a → B₁ (f₀ a))
  → (g₀ : A₁ → A₂) → (g₁ : (a : A₁) → B₁ a → B₂ (g₀ a))
  → (x : ∑ A₀ B₀)
  --------------------------------------------------------
  → ∑-map (g₀ ∘ f₀) (λ a b → g₁ (f₀ a) (f₁ a b)) x
  ≡ (∑-map {B' = B₂} g₀ g₁ (∑-map {B' = B₁} f₀ f₁ x))

∑-map-compose _ _ _ _ (a , b) = idp
```

```
∑-lift
  :  ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂} {C : A → Type ℓ₂}
  → (∀ a → B a → C a)
  --------------------
  → ∑ A B → ∑ A C

∑-lift f = ∑-map id f
```

Some of the following identities are a customized version of the lemmas above.

```
module Sigma {ℓᵢ ℓⱼ} {A : Type ℓᵢ} {P : A → Type ℓⱼ} where
```

Two dependent pairs are equal if they are componentwise equal.

```
Σ-componentwise
  : ∀ {v w : Σ A P}
  → v == w
  ------------------------------------------------
  → Σ (π₁ v == π₁ w) (λ p → tr P p (π₂ v) == π₂ w)

Σ-componentwise  idp = (idp , idp)
```

```
Σ-bycomponents
  : ∀ {v w : Σ A P}
  → Σ (π₁ v == π₁ w) (λ p → (π₂ v) == (π₂ w) [ P ↓ p ] )
  ----------------------------------------------
  → v == w

Σ-bycomponents (idp , idp) = idp
```

Synonym of Σ-bycomponents:

```
pair= = Σ-bycomponents
```

A trivial consequence is the following identification:

```
lift-pair=
  : ∀ {x y : A} {u : P x}
  → (p : x == y)
  -------------------------------------------------------
  → lift {A = A}{C = P} p  u == pair= (p , refl (tr P p u))

lift-pair= idp = idp
```

Uniqueness principle property for products

```
uppt : (x : Σ A P) → (π₁ x , π₂ x) == x

uppt (a , b) = idp
```

```
Σ-ap-π₁
  : ∀ {a₁ a₂ : A} {b₁ : P a₁} {b₂ : P a₂}
  → (α : a₁ == a₂)
  → (γ : b₁ == b₂ [ P ↓ α ])
  -----------------------------
  → ap π₁ (pair= (α , γ)) == α

Σ-ap-π₁ idp idp = idp
```

```
ap-π₁-pair= = Σ-ap-π₁
```

```
open Sigma public
```

```
transport-fun-dependent-bezem
  : ∀ {ℓᵢ ℓⱼ} {X : Type ℓᵢ} {A : X → Type ℓⱼ}
      {B : (x : X) → (a : A x) → Type ℓⱼ} {x y : X}
  → (p : x == y)
  → (f : (a : A x) → B x a)
  → (a' : A y)
  ---------------------------------------------------------
  → (tr (λ x → (a : A x) → B x a) p f) a'
    == tr (λ w → B (π₁ w) (π₂ w))
          (pair= (p , transport-inv p )) (f (tr A (! p) a'))

transport-fun-dependent-bezem idp f a' = idp
```

1.14 When $\sum$ and Π commute

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

(continues on next page)

(continued from previous page)

```
open import BasicTypes
open import BasicFunctions

open import EquivalenceType
open import QuasiinverseType
```

```
∑-comm
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}
  → (B : A → Type ℓ₂) (C : A → Type ℓ₃)
    -----------------------------------------
  → Σ (Σ A B) (C ∘ fst) ≃ Σ (Σ A C) (B ∘ fst)

∑-comm {A = A} B C = qinv-≃ there (back , there-back , back-there)
  where
  there : Σ (Σ A B) (C ∘ fst) → Σ (Σ A C) (B ∘ fst)
  there ((a , b) , c) = ((a , c ) , b)

  back : Σ (Σ A C) (B ∘ fst) → Σ (Σ A B) (C ∘ fst)
  back  ((a , c ) , b) = ((a , b) , c)

  there-back : ∀ acb → there (back acb) == acb
  there-back ((a , c) , b) = idp

  back-there : ∀ abc → back (there abc) == abc
  back-there ((a , b) , c) = idp

sum-commute = ∑-comm
```

```
∏-comm
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}
  → (B : A → Type ℓ₂)
  → (C : {a : A} → B a → Type ℓ₃)
  -----------------------------------------------------------
  → (Σ[ f ∶ Π A B ] Π[ x ∶ A ] (C (f x))) ≃ (Π[ x ∶ A ] Σ[ y ∶ B x ] C y)

∏-comm {A = A} B C = qinv-≃ there (back , there-back , back-there)
  where
  there : (Σ (Π A B) (\f → Π A (C ∘ f))) → (Π A (\x → Σ (B x) C))
  there (f , s) x = (f x , s x)

  back : (Π A (\x → Σ (B x) C)) → (Σ (Π A B) (\f → Π A (C ∘ f)))
  back  F = (\x → fst (F x)) , (\x → snd (F x))

  there-back : ∀ F → there (back F) == F
  there-back F = idp

  back-there : ∀ fs → back (there fs) == fs
  back-there fs = idp
```

```
prod-commute = ∏-comm
```

```
{-# OPTIONS --without-K --exact-split #-}
open import BasicTypes
```

1.15 Fibre type

```
module
  FibreType {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
    where
```

```
fibre
  : (f : A → B)
  → (b : B)
  ---------------
  → Type (ℓ₁ ⊔ ℓ₂)

fibre f b = Σ A (λ a → f a == b)
```

Synomyms and syntax sugar:

```
fib   = fibre
fiber = fibre

syntax fibre f b = f // b
```

A function applied over the fiber returns the original point

```
fib-eq
  : ∀ {f : A → B} {b : B}
  → (h : fib f b)
  ------------------
  → f (proj₁ h) == b

fib-eq (a , α) = α
```

Each point is on the fiber of its image.

```
fib-image
  :  ∀ {f : A → B} → {a : A}
  → fib f (f a)

fib-image {f} {a} = a , refl (f a)
```

1.16 Transport with fibers

```
{-# OPTIONS --without-K  --exact-split #-}
open import BasicTypes
open import Transport
open import FibreType
```

```
fibre-transport
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁} {B : Type ℓ₂}
  → (f : A → B)
  → {b b' : B} → (h : b == b')
  ----------------------------------------------
  → ∀ a e → (a , e) == (a , (e · h)) [ (fibre f) ↓ h ]

fibre-transport f idp a idp = idp
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
```

1.17 Homotopies

```
module HomotopyType
  {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {P : A → Type ℓ₂}
  where
```

> In a type-theoretical sense, a homotopy between two functions is a family of equalities between their applications.

Let $f, g : \prod_{(x:A)} P(x)$ be two sections of a type family $P : A \to \mathcal{U}$. A **homotopy** from f to g is a dependent function of type

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

1.17.1 Homotopy types

```
homotopy
  : (f g : Π A P)
  ---------------
  → Type (ℓ₁ ⊔ ℓ₂)

homotopy f g = ∀ (x : A) → f x == g x
```

Usual notation for homotopy:

```
_∼_ : (f g : ((x : A) → P x)) → Type (ℓ₁ ⊔ ℓ₂)
f ∼ g = homotopy f g
```

1.17.2 Homotopy is an equivalence relation

Reflexivity

```
h-refl
  : (f : Π A P)
  -------------
  → f ∼ f

h-refl f x = idp
```

Symmetry

```
h-sym
  : (f g : Π A P)
  → f ∼ g
  -------
  → g ∼ f

h-sym _ _ e x = ! (e x)
```

Transitivity

```
h-comp
  : {f g h : Π A P}
  → f ∼ g
  → g ∼ h
  -------
  → f ∼ h

h-comp u v x = (u x) · (v x)
```

Syntax:

```
_l_
  : {f g h : Π A P}
  → f ∼ g
  → g ∼ h
  -------
  → f ∼ h

α l β = h-comp α β
```

```
{-# OPTIONS --without-K --exact-split #-}
open import Transport
open import HomotopyType
```

1.18 Composition with homotopies

```
hl-comp
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {C : Type ℓ₃}
  → {f g : A → B}
  → {j k : B → C}
  → f ∼ g
  → j ∼ k
  -------------------
  → (j ∘ f) ∼ (k ∘ g)

hl-comp {g = g}{j = j} f-g j-k = λ x → ap j (f-g x) · j-k (g x)
```

```
rcomp-∼
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {C : Type ℓ₃}
  → (f : A → B)
  → {j k : B → C}
  → j ∼ k
  -------------------
  → (j ∘ f) ∼ (k ∘ f)

rcomp-∼ f j-k = hl-comp (h-refl f) j-k
```

```
lcomp-∼
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {C : Type ℓ₃}
  → {f g : A → B}
  → (j : B → C)
  → f ∼ g
  -------------------
  → (j ∘ f) ∼ (j ∘ g)

lcomp-∼ j α = hl-comp α (h-refl j)
```

1.19 Naturality

Homotopy is natural, meaning that it satisfies the following square commutative diagram.

```
h-naturality
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → {f g : A → B} → {x y : A}
  → (H : f ∼ g)
  → (p : x == y)
  -----------------------------
  → H x · ap g p == ap f p · H y

h-naturality {x = x} H idp = ! (·-runit (H x))
```

A particular case of naturality on the identity function.

```
h-naturality-id
  : ∀ {ℓ : Level} {A : Type ℓ}  {f : A → A} → {x : A}
  → (H : f ∼ id)
  -----------------------
  → H (f x) == ap f (H x)

h-naturality-id {f = f} {x = x} H =
  begin
    H (f x)
      ==⟨ ·-runit (H (f x)) ⟩
    H (f x) · refl (f x)
      ==⟨ ap (H (f x) ·_) (! (·-rinv (H x))) ⟩
    H (f x) · ((H x) · (! (H x)))
      ==⟨ ap (H (f x) ·_) (ap (_· (! (H x))) (! ap-id (H x))) ⟩
    H (f x) · (ap id (H x) · ! (H x))
      ==⟨ ! (·-assoc (H (f x)) (ap id (H x)) (! (H x))) ⟩
    (H (f x) · ap id (H x)) · ! (H x)
      ==⟨ ·-right-to-left-r (h-naturality H (H x)) ⟩
    ap f (H x)
  ■
```

1.20 Equivalences

There are three definitions to say a function is an equivalence. All these definitions are required to be mere propositions and to hold the bi-implication of begin *quasi-inverse*. We show this clearly in what follows. Nevertheless, we want to get the following fact:

```
{-# OPTIONS --without-K --exact-split #-}
open import BasicTypes
open import HLevelTypes
open import FibreType

open import Transport
open import HomotopyType

module EquivalenceType where
```

$$\mathsf{isContr}(f) \cong \mathsf{ishae}(f) \cong \mathsf{biinv}(f).$$

1.20.1 Contractible maps

A map is *contractible* if the fiber in any point is contractible, that is, each element has a unique preimagen.

```
isContrMap
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

isContrMap {B = B} f = (b : B) → isContr (fib f b)
```

Synomyns:

```
map-contractible = isContrMap
_is-contr-map    = isContrMap
```

There exists an equivalence between two types if there exists a contractible function between them.

```
isEquiv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

isEquiv f = isContrMap f
```

Synomyms:

```
isEquivalence   = isEquiv
_is-equivalence = isEquiv
_is-equiv       = isEquiv
```

1.20.2 Equivalence Type

```
_≃_
  : ∀ {ℓ₁ ℓ₂ : Level} (A : Type ℓ₁)(B : Type ℓ₂)
  → Type (ℓ₁ ⊔ ℓ₂)

A ≃ B = Σ (A → B) isEquiv
```

```
module EquivalenceMaps {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}  where
```

```
lemap
  : A ≃ B
  -------
  → (A → B)

lemap = π₁
```

More syntax:

```
≃-to-→ = lemap
fun≃    = lemap
_●      = lemap
_●→     = lemap
apply  = lemap

infixl 70 _● _●→
```

```
remap
  : A ≃ B
  ---------
  → (B → A)
```

(continues on next page)

(continued from previous page)

```
remap (f , contrf) b = π₁ (π₁ (contrf b))
```

```
≃-to-←  = remap
invfun≃ = remap
_•←     = remap
rapply  = remap

infixl 70 _•←
```

The maps of an equivalence are inverses in particular

```
lrmap-inverse
  : (e : A ≃ B) → {b : B}
  ------------------------
  → (e •→) ((e •←) b) == b

lrmap-inverse (f , eqf) {b} = fib-eq (π₁ (eqf b))
```

```
•→∘•← = lrmap-inverse
```

```
rlmap-inverse
  : (e : A ≃ B) → {a : A}
  ------------------------
  → (e •←) ((e •→) a) == a

rlmap-inverse (f , eqf) {a} = ap π₁ ((π₂ (eqf (f a))) fib-image)
```

```
•←∘•→ = rlmap-inverse
```

```
lrmap-inverse-h
  : (e : A ≃ B)
  ------------------------
  → ((e •→) ∘ (e •←)) ∼ id

lrmap-inverse-h e = λ x → lrmap-inverse e {x}
```

```
•→∘•←-h = lrmap-inverse-h
```

```
rlmap-inverse-h
  : (e : A ≃ B)
  ------------------------
  → ((e •←) ∘ (e •→)) ∼ id

rlmap-inverse-h e = λ x → rlmap-inverse e {x}
```

```
•←∘•→-h = rlmap-inverse-h
```

```
open EquivalenceMaps public
```

1.21 Quasiinverses

Two functions are quasi-inverses if we can construct a function providing `(g ∘ f) x = x` and `(f ∘ g) y = y` for any given `x` and `y`.

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import HomotopyLemmas


open import HalfAdjointType
```

```
module QuasiinverseType {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂} where
```

```
qinv
  : (A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

qinv f = Σ (B → A) (λ g → ((f ∘ g) ∼ id) × ((g ∘ f) ∼ id))

quasiinverse = qinv
```

```
linv
  : (A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

linv f = Σ (B → A) (λ g → (g ∘ f) ∼ id-on A)

left-inverse = linv
```

```
rinv
  : (A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

rinv f = Σ (B → A) (λ g → (f ∘ g) ∼ id-on B)

right-inverse = rinv
```

Biinverse is another equivalent notion of the right equivalence for HoTT.

```
biinv : (A → B) → Type (ℓ₁ ⊔ ℓ₂)
biinv f = linv f × rinv f

biinverse  = biinv
bi-inverse = biinv
```

A desire consequence (qinv → biinv):

```
qinv-biinv : (f : A → B) → qinv f → biinv f
qinv-biinv f (g , (u1 , u2)) = (g , u2) , (g , u1)
```

```
biinv-qinv
  : (f : A → B)
  → biinv f → qinv f

biinv-qinv f ((h , α) , (g , β)) = g , (β , δ)
  where
    γ1 : g ∼ ((h ∘ f) ∘ g)
    γ1 = rcomp-∼ g (h-sym (h ∘ f) id α)

    γ2 : ((h ∘ f) ∘ g) ∼ (h ∘ (f ∘ g))
    γ2 x = idp
```

(continues on next page)

(continued from previous page)

```
    γ : g ∼ h
    γ = γ1 · (γ2 · (lcomp-∼ h β))

    δ : (g ∘ f) ∼ id
    δ = (rcomp-∼ f γ) · α
```

```
equiv-biinv
  : (f : A → B)
  → isContrMap f
  --------------
  → biinv f

equiv-biinv f contrf =
  (remap eq , rlmap-inverse-h eq) , (remap eq , lrmap-inverse-h eq)
  where
    eq : A ≃ B
    eq = f , contrf
```

```
qinv-ishae
  : {f : A → B}
  → qinv f
  ---------
  → ishae f

qinv-ishae {f} (g , (ϵ , η)) = record {
    g = g ;
    η = η ;
    ϵ = λ b → inv (ϵ (f (g b))) · ap f (η (g b)) · ϵ b ;
    τ = τ
  }
  where
    aux-lemma : (a : A) → ap f (η (g (f a))) · ϵ (f a) == ϵ (f (g (f a))) · ap f (η a)
    aux-lemma a =
      begin
        ap f (η ((g ∘ f) a)) · ϵ (f a)
          ==⟨ ap (λ u → ap f u · ϵ (f a)) (h-naturality-id η) ⟩
        ap f (ap (g ∘ f) (η a)) · ϵ (f a)
          ==⟨ ap (_· ϵ (f a)) (ap-comp (g ∘ f) f (η a)) ⟩
        ap (f ∘ (g ∘ f)) (η a) · ϵ (f a)
          ==⟨ inv (h-naturality (λ x → ϵ (f x)) (η a)) ⟩
        ϵ (f (g (f a))) · ap f (η a)
      ■

    τ : (a : A) → ap f (η a) == (inv (ϵ (f (g (f a)))) · ap f (η (g (f a))) · ϵ (f a))
    τ a =
      begin
        ap f (η a)
          ==⟨ ap (_· ap f (η a)) (inv (·-linv (ϵ (f (g (f a)))))) ⟩
        inv (ϵ (f (g (f a)))) · ϵ (f (g (f a))) · ap f (η a)
          ==⟨ ·-assoc (inv (ϵ (f (g (f a))))) _ (ap f (η a)) ⟩
        inv (ϵ (f (g (f a)))) · (ϵ (f (g (f a))) · ap f (η a))
          ==⟨ ap (inv (ϵ (f (g (f a)))) ·_) (inv (aux-lemma a)) ⟩
        inv (ϵ (f (g (f a)))) · (ap f (η (g (f a))) · ϵ (f a))
          ==⟨ inv (·-assoc (inv (ϵ (f (g (f a))))) _ (ϵ (f a))) ⟩
        inv (ϵ (f (g (f a)))) · ap f (η (g (f a))) · ϵ (f a)
      ■
```

Quasiinverses create equivalences.

```
qinv-≃
  : (f : A → B)
  → qinv f
  -------------
  → A ≃ B

qinv-≃ f = ishae-≃ ∘ qinv-ishae

quasiinverse-to-≃ = qinv-≃
```

```
≃-qinv
  : A ≃ B
  ----------------
  → Σ (A → B) qinv

≃-qinv eq =
  lemap eq , (remap eq , (lrmap-inverse-h eq , rlmap-inverse-h eq))

≃-to-quasiinverse = ≃-qinv
```

Half-adjoint equivalences are quasiinverses.

```
ishae-qinv
  : {f : A → B}
  → ishae f
  ---------
  → qinv f

ishae-qinv {f} (hae g η ε τ) = g , (ε , η)
```

```
≃-ishae
  : (e : A ≃ B)
  --------------
  → ishae (e ∙→)

≃-ishae e = qinv-ishae (π₂ (≃-qinv e))
```

1.22 Quasiinverse Lemmas

Two functions are quasi-inverses if we can construct a function providing `(g ∘ f) x = x` and `(f ∘ g) y = y` for any given `x` and `y`.

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import HomotopyLemmas


open import QuasiinverseType
```

1.22.1 Equivalence composition

```
module QuasiinverseLemmas where
```

The equivalence types are indeed equivalence

```
qinv-comp
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
  → Σ (A → B) qinv
  → Σ (B → C) qinv
  ----------------
  → Σ (A → C) qinv

qinv-comp (f , (if , (ϵf , ηf))) (g , (ig , (ϵg , ηg))) = (g ∘ f) , ((if ∘ ig) ,
   ( (λ x → ap g (ϵf (ig x)) · ϵg x)
   ,  λ x → ap if (ηg (f x)) · ηf x))
```

```
qinv-inv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → Σ (A → B) qinv
  ----------------
  → Σ (B → A) qinv

qinv-inv (f , (g , (ϵ , η))) = g , (f , (η , ϵ))
```

Equivalence types are equivalence relations.

```
idEqv
  : ∀ {ℓ} {A : Type ℓ}
  -------
  → A ≃ A

idEqv = id , λ a → (a , refl a) , λ { (_ , idp) → refl (a , refl a) }
```

More syntax:

```
≃-refl = idEqv
A≃A    = idEqv
```

```
_:>≃_
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
  → A ≃ B
  → B ≃ C
  -------
  → A ≃ C

_:>≃_ {A = A} {C = C} eq-f eq-g = qinv-≃ (π₁ qcomp) (π₂ qcomp)
 where
   qcomp : Σ (A → C) qinv
   qcomp = qinv-comp (≃-qinv eq-f) (≃-qinv eq-g)
```

More syntax:

```
compEqv = _:>≃_
≃-trans = _:>≃_
```

```
≃-sym
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → A ≃ B
  -------
  → B ≃ A

≃-sym {ℓ}{_} {A} {B} eq-f = qinv-≃ (π₁ qcinv) (π₂ qcinv)
 where
   qcinv : Σ (B → A) qinv
   qcinv = qinv-inv (≃-qinv eq-f)
```

More syntax:

```
invEqv = ≃-sym
≃-flip = ≃-sym
```

1.23 Biinverses

Two functions are quasi-inverses if we can construct a function providing `(g ∘ f) x = x` and `(f ∘ g) y = y` for any given `x` and `y`.

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
```

```
module
  BiinverseEquivalenceType {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁} {B : Type ℓ₂}
   where
```

```
record
  Equivalence {ℓ₁ ℓ₂} {A : Type ℓ₁} {B : Type ℓ₂} (f : A → B)
  : Type (ℓ₁ ⊔  ℓ₂)
  where
  constructor equivalence
  field
    left-inverse   : B → A
    right-inverse  : B → A
    left-identity  : ∀ a →  left-inverse (f a) ≡ a
    right-identity : ∀ b → f (right-inverse b) ≡ b

infix 10 _≃_
```

Synonym:

```
biinv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔  ℓ₂)
biinv f = Equivalence f
```

```
isequiv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔  ℓ₂)
isequiv f = Equivalence f
```

```
record
  _≃_ {ℓ₁}{ℓ₂} (A : Type ℓ₁) (B : Type ℓ₂)
  : Type (ℓ₁ ⊔ ℓ₂)
  where
  constructor eq
  field
    apply-eq : A → B
    biinverse : Equivalence apply-eq
```

```
ide
  : ∀ {ℓ₁} {A : Type ℓ₁}
  → A ≃ A

ide = eq id (equivalence id id (λ a → idp) (λ a → idp))
```

```
≃-from-≡
  : {A : Type ℓ₁}
  → (F : A → Type ℓ₂)
  → (a b : A)
  → a ≡ b → F a ≃ F b

≃-from-≡ F a b p = tr₁ (_≃_ _ ∘ F) p ide
```

1.24 Half-adjoints

Half-adjoints are an auxiliary notion that helps us to define a suitable notion of equivalence, meaning that it is a proposition and that it captures the usual notion of equivalence.

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

open import Transport
open import TransportLemmas

open import ProductIdentities
open import CoproductIdentities

open import EquivalenceType

open import HomotopyType
open import HomotopyLemmas
open import FibreType
```

```
module
  HalfAdjointType {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
    where
```

Half-adjoint equivalence:

```
record
  ishae (f : A → B)
  : Type (ℓ₁ ⊔ ℓ₂)
  where
  constructor hae
  field
    g : B → A
    η : (g ∘ f) ∼ id
    ϵ : (f ∘ g) ∼ id
    τ : (a : A) → ap f (η a) == ϵ (f a)
```

Half adjoint equivalences give contractible fibers.

```
ishae-contr
  : (f : A → B)
  → ishae f
  -------------
  → isContrMap f

ishae-contr f (hae g η ϵ τ) y = ((g y) , (ϵ y)) , contra
  where
    lemma : (c c' : fib f y) → Σ (π₁ c == π₁ c') (λ γ → (ap f γ) · π₂ c' == π₂ c) → c == c'
    lemma c c' (p , q) = Σ-bycomponents (p , lemma2)
      where
        lemma2 : transport (λ z → f z == y) p (π₂ c) == π₂ c'
        lemma2 =
          begin
```

(continues on next page)

(continued from previous page)

```
            transport (λ z → f z == y) p (π₂ c)
              ==⟨ transport-eq-fun-l f p (π₂ c) ⟩
            inv (ap f p) · (π₂ c)
              ==⟨ ap (inv (ap f p) ·_) (inv q) ⟩
            inv (ap f p) · ((ap f p) · (π₂ c'))
              ==⟨ inv (·-assoc (inv (ap f p)) (ap f p) (π₂ c')) ⟩
            inv (ap f p) · (ap f p) · (π₂ c')
              ==⟨ ap (_· (π₂ c')) (·-linv (ap f p)) ⟩
            π₂ c'
          ■

    contra : (x : fib f y) → (g y , ϵ y) == x
    contra (x , p) = lemma (g y , ϵ y) (x , p) (γ , lemma3)
      where
        γ : g y == x
        γ = inv (ap g p) · η x

        lemma3 : (ap f γ · p) == ϵ y
        lemma3 =
          begin
            ap f γ · p
              ==⟨ ap (_· p) (ap-· f (inv (ap g p)) (η x)) ⟩
            ap f (inv (ap g p)) · ap f (η x) · p
              ==⟨ ·-assoc (ap f (inv (ap g p))) _ p ⟩
            ap f (inv (ap g p)) · (ap f (η x) · p)
              ==⟨ ap (_· (ap f (η x) · p)) (ap-inv f (ap g p)) ⟩
            inv (ap f (ap g p)) · (ap f (η x) · p)
              ==⟨ ap (λ u → inv (ap f (ap g p)) · (u · p)) (τ x) ⟩
            inv (ap f (ap g p)) · (ϵ (f x) · p)
              ==⟨ ap (λ u → inv (ap f (ap g p)) · (ϵ (f x) · u)) (inv (ap-id p)) ⟩
            inv (ap f (ap g p)) · (ϵ (f x) · ap id p)
              ==⟨ ap (inv (ap f (ap g p)) ·_) (h-naturality ϵ p) ⟩
            inv (ap f (ap g p)) · (ap (f ∘ g) p · ϵ y)
              ==⟨ ap (λ u → inv u · (ap (f ∘ g) p · ϵ y)) (ap-comp g f p) ⟩
            inv (ap (f ∘ g) p) · (ap (f ∘ g) p · ϵ y)
              ==⟨ inv (·-assoc (inv (ap (f ∘ g) p)) _ (ϵ y)) ⟩
            (inv (ap (f ∘ g) p) · ap (f ∘ g) p) · ϵ y
              ==⟨ ap (_· ϵ y) (·-linv (ap (λ z → f (g z)) p)) ⟩
            ϵ y
          ■
```

Half-adjointness implies equivalence:

```
ishae-≃
  : {f : A → B}
  → ishae f
  ---------
  → A ≃ B

ishae-≃ ishaef = _ , (ishae-contr _ ishaef)
```

```
{-# OPTIONS --without-K --exact-split #-}
-- module _ where

open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import HomotopyLemmas

open import HalfAdjointType
```

(continues on next page)

(continued from previous page)

```
open import QuasiinverseType
open import QuasiinverseLemmas
```

1.25 Equivalence reasoning

```
module EquivalenceReasoning where
```

```
_≃⟨⟩_
  : ∀ {ℓ₁ ℓ₂} (A : Type ℓ₁) {B : Type ℓ₂}
  → A ≃ B
  -------
  → A ≃ B

_ ≃⟨⟩ e = e
infixr 2 _≃⟨⟩_
```

```
_≃⟨by-def⟩_
  : ∀ {ℓ₁ ℓ₂} (A : Type ℓ₁) {B : Type ℓ₂}
  → A ≃ B
  -------
  → A ≃ B

_ ≃⟨by-def⟩ e = e
infixr 2 _≃⟨by-def⟩_
```

```
_≃⟨_⟩_
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} (A : Type ℓ₁){B : Type ℓ₂} {C : Type ℓ₃}
  → A ≃ B → B ≃ C
  ---------------
  → A ≃ C

_ ≃⟨ e₁ ⟩ e₂ = ≃-trans e₁ e₂

infixr 2 _≃⟨_⟩_
```

```
_≃■
  : ∀ {ℓ : Level} (A : Type ℓ)
  → A ≃ A

_≃■ = λ A → idEqv {A = A}
infix  3 _≃■
```

```
begin≃_
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → A ≃ B
  -------
  → A ≃ B

begin≃_ e = e
infix  1 begin≃_
```

```
postulate
 move-right-from-composition
   : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}{A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
   → (e1 : A → B) → (e2 : B ≃ C) → (e3 : A → C)
   → e1 :> (e2 ●) ≡ e3
   --------------------------------------
```

(continues on next page)

(continued from previous page)

```
  →            e1 ≡ e3 :> (e2 •←)

move-left-from-composition
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}{A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
  → (e1 : A → B) → (e2 : B ≃ C) → (e3 : A → C)
  →            e1 ≡ e3 :> (e2 •←)
  -------------------------------------
  → e1 :> (e2 •) ≡ e3

2-out-of-3-property
 : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}{A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
 → (e1 : A → C) → (e2 : A ≃ B) → (e3 : B ≃ C)
 → e1 ≡ (e2 •) :> (e3 •)
 ------------------------
 → isEquiv e1

inv-of-equiv-composition
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
  → (f : A ≃ B)
  → (g : B ≃ C)
  → remap ((f •→) :> (g •→) , π₂ (≃-trans f g)) ≡ (g •←) :> (f •←)
```

```
{-# OPTIONS --without-K --exact-split #-}
open import BasicTypes
open import BasicFunctions
open import AlgebraOnPaths
open import EquivalenceType
open import HomotopyType
open import QuasiinverseType
open import EquivalenceReasoning
```

1.26 Basic Equivalences

:: module BasicEquivalences where

There are some well-known equivalences very handy about products and coproducts.

```
≃-+-comm
  : ∀ {ℓ₁ ℓ₂ : Level} {X : Type ℓ₁}{Y : Type ℓ₂}
  → X + Y ≃ Y + X

≃-+-comm {X = X}{Y} = qinv-≃ f (g , H₁ , H₂ )
  where
  private
    f : X + Y → Y + X
    f (inl x) = inr x
    f (inr x) = inl x

    g : Y + X → X + Y
    g (inl x) = inr x
    g (inr x) = inl x

    H₁ : (f ∘ g) ∼ id
    H₁ (inl x) = idp
    H₁ (inr x) = idp

    H₂ : (g ∘ f) ∼ id
    H₂ (inl x) = idp
    H₂ (inr x) = idp
```

```
≃-+-assoc
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁}{Y : Type ℓ₂} {Z : Type ℓ₃}
  → X + (Y + Z) ≃ (X + Y) + Z

≃-+-assoc {X = X}{Y}{Z} = qinv-≃ f (g , (H₁ , H₂))
  where
  private
    f : X + (Y + Z) → (X + Y) + Z
    f (inl x) = inl (inl x)
    f (inr (inl x)) = inl (inr x)
    f (inr (inr x)) = inr x

    g : (X + Y) + Z → X + (Y + Z)
    g (inl (inl x)) = inl x
    g (inl (inr x)) = inr (inl x)
    g (inr x) = inr (inr x)

    H₁ : (f ∘ g) ∼ id
    H₁ (inl (inl x)) = idp
    H₁ (inl (inr x)) = idp
    H₁ (inr x) = idp

    H₂ : g ∘ f ∼ id
    H₂ (inl x) = idp
    H₂ (inr (inl x)) = idp
    H₂ (inr (inr x)) = idp
```

```
≃-+-runit
  : ∀ {ℓ₁ ℓ₂ : Level}{X : Type ℓ₁}
  → X ≃ X + (0 ℓ₂)

≃-+-runit {ℓ₁ = ℓ₁}{ℓ₂}{X} = qinv-≃ f (g , (H₁ , H₂ ))
  where
  private
    f : X →  X + (0 ℓ₂)
    f  x = inl x

    g : X + (0 ℓ₂) → X
    g (inl x) = x


    H₁ : (f ∘ g) ∼ id
    H₁ (inl x) = idp

    H₂ : (x : X) → (g (f x)) ≡ x
    H₂ x = idp
```

```
≃-+-lunit
  : ∀ {ℓ₁ ℓ₂ : Level}{X : Type ℓ₁}
  → X ≃ 0 ℓ₂ + X

≃-+-lunit {ℓ₂ = ℓ₂}{X} =
  X             ≃⟨ ≃-+-runit ⟩
  X + 0 ℓ₂      ≃⟨ ≃-+-comm ⟩
  (0 ℓ₂) + X    ≃∎
```

```
≃-×-comm
  : ∀ {ℓ₁ ℓ₂ : Level} {X : Type ℓ₁}{Y : Type ℓ₂}
  → X × Y ≃ Y × X

≃-×-comm {X = X}{Y} = qinv-≃ f (g , (H₁ , H₂))
```
(continues on next page)

(continued from previous page)

```
  where
  private
    f : X × Y → Y × X
    f (x , y) = (y , x)

    g : Y × X → X × Y
    g (y , x) = (x , y)

    H₁ : (f ∘ g) ∼ id
    H₁ x = idp

    H₂ : (g ∘ f) ∼ id
    H₂ x = idp
```

```
≃-×-runit
  : ∀ {ℓ₁ ℓ₂} {X : Type ℓ₁}
  → X ≃ X × (𝟙 ℓ₂)

≃-×-runit {ℓ₁}{ℓ₂}{X = X} = qinv-≃ f (g , (H₁ , H₂))
  where
  private
    f : X → X × 𝟙 ℓ₂
    f x = (x , unit)

    g : X × 𝟙 ℓ₂ → X
    g (x , _) = x

    H₁ : (f ∘ g) ∼ id
    H₁ x = idp

    H₂ : (g ∘ f) ∼ id
    H₂ x = idp
```

```
≃-×-lunit
  : ∀ {ℓ₁ ℓ₂ : Level} {X : Type ℓ₁}
  → X ≃ 𝟙 ℓ₂ × X

≃-×-lunit {ℓ₁}{ℓ₂} {X = X} =
  X            ≃⟨ ≃-×-runit ⟩
  X × (𝟙 ℓ₂)    ≃⟨ ≃-×-comm ⟩
  (𝟙 ℓ₂) × X    ≃∎
```

```
≃-×-assoc
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁}{Y : Type ℓ₂} {Z : Type ℓ₃}
  → X × (Y × Z) ≃ (X × Y) × Z

≃-×-assoc {X = X}{Y}{Z} = qinv-≃ f (g , (H₁ , H₂))
  where
  private
    f : X × (Y × Z) → (X × Y) × Z
    f (x , (y , z)) = ( (x , y) , z)

    g : (X × Y) × Z → X × (Y × Z)
    g ((x , y) , z) = (x , (y , z))

    H₁ : (f ∘ g) ∼ id
    H₁ ((x , y) , z) = idp

    H₂ : g ∘ f ∼ id
    H₂ (x , (y , z)) = idp
```

```
≃-×-+-distr
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁}{Y : Type ℓ₂} {Z : Type ℓ₃}
  → (X × (Y + Z)) ≃ ((X × Y) + (X × Z))

≃-×-+-distr {X = X}{Y}{Z} = qinv-≃ f (g , (H₁ , H₂))
  where
  private
    f : (X × (Y + Z)) → ((X × Y) + (X × Z))
    f (x , inl y) = inl (x , y)
    f (x , inr z) = inr (x , z)

    g : ((X × Y) + (X × Z)) → (X × (Y + Z))
    g (inl (x , y)) = x , inl y
    g (inr (x , z)) = x , inr z

    open import CoproductIdentities
    H₁ : (f ∘ g) ∼ id
    H₁ (inl x) = ap inl (uppt x )
    H₁ (inr x) = ap inr (uppt x)

    H₂ : (g ∘ f) ∼ id
    H₂ (p , inl x) = pair= (idp , idp)
    H₂ (p , inr x) = pair= (idp , idp)
```

A type and its lifting to some universe are equivalent as types.

```
lifting-equivalence
  : ∀ {ℓ₁ ℓ₂ : Level}
  → (A : Type ℓ₁)
  → A ≃ (↑ ℓ₂ A)

lifting-equivalence {ℓ₁}{ℓ₂} A =
  quasiinverse-to-≃ f (g , (λ { (Lift a) → idp}) , λ {p → idp})
  where
  f : A → ↑ ℓ₂ A
  f a = Lift a

  g : A ← ↑ ℓ₂ A
  g (Lift a) = a
```

Some synomys:

```
≃-↑ = lifting-equivalence
```

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

open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
```

1.27 Equivalence with Pi type

```
module PiPreserves {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{C : A → Type ℓ₂}{D : A → Type ℓ₃}
    (e : (a : A) → (C a ≃ D a)) where

  private
```

(continues on next page)

(continued from previous page)

```
    f : (a : A) → C a → D a
    f a = lemap (e a)

    f⁻¹ : (a : A) → D a → C a
    f⁻¹ a = remap (e a)

    α : (a : A) → (f a) ∘ (f⁻¹ a) ∼ id
    α a x = lrmap-inverse (e a)

    β : (a : A) → (f⁻¹ a) ∘ (f a) ∼ id
    β a x = rlmap-inverse (e a)

    ΠAC→ΠAD : Π A C → Π A D
    ΠAC→ΠAD p = λ a → (f a) (p a)

    ΠAD→ΠAC : Π A D → Π A C
    ΠAD→ΠAC p = λ a → (f⁻¹ a) (p a)

    H₁ : ΠAC→ΠAD ∘ ΠAD→ΠAC ∼ id
    H₁ p =
      begin
        (ΠAC→ΠAD ∘ ΠAD→ΠAC) p
          ==⟨ idp ⟩
        ΠAC→ΠAD (λ a → (f⁻¹ a) (p a))
          ==⟨ idp ⟩
        (λ aa → (f aa) (f⁻¹ aa (p aa)))
          ==⟨ funext (λ x → α x (p x)) ⟩
        (λ aa → p aa)
      ■

    H₂ : ΠAD→ΠAC ∘ ΠAC→ΠAD ∼ id
    H₂ p =
      begin
        (ΠAD→ΠAC ∘ ΠAC→ΠAD) p
          ==⟨ idp ⟩
        ΠAD→ΠAC (λ a → (f a) (p a))
          ==⟨ idp ⟩
        (λ aa → (f⁻¹ aa) (f aa (p aa)))
          ==⟨ funext (λ x → β x (p x)) ⟩
        (λ aa → p aa)
      ■
```

```
pi-equivalence
  : Π A C ≃ Π A D -- by (e : (a : A) → (C a ≃ D a))

pi-equivalence = qinv-≃ ΠAC→ΠAD (ΠAD→ΠAC , H₁ , H₂)
```

1.28 Equivalences preserved by Sigma types

```
{-# OPTIONS --without-K --exact-split #-}
module _ where

open import TransportLemmas
open import ProductIdentities
open import CoproductIdentities

open import EquivalenceType

open import HomotopyType
```

(continues on next page)

(continued from previous page)

```
open import HomotopyLemmas

open import HalfAdjointType
open import QuasiinverseType
open import QuasiinverseLemmas
```

```
module
  SigmaPreserves {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{C : A → Type ℓ₂}{D : A → Type ℓ₃}
    (e : (a : A) → C a ≃ D a)
  where

  private
    f : (a : A) → C a → D a
    f a = lemap (e a)

    f⁻¹ : (a : A) → D a → C a
    f⁻¹ a = remap (e a)

    α : (a : A) → (f a) ∘ (f⁻¹ a) ∼ id
    α a x = lrmap-inverse (e a)

    β : (a : A) → (f⁻¹ a) ∘ (f a) ∼ id
    β a x = rlmap-inverse (e a)

    ΣAC-to-ΣAD :  Σ A C → Σ A D
    ΣAC-to-ΣAD (a , c) = (a , (f a) c)

    ΣAD-to-ΣAC :  Σ A D → Σ A C
    ΣAD-to-ΣAC (a , d) = (a , (f⁻¹ a) d)

    H₁ : ΣAC-to-ΣAD ∘ ΣAD-to-ΣAC ∼ id
    H₁ (a , d) = pair= (idp , α a d)

    H₂ : ΣAD-to-ΣAC ∘ ΣAC-to-ΣAD ∼ id
    H₂ (a , c) = pair= (idp , β a c)
```

```
  sigma-preserves
    : Σ A C ≃ Σ A D

  sigma-preserves = qinv-≃ ΣAC-to-ΣAD (ΣAD-to-ΣAC , H₁ , H₂)

open SigmaPreserves
```

```
module SigmaPreserves-≃ {ℓ₁ ℓ₂ ℓ₃}
  {A : Type ℓ₁} {B : Type ℓ₂} (e : B ≃ A) {C : A → Type ℓ₃} where

  private
    f : B → A
    f = lemap e

    ishaef : ishae f
    ishaef = ≃-ishae e

    f⁻¹ : A → B
    f⁻¹ = ishae.g ishaef

    α : f ∘ f⁻¹ ∼ id
    α = ishae.ϵ ishaef

    β : f⁻¹ ∘ f  ∼ id
```

(continues on next page)

(continued from previous page)

```
  β = ishae.η ishaef

  τ : (b : B) → ap f (β b) == α (f b)
  τ = ishae.τ ishaef

ΣAC-to-ΣBCf : Σ A C → Σ B (λ b → C (f b))
ΣAC-to-ΣBCf (a , c) = f⁻¹ a , c'
  where
    c' : C (f (f⁻¹ a))
    c' = transport C ((α a) ⁻¹) c

ΣBCf-to-ΣAC : Σ B (λ b → C (f b)) → Σ A C
ΣBCf-to-ΣAC (b , c') = f b , c'

private
  H₁ : ΣAC-to-ΣBCf ∘ ΣBCf-to-ΣAC ∼ id
  H₁ (b , c') = pair= (β b , patho)
    where
    c'' : C (f (f⁻¹ (f b)))
    c'' = transport C ((α (f b)) ⁻¹) c'

    -- patho : c'' == c' [ (C ∘ f) ↓ (β b)]
    patho : transport (λ x → C (f x)) (β b) c'' == c'
    patho =
      begin
        transport (λ x → C (f x)) (β b) c''
          ==⟨ transport-family (β b) c'' ⟩
        transport C (ap f (β b)) c''
          ==⟨ ap (λ γ → transport C γ c'') (τ b) ⟩
        transport C (α (f b)) c''
          ==⟨ transport-comp-h ((α (f b)) ⁻¹) (α (f b)) c' ⟩
        transport C ( ((α (f b)) ⁻¹) · α (f b)) c'
          ==⟨ ap (λ γ → transport C γ c') (·-linv (α (f b))) ⟩
        transport C idp c'
          ==⟨⟩
        c'
      ∎

private
  H₂ : ΣBCf-to-ΣAC ∘ ΣAC-to-ΣBCf ∼ id
  H₂ (a , c) = pair= (α a , patho)
    where
    patho : transport C (α a) (transport C ((α a) ⁻¹) c) == c
    patho =
      begin
        transport C (α a) (transport C ((α a) ⁻¹) c)
          ==⟨ transport-comp-h (((α a) ⁻¹)) (α a) c ⟩
        transport C ( ((α a) ⁻¹) · (α a) ) c
          ==⟨ ap (λ γ → transport C γ c) (·-linv (α a)) ⟩
        transport C idp c
          ==⟨⟩
        c
      ∎
```

```
sigma-preserves-≃
  : Σ A C ≃ Σ B (λ b → C (f b))

sigma-preserves-≃ = qinv-≃ ΣAC-to-ΣBCf (ΣBCf-to-ΣAC , H₁ , H₂)
```

```
sigma-maps-≃
```

(continues on next page)

(continued from previous page)

```
  : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄} {A : Type ℓ₁} {A' : Type ℓ₄} {B : A → Type ℓ₂}{B' : A' → Type ℓ₃}
  → (α : A ≃ A')
  → ((a : A) → (B a ≃ B' ((α •) a)))
  ---------------------------------
  → Σ A B ≃ Σ A' B'

sigma-maps-≃ {A = A}{A'}{B}{B'} α β =
  ≃-trans (sigma-preserves β) (≃-sym (sigma-preserves-≃ α))
  where
    open SigmaPreserves
    open SigmaPreserves-≃
```

1.29 Sigma equivalences

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import QuasiinverseType

open import CoproductIdentities
```

```
module SigmaEquivalence where
```

```
module _ {ℓ₁ ℓ₂ : Level} (A : Type ℓ₁) (P : A → Type ℓ₂) where
```

```
pair=Equiv
  : {v w : Σ A P}
  ------------------------------------------------------------------
  → Σ (π₁ v == π₁ w) (λ p → tr (λ a → P a) p (π₂ v) == π₂ w) ≃ (v == w)

pair=Equiv = qinv-≃ Σ-bycomponents (Σ-componentwise , HΣ₁ , HΣ₂)
  where
    HΣ₁ : Σ-bycomponents ∘ Σ-componentwise ∼ id
    HΣ₁ idp = idp

    HΣ₂ : Σ-componentwise ∘ Σ-bycomponents ∼ id
    HΣ₂ (idp , idp) = idp

private
  f : {a₁ a₂ : A} {α : a₁ == a₂}{c₁ : P a₁} {c₂ : P a₂}
    → {β : a₁ == a₂}
    → {γ : transport P β c₁ == c₂}
    → ap π₁ (pair= (β , γ)) == α → β == α
  f {β = idp} {γ = idp} idp = idp

  g : {a₁ a₂ : A} {α : a₁ == a₂}{c₁ : P a₁} {c₂ : P a₂}
    → {β : a₁ == a₂}
    → {γ : transport P β c₁ == c₂}
    → β == α → ap π₁ (pair= (β , γ)) == α
  g {β = idp} {γ = idp} idp = idp

  f-g : {a₁ a₂ : A} {α : a₁ == a₂}{c₁ : P a₁} {c₂ : P a₂}
    → {β : a₁ == a₂}
    → {γ : transport P β c₁ == c₂}
    → f {α = α}{β = β}{γ} ∘ g {α = α}{β = β} ∼ id
  f-g {β = idp} {γ = idp} idp = idp
```

(continues on next page)

(continued from previous page)

```
  g-f : {a₁ a₂ : A} {α : a₁ == a₂}{c₁ : P a₁} {c₂ : P a₂}
    → {β : a₁ == a₂}
    → {γ : transport P β c₁ == c₂}
    → g {α = α}{β = β}{γ} ∘ f {α = α}{β = β}{γ} ∼ id
  g-f {β = idp} {γ = idp} idp = idp
```

```
ap-π₁-pair=Equiv
  : ∀ {a₁ a₂ : A} {c₁ : P a₁} {c₂ : P a₂}
  → (α : a₁ == a₂)
  → (γ : Σ (a₁ == a₂) (λ p → c₁ == c₂ [ P ↓ p ]))
  ------------------------------------------------
  → (ap π₁ (pair= γ) == α) ≃ (π₁ γ == α)

ap-π₁-pair=Equiv {a₁ = a₁} α (β , γ) = qinv-≃ f (g , f-g , g-f)
```

```
postulate
  ∑-≃-∑-with-≃
    : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level}{A : Type ℓ₁}{B : A → Type ℓ₂}{A' : Type ℓ₃}{B' : A' → Type ℓ₄}
     → (e : A ≃ A')
     → ∑ A B ≃ ∑ A' B'
```

```
postulate
  ∑-≃-∑-with-→
    : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄}{A : Type ℓ₁}{B : A → Type ℓ₂}{A' : Type ℓ₃}{B' : A' → Type ℓ₄}
     → (f : A → A')
     → (α : (a : A) → B a → B' (f a))
     -- Then if
     → isEquiv f
     → (a : A) → isEquiv (α a)
     ------------------------------------------
     → isEquiv (∑-map {B = B}{B' = B'} f α)
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import QuasiinverseLemmas
```

1.30 Voevodsky's Univalence Axiom

Voevodsky's Univalence axiom is postulated. It induces an equality between any two equivalent types. Some β and η rules are provided.

```
module UnivalenceAxiom {ℓ : Level} {A B : Type ℓ} where
```

```
idtoeqv
  : A == B
  --------
  → A ≃ B

idtoeqv p =
  qinv-≃
    (coe p)
```

(continues on next page)

(continued from previous page)

```
    ((!coe p) ,
      (coe-inv-l p , coe-inv-r p))
```

More syntax:

```
==-to-≃ = idtoeqv
≡-to-≃  = idtoeqv
ite     = idtoeqv
cast    = idtoeqv  -- Used in the Symmetry Book.
```

The **Univalence axiom** induces an equivalence between equalities and equivalences.

Univalence Axiom.

```
postulate
  axiomUnivalence
    : isEquivalence ≡-to-≃
```

In Slide 20 from an Escardo's talk[9], base on what we saw, we give the following no standard definition of Univalence axiom (without transport).

```
open import HLevelTypes

UA
  : ∀ {ℓ : Level}
  → (Type (lsuc ℓ))

UA {ℓ = ℓ}  = (X : Type ℓ) → isContr (∑ (Type ℓ) (λ Y → (X ≃ Y)))
```

About this Univalence axiom version:

- ∑ (Type ℓ) (λ Y → X ≃ Y) is inhabited, but we don't know if it's contractible unless, we demand (assume) to be propositional. Then, in such a case, we use the theorem (isProp P ≃ (P → isContr P)). To be more precise, we know it's contractible, in fact, the center of contractibility, is indeed (X, id-≃ X : X ≃ X).
- Univalence is a generalized extensionality axiom for intensional MLTT theory.
- The type UA is a proposition.
- UA is consistent with MLTT.
- Theorem of MLTT+UA: $P(X)$ and $X \simeq Y$ imply $P(Y)$ for any $P : \mathsf{Type} \to \mathsf{Type}$.
- Theorem of spartan MLTT with two universes. The univalence axiom formulated with crude isomorphism rather than equivalence is false!.

```
eqvUnivalence
  : (A == B) ≃ (A ≃ B)

eqvUnivalence = idtoeqv , axiomUnivalence
```

More syntax:

```
==-equiv-≃ = eqvUnivalence
==-≃-≃     = eqvUnivalence
≡-≃-≃      = eqvUnivalence
```

Introduction rule for equalities:

```
ua
  : A ≃ B
  -------
  → A == B
```

(continues on next page)

[9] https://www.newton.ac.uk/files/seminar/20170711100011001-1442677.pdf

(continued from previous page)

```
ua = remap eqvUnivalence
```

More syntax:

```
≃-to-==   = ua
eqv-to-eq = ua
```

Computation rules

```
ua-β
  : (α : A ≃ B)
  ----------------------
  → idtoeqv (ua α) == α

ua-β eqv = lrmap-inverse eqvUnivalence
```

```
ua-η
  : (p : A == B)
  ---------------------
  → ua (idtoeqv p) == p

ua-η p = rlmap-inverse eqvUnivalence
```

```
{-# OPTIONS --without-K --exact-split #-}
open import Transport
open import EquivalenceType
open import HomotopyType
```

1.31 Function extensionality

```
module FunExtAxiom {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : A → Type ℓ₂} {f g : (a : A) → B a} where
```

Application of an homotopy

```
happly
  : f == g
  ------------------------
  → ((x : A) → f x == g x)

happly idp x = refl (f x)
```

More syntax:

```
≡-app = happly
```

```
postulate
  axiomFunExt : isEquiv happly
```

```
eqFunExt
  : (f == g) ≃ ((x : A) → f x == g x)

eqFunExt = happly , axiomFunExt
```

From this, the usual notion of function extensionality follows.

```
funext
  : ((x : A) → f x == g x)
  ------------------------
  → f == g

funext = remap eqFunExt
```

Beta and eta rules for function extensionality

```
funext-β
  : (h : ((x : A) → f x == g x))
  ------------------------------
  → happly (funext h) == h

funext-β h = lrmap-inverse eqFunExt
```

```
funext-η
  : (p : f == g)
  ------------------------
  → funext (happly p) == p

funext-η p = rlmap-inverse eqFunExt
```

```
{-# OPTIONS --without-K --exact-split #-}
module _ where
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom

open import EquivalenceType
open import QuasiinverseType
open import QuasiinverseLemmas
open import UnivalenceAxiom
open import UnivalenceTransport
open import UnivalenceIdEquiv
open import HLevelLemmas
```

1.32 Univalence lemmas

```
module UnivalenceComposition where
```

```
ua-comp
  : ∀ {ℓ : Level} {A B C : Type ℓ}
  → (α : A ≃ B)
  → (β : B ≃ C)
  ---------------------------------
  → ua (α :>≃ β) == (ua α) · (ua β)

ua-comp α β  =
 begin
   ua (α :>≃ β)
     ≡⟨ ap ua (:>≃-ite-ua α β) ⟩
   ua (ite (ua α · ua β))
     ≡⟨ ua-η ((ua α) · (ua β)) ⟩
   (ua α) · (ua β)
 ∎
```

Inverses are preserved (Type-check this takes ~30min)

```
postulate
 ua-inv-r
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → (α : A ≃ B)
  -------------------------------
  → ua α · ua (≃-sym α) == refl A

{-
ua-inv-r {A = A} α =
  begin
    ua α · ua (≃-sym α)
      ==⟨ ! (ua-comp α (≃-sym α)) ⟩
    ua (≃-trans α (≃-sym α))
      ==⟨ ap ua (≃-trans-inv α) ⟩
    ua idEqv
      ==⟨ UnivalenceLemmas.ua-id ⟩
    refl A
  ■
-}
```

```
ua-inv
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → (α : A ≃ B)
  -------------------------
  → ua (≃-sym α) ≡ ! (ua α)

ua-inv α =
  begin
    ua (≃-sym α)
      ≡⟨ ap (_· ua (≃-sym α)) (! (·-linv (ua α))) ⟩
    ! (ua α) · ua α · ua (≃-sym α)
      ≡⟨ ·-assoc (! (ua α)) _ _ ⟩
    ! (ua α) · (ua α · ua (≃-sym α))
      ≡⟨ ap (! (ua α) ·_) (ua-inv-r α) ⟩
    ! (ua α) · refl _
      ≡⟨ ! (·-runit (! ((ua α)))) ⟩
    ! (ua α)
  ■
```

1.33 Univalence Lemma for Identity equivalence

```
{-# OPTIONS --without-K --exact-split #-}
module _ where
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom

open import EquivalenceType
open import QuasiinverseType
open import QuasiinverseLemmas
open import UnivalenceAxiom
open import UnivalenceTransport
open import HLevelLemmas

module UnivalenceIdEquiv where
```

The identity equivalence creates the trivial path.

```
ua-id
  : ∀ {ℓ : Level} {A : Type ℓ}
  --------------------
  → ua idEqv ≡ refl A

ua-id {A = A} =
  begin
    ua idEqv
      ≡⟨ ap ua (sameEqv (refl id)) ⟩
    ua (idtoeqv (refl A))
      ≡⟨ ua-η (refl A) ⟩
    refl A
  ∎
```

1.34 Transport and Univalence

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import QuasiinverseLemmas

open import UnivalenceAxiom

module UnivalenceTransport where
```

```
transport-family-ap
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}
  → (B : A → Type ℓ₂)
  → {x y : A}
  → (p : x == y)
  → (u : B x)
  ---------------------------------------------------
  → transport B p u == transport (λ X → X) (ap B p) u

transport-family-ap B idp u = idp
```

```
transport-family-idtoeqv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}
  → (B : A → Type ℓ₂)
  → {x y : A}
  → (p : x == y)
  → (u : B x)
  ----------------------------------------------
  → transport B p u == fun≃ (idtoeqv (ap B p)) u

transport-family-idtoeqv B idp u = idp
```

```
transport-ua
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}
  → (B : A → Type ℓ₂)
  → {x y : A}
  → (p : x == y)
  → (e : B x ≃ B y)
  → ap B p == ua e
  -----------------
```

(continues on next page)

(continued from previous page)

```
  → (u : B x) → transport B p u == (fun≃ e) u

transport-ua B idp e q u =
  begin
    transport B idp u
      ==⟨ transport-family-idtoeqv B idp u ⟩
    fun≃ (idtoeqv (ap B idp)) u
      ==⟨ ap (λ r → fun≃ (idtoeqv r) u) q ⟩
    fun≃ (idtoeqv (ua e)) u
      ==⟨ ap (λ r → fun≃ r u) (ua-β e) ⟩
    fun≃ e u
  ■
```

```
funext-transport-ua
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}
  → (B : A → Type ℓ₂)
  → {x y : A}
  → (p : x == y)
  → (e : B x ≃ B y)
  → ap B p == ua e
  -----------------
  → transport B p == (fun≃ e)

funext-transport-ua B p e x₁ = funext (transport-ua B p e x₁)
```

```
coe-ua
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → (α : A ≃ B)
  -------------------------------------
  → (∀ x → (coe (ua α) x) == ((α •) x))

coe-ua α = happly (ap (lemap) (ua-β α))
```

```
coe-ua-·
  : ∀ {ℓ : Level} {A B C : Type ℓ}
  → (α : A ≃ B)
  → (β : B ≃ C)
  --------------------------------------------------
  → coe (ua α · ua β) ≡ ((coe (ua α)) :> coe (ua β))

coe-ua-· α β =
  begin
    coe (ua α · ua β)
     ≡⟨⟩
    tr (λ X → X) (ua α · ua β)
     ≡⟨ ! (transport-comp (ua α) (ua β)) ⟩
    (tr (λ X → X) (ua α)) :> (tr (λ X → X) (ua β))
      ≡⟨ idp ⟩
   ((coe (ua α)) :> coe (ua β))
  ■
```

In addition, we can state a similar result with `idtoequiv`:

```
idtoequiv-ua-·
  : ∀ {ℓ : Level} {A B C : Type ℓ}
  → (α : A ≃ B)
  → (β : B ≃ C)
  ---------------------------------------------------
  → ite (ua α · ua β) ≡ ((ite (ua α)) :>≃ (ite (ua β)))
```

(continues on next page)

(continued from previous page)

```
idtoequiv-ua-· α β = sameEqv (coe-ua-· α β)
  where open import HLevelLemmas

ite-ua-· = idtoequiv-ua-·
```

```
postulate
 :>≃-ite-ua
  : ∀ {ℓ : Level} {A B C : Type ℓ}
  → (α : A ≃ B)     → (β : B ≃ C)
  -------------------------------
  → (α :>≃ β) ≡ ite (ua α · ua β)

{- -- below is the proof, but it blows up the time of type-checking.
  lemma α β =
      begin
        (α :>≃ β)
          ≡⟨ ap₂ (λ x y → x :>≃ y) (! (ua-β α)) (! (ua-β β)) ⟩
        (ite (ua α)) :>≃ (ite (ua β))
          ≡⟨ ! (ite-ua-· α β) ⟩
        ite (ua α · ua β)
-}
```

```
{-# OPTIONS --without-K --exact-split #-}
open import Transport
open import EquivalenceType
open import FunExtAxiom
```

1.35 Function extensionality transport case

```
module
  FunExtTransport {ℓ₁ ℓ₂ : Level}{X : Type ℓ₁} {A B : X → Type ℓ₂} {x y : X}
    where
```

```
funext-transport
  : (p : x == y) → (f : A x → B x) → (g : A y → B y)
  ------------------------------------------------------------
  → (tr (λ z₁ → (x : A z₁) → B z₁) p f == g)
  ≃ ((a : A(x)) → tr B p (f a) == g (tr A p a))

funext-transport idp f g = eqFunExt
```

```
funext-transport-l
  : (p : x == y)
  → (f : A x → B x)
  → (g : A y → B y)
  → ((tr (λ z₁ → (x : A z₁) → B z₁) p f == g)
  -------------------------------------------
  → ((a : A(x)) → tr B p (f a) == g (tr A p a)))

funext-transport-l p f g = lemap (funext-transport p _ _)
```

```
funext-transport-r
  : (p : x == y)
  → (f : A x → B x)
  → (g : A y → B y)
  → ((a : A(x)) → tr B p (f a) == g (tr A p a))
  -------------------------------------------
```

(continues on next page)

(continued from previous page)

```
  → (tr (λ z₁ → (x : A z₁) → B z₁) p f == g)

funext-transport-r p f g = remap (funext-transport p _ _)
```

1.36 Dependent function extensionality transport case

```
{-# OPTIONS --without-K --exact-split #-}
module _ where
open import Transport
open import TransportLemmas

open import CoproductIdentities

open import EquivalenceType
open import FunExtAxiom
open import FunExtTransport
open import QuasiinverseLemmas
```

```
module
  FunExtTransportDependent
  {ℓ₁ ℓ₂ : Level}{X : Type ℓ₁} {A : X → Type ℓ₂}{B : (x : X) → A x → Type ℓ₂}{x y : X}
  where
```

```
funext-transport-dfun
  : (p : x == y)
  → (f : (a : A x) → B x a)
  → (g : (a : A y) → B y a)
  ---------------------------------------------------------------------------
  →   (f == g
         [ (λ z₁ → (x : A z₁) → B z₁ x) ↓ p ])
    ≃ ((a : A x) →
      (f a) == g (tr A p a)
             [ (λ w → B (π₁ w) (π₂ w)) ↓ (pair= (p , refl (tr A p a))) ])

funext-transport-dfun idp f g = eqFunExt
```

```
funext-transport-dfun-l
  : (p : x == y) → (f : (a : A x) → B x a) → (g : (a : A y) → B y a)
 → (tr (λ z₁ → (x : A z₁) → B z₁ x) p f == g)
  ---------------------------------------------------------------------------
  → (a : A x) →
       (f a) == g (tr A p a)
              [ (λ w → B (π₁ w) (π₂ w)) ↓ (pair= (p , refl (tr A p a))) ]

funext-transport-dfun-l p f g = lemap (funext-transport-dfun p _ _)
```

```
funext-transport-dfun-r
  : (p : x == y)
  → (f : (a : A x) → B x a)
  → (g : (a : A y) → B y a)
  → ((a : A x) → tr (λ w → B (π₁ w) (π₂ w)) (pair= (p , refl (tr A p a))) (f a) == g (tr A␣
↪p a))
  --------------------------------------------------------------------------
  → (tr (λ z₁ → (x : A z₁) → B z₁ x) p f == g)

funext-transport-dfun-r p f g = remap (funext-transport-dfun p _ _)
```

```
funext-transport-dfun-bezem
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁}{A : X → Type ℓ₂}{B : (x : X) → A x → Type ℓ₃} {x y : X}
  → (p : x == y)
  → (f : (a : A x) → B x a)
  → (g : (a : A y) → B y a)
  → (a : A y)
  ----------------------------------------------------------------------------------
  → (tr (λ x → (a : A x) → B x a) p f) a == g a
  ≃  tr (λ w → B (π₁ w) (π₂ w)) (pair= (p , transport-inv p)) (f (tr A (! p)a)) == g a

funext-transport-dfun-bezem idp f g a = idEqv
```

```
funext-transport-dfun-bezem-l
  :  ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁}{A : X → Type ℓ₂}{B : (x : X) → A x → Type ℓ₃} {x y :␣
↪X}
  → (p : x == y)
  → (f : (a : A x) → B x a)
  → (g : (a : A y) → B y a)
  → (a : A y)
  → (tr (λ x → (a : A x) → B x a) p f) a == g a
  ----------------------------------------------------------------------------------
  →  tr (λ w → B (π₁ w) (π₂ w)) (pair= (p , transport-inv p)) (f (tr A (! p) a)) == g a

funext-transport-dfun-bezem-l p f g a x₁ = lemap (funext-transport-dfun-bezem p f g a) x₁
```

```
funext-transport-dfun-bezem-r
  :  ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {X : Type ℓ₁}{A : X → Type ℓ₂}{B : (x : X) → A x → Type ℓ₃} {x y :␣
↪X}
  → (p : x == y)
  → (f : (a : A x) → B x a)
  → (g : (a : A y) → B y a)
  → (a : A y)
  →  tr (λ w → B (π₁ w) (π₂ w)) (pair= (p , transport-inv p)) (f (tr A (! p) a)) == g a
  ----------------------------------------------------------------------------------
  → (tr (λ x → (a : A x) → B x a) p f) a == g a

funext-transport-dfun-bezem-r p f g a x₁ = remap (funext-transport-dfun-bezem p f g a) x₁
```

```
{-# OPTIONS --without-K --exact-split #-}
module _ where

open import BasicTypes
open import BasicFunctions
```

1.37 Hlevel types

Higher levels of the homotopical structure:

- Contractible types (-2)
- Propositions (-1)
- Sets (0)
- Groupoids (1)

1.37.1 Contractible types

A *contractible* type is a type such that **every** element is equal to a point, the *center* of contraction.

```
isContr
  : ∀ {ℓ : Level} (A : Type ℓ)
  --------------
  → Type ℓ

isContr A = Σ A (λ a → ((x : A) → a == x))
```

Synonym:

```
Contractible = isContr
is-singleton = isContr
isSingleton  = isContr
_is-contr    = isContr
```

If a type is contractible, any of its points is a center of contraction:

<img “src=”assets/images/issinglenton.png” width=“400” align=“right”>

```
contr-connects
  : ∀ {ℓ : Level} {A : Type ℓ}
  → A is-contr
  ----------------------
  → (a a' : A) → a ≡ a'

contr-connects (c₁ , f) c₂ x = ! (f c₂) · (f x)
```

1.37.2 Propositions

A type is a *mere proposition* if any two inhabitants of the type are equal.

```
isProp
  : ∀ {ℓ : Level} (A : Type ℓ) → Type ℓ

isProp A = ((x y : A) → x == y)
```

More syntax:

```
is-subsingleton = isProp
isSubsingleton  = isProp
_is-prop         = isProp
```

Let's stop a bit. So, these propositios, also named "mere" propositions tell us: in a proposition, its elements are connected with each other. Subsinglenton (which is, subset of a singlenton (a unit point set)) is empty or it has the element. Propositions can be seen as equivalent to 0 or $\mathbb{1}$.

- Contractible types $\simeq \mathbb{1}$
- Propositions $\simeq$ (0 if it's not inhabited or $\mathbb{1}$ if it's inhabited)

Therefore, we will find a theorem later that says "if A is a proposition, and it's inhabited then it's contractible", and it makes sense perfectly.

```
hProp
  : ∀ (ℓ : Level) → Type (lsuc ℓ)

hProp ℓ = ∑ (Type ℓ) isProp
```

In practice, we might want to say a type holds certain property and then we can use the convenient following predicate.

```
_has-property_
  : ∀ {ℓ : Level}
  → (A : Type ℓ)
  → (P : Type ℓ → hProp ℓ)
  → Type ℓ

A has-property P = π₁ (P A)

_holds-property = _has-property_
```

```
_has-fun-property_
  : ∀ {ℓ₁ ℓ₂ : Level} {X : Type ℓ₁}{Y : Type ℓ₂}
  → (f : X → Y)
  → (P : ∀ {X : Type ℓ₁}{Y : Type ℓ₂} → (X → Y) → hProp (ℓ₁ ⊔ ℓ₂))
  → Type (ℓ₁ ⊔ ℓ₂)

f has-fun-property P = π₁ (P f)
```

```
_has-endo-property_
  : ∀ {ℓ₁ ℓ₂ : Level} {X : Type ℓ₁}
  → (f : X → X)
  → (P : ∀ {X : Type ℓ₁} → (X → X) → hProp ℓ₂)
  → Type ℓ₂

f has-endo-property P = π₁ (P f)
```

Additionally, we may need to say, more explicity that two type share any property whenever they are equivalent. Recall, these types do not need to be in the same universe, however, for simplicity and to avoid dealing with lifting types, we'll assume they live in the same universe. Also, we require a path, instead of the equivalence because at this point, we have not define yet its type.

```
_has-all-properties-of_because-of-≡_
    : ∀ {ℓ : Level}
    → (A : Type ℓ)
    → (B : Type ℓ)
    → A ≡ B
    -------------------------------------
    → (P : Type ℓ → hProp ℓ)
    → B has-property P → A has-property P

A has-all-properties-of B because-of-≡ path
  = λ P BholdsP → tr (_has-property P) (! path) BholdsP
  where open import Transport
```

Now with (homotopy) propositional, we can consider the notion of subtype, which is, just the $\sum$-type about the collection of terms that holds some given property, we can use the following type `sub-type` for a proposition $P : A \to U$ for some type A. Assuming at least there

```
subtype
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (P : A → hProp ℓ)
  → Type ℓ

subtype P = ∑ (domain P) (π₁ ∘ P)
```

We prove now that the basic type (⊥, ⊤) are indeed (mere) propositions:

```
⊥-is-prop : ∀ {ℓ : Level} →  isProp (⊥ ℓ)
⊥-is-prop ()
```

```
⊤-is-prop : ∀ {ℓ : Level} →  isProp (⊤ ℓ)
⊤-is-prop _ _ = idp
```

More syntax:

```
𝟘-is-prop = ⊥-is-prop
𝟙-is-prop = ⊤-is-prop
```

1.37.3 Sets

A type is a *set& by definition if any two equalities on the type are equal Sets are types without any higher*dimensional structure*, all parallel paths are homotopic and the homotopy is given by a continuous function on the two paths.

```
isSet
  : ∀ {ℓ : Level} → Type ℓ → Type ℓ
isSet A = (x y : A) → isProp (x == y)
```

More syntax:

```
_is-set = isSet
```

The type of sets

```
hSet
  :  ∀ (ℓ : Level) → Type (lsuc ℓ)

hSet ℓ = ∑ (Type ℓ) isSet
```

1.37.4 Groupoids

```
isGroupoid
  : ∀ {ℓ : Level} → Type ℓ → Type ℓ

isGroupoid A  = (a₀ a₁ : A) → isSet (a₀ ≡ a₁)
```

More syntax:

```
_is-groupoid = isGroupoid
```

```
Groupoid
  : ∀ (ℓ : Level) → Type (lsuc ℓ)

Groupoid ℓ = ∑ (Type ℓ) isGroupoid
```

And, in case, we go a bit further, we have 2-groupoids. We can continue define more h-levels for all natural numbers, however, we are not going to use them.

```
is-2-Groupoid
   : ∀ {ℓ : Level} → Type ℓ → Type ℓ

is-2-Groupoid A  = (a₀ a₁ : A) → isGroupoid (a₀ ≡ a₁)

is-2-groupoid = is-2-Groupoid
```

1.38 Hedberg theorem

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType
open import HLevelTypes
open import RelationType
open import FunExtAxiom
open import FunExtTransport

module
  HedbergLemmas
  where
  module
    HedbergLemmas2
    where
```

A set is a type when it holds Axiom K.

```
axiomKisSet
  : ∀ {ℓ : Level} {A : Type ℓ}
  → ((a : A) → (p : a == a) → p == refl a)
  → isSet A

axiomKisSet k x _ p idp = k x p
```

```
reflRelIsSet
  : ∀ {ℓ : Level} (A : Type ℓ)
  → (r : Rel A)
  → ((x y : A) → R {{r}} x y → x == y)  -- xRy ⇒ Id(x,y)
  →   ((x : A) → R {{r}} x x)           -- ∀ x ⇒ xRx
  -----------------------------------
  → isSet A

reflRelIsSet A r f ρ x .x p idp = lemma p
  where
  lemma2
    : {a : A}
    → (p : a == a) → (o : R {{r}} a a)
    → (f a a o) == f a a (transport (R {{r}} a) p o)
                 [ (λ x → a == x) ↓ p ]

  lemma2 {a} p = funext-transport-l p (f a a) (f a a) (apd (f a) p)

  lemma3
    :  {a : A}
    → (p : a == a)
    → (f a a (ρ a)) · p == (f a a (ρ a))

  lemma3 {a} p = inv (transport-concat-r p _) · lemma2 p (ρ a) ·
                  ap (f a a) (Rprop {{r}} a a _ (ρ a))

  lemma
    : {a : A}
    → (p : a == a)
    → p == refl a

  lemma {a} p = ·-cancellation ((f a a (ρ a))) p (lemma3 p)
```

Lema: if a type is decidable, then ¬¬A is actually A.

```
lemDoubleNeg
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (A + ¬ A)
  ---------------
  → (¬ (¬ A) → A)

lemDoubleNeg (inl x) _ = x
lemDoubleNeg (inr f) n = exfalso (n f)
```

:: open HedbergLemmas2 public

- Hedberg's theorem. A type with decidable equality is a set.

```
hedberg
  : ∀ {ℓ : Level} {A : Type ℓ}
  → ((a b : A) → (a ≡ b) + ¬ (a ≡ b))
  ----------------------------------
  → isSet A

hedberg {ℓ}{A = A} f
  = reflRelIsSet A
      (record { R = λ a b → ¬ (¬ (a == b))
              ; Rprop = isPropNeg })
              doubleNegEq (λ a z → z (refl a))

  where
  doubleNegEq
   : (a b : A) → ¬ (¬ (a == b))
   → (a == b)

  doubleNegEq a b p = lemDoubleNeg (f a b) p

  isPropNeg
    : (a b : A)
    → isProp (¬ (¬ (a == b)))

  isPropNeg a b x y = funext λ u → exfalso (x u)
```

More syntax:

```
decidable-is-set = hedberg
```

```
{-# OPTIONS --without-K --exact-split #-}
module _ where

open import TransportLemmas
open import EquivalenceType

open import ProductIdentities
open import CoproductIdentities

open import HomotopyType
open import HomotopyLemmas

open import HalfAdjointType
open import QuasiinverseType
open import QuasiinverseLemmas

open import FunExtAxiom
open import UnivalenceAxiom
open import HLevelTypes
```

1.39 HLevel Lemmas

The following lemmas are not exactly in some coherent order. We are planning to fix that. For now, we are only adding lemmas as soon as we need them.

```
module HLevelLemmas where
```

For any type,

$$A : \mathsf{Type}$$

,

$$\mathsf{isContr}(A) \Rightarrow \mathsf{isProp}(A) \Rightarrow \mathsf{isSet}(A) \Rightarrow \mathsf{isGroupoid} A.$$

Contractible types are Propositions:

```
contrIsProp
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isContr A
  -----------
  → isProp A

contrIsProp (a , p) x y = ! (p x) · p y

-- More syntax:
isContr→isProp = contrIsProp
contr-is-prop  = contrIsProp
```

To be contractible is itself a proposition.

```
contractible-from-inhabited-prop
  : ∀ {ℓ : Level} {A : Type ℓ}
  → A
  → isProp A
  ----------------
  → Contractible A

contractible-from-inhabited-prop a p = (a , p a )
```

```
∑-contr
  : ∀ {ℓ : Level}{A : Type ℓ}
  → (x : A)
  → isContr (∑ A (λ a → a ≡ x ))

∑-contr x = (x , refl x) , λ {(a , idp) → pair= (idp , idp) }
```

Propositions are Sets:

```
propIsSet
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isProp A
  ----------
  → isSet A

propIsSet {A = A} f a _ p q = lemma p · inv (lemma q)
  where
    triang : {y z : A} {p : y == z} → (f a y) · p == f a z
    triang {y}{p = idp} = inv (·-runit (f a y))

    lemma : {y z : A} (p : y == z) → p == ! (f a y) · (f a z)
```

(continues on next page)

(continued from previous page)

```
    lemma {y = y}{w} p =
      begin
        p                        ==⟨ ap (_· p) (inv (·-linv (f a y))) ⟩
        ! (f a y) · f a y · p   ==⟨ ·-assoc (! (f a y)) (f a y) p ⟩
        ! (f a y) · (f a y · p) ==⟨ ap (! (f a y) ·_) triang ⟩
        ! (f a y) · (f a w)
      ■
```

More syntax:

```
prop-is-set  = propIsSet
prop→set     = propIsSet
isProp-isSet = propIsSet
Prop-is-Set  = propIsSet
```

Propositions are Sets:

```
Set-is-Groupoid
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isSet A
  --------------
  → isGroupoid A

Set-is-Groupoid {A} A-is-set = λ x y → prop-is-set (A-is-set x y)
```

```
is-prop-A+B
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isProp A → isProp B → ¬ (A × B)
  ---------------------------------
  → isProp (A + B)

is-prop-A+B ispropA ispropB ¬A×B (inl x) (inl x₁) = ap inl (ispropA x x₁)
is-prop-A+B ispropA ispropB ¬A×B (inl x) (inr x₁) = ⊥-elim (¬A×B (x , x₁))
is-prop-A+B ispropA ispropB ¬A×B (inr x) (inl x₁) =  ⊥-elim (¬A×B (x₁ , x))
is-prop-A+B ispropA ispropB ¬A×B (inr x) (inr x₁) = ap inr (ispropB x x₁)
```

Propositions are propositions. This time, please notice the strong use of function extensionality, used twice here.

```
propIsProp
  : ∀ {ℓ : Level} {A : Type ℓ}
  -- (funext : Function-Extensionality)
  ------------------------------------
  → isProp (isProp A)

propIsProp {_}{A} =
  λ x y → funext (λ a →
           funext (λ b
             → propIsSet x a b (x a b) (y a b)))
```

```
prop-is-prop-always = propIsProp
prop-is-prop        = propIsProp
prop→prop           = propIsProp
isProp-isProp       = propIsProp
is-prop-is-prop     = propIsProp
```

The dependent function type to proposition types is itself a proposition.

```
isProp-pi
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}
```

(continues on next page)

(continued from previous page)

```
  -- (funext : Function-Extensionality)
  → ((a : A) → isProp (B a))
  --------------------------
  → isProp ((a : A) → B a)

isProp-pi props f g = funext λ a → props a (f a) (g a)
```

```
pi-is-prop = isProp-pi
Π-isProp   = isProp-pi
piIsProp   = isProp-pi
```

Propositional extensionality, here stated as `prop-ext`, is a consequence of univalence axiom.

```
prop-ext
  : ∀ {ℓ : Level} {A B : Type ℓ}
  -- (ua : Univalence Axiom)
  → isProp A
  → isProp B
  → (A ⇔ B)
  -----------
  → A == B

prop-ext propA propB (f , g) =
  ua (qinv-≃ f (g , (λ x → propB _ _) , (λ x → propA _ _)))
```

Synomyms:

```
props-⇔-to-== = prop-ext
ispropA-B     = prop-ext
propositional-extensionality = prop-ext
```

```
setIsProp
  : ∀ {ℓ : Level} {A : Type ℓ}
  -----------------
  → isProp (isSet A)

setIsProp {ℓ} {A} p₁ p₂ =
  funext (λ x →
    funext (λ y →
      funext (λ p →
        funext (λ q → propIsSet (p₂ x y) p q (p₁ x y p q) (p₂ x y p q)))))
```

```
set-is-prop = setIsProp
set→prop    = setIsProp
```

The product of propositions is itself a proposition.

```
isProp-prod
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isProp A
  → isProp B
  ---------------------
  → isProp (A × B)

isProp-prod p q x y = prodByComponents ((p _ _) , (q _ _))
```

```
×-is-prop      = isProp-prod
ispropA×B      = isProp-prod
×-isProp       = isProp-prod
prop×prop→prop = isProp-prod
```

```
isSet-prod
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isSet A → isSet B
  -------------------
  → isSet (A × B)

isSet-prod sa sb (a , b) (c , d) p q = begin
   p
    ==⟨ inv (prodByCompInverse p) ⟩
   prodByComponents (prodComponentwise p)
    ==⟨ ap prodByComponents (prodByComponents (sa a c _ _ , sb b d _ _)) ⟩
   prodByComponents (prodComponentwise q)
    ==⟨ prodByCompInverse q ⟩
   q
  ■
```

Synomys:

```
×-is-set      = isSet-prod
isSetA×B      = isSet-prod
×-isSet       = isSet-prod
set×set→set    = isSet-prod
```

```
postulate
  ×-groupoid
    : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
    → isGroupoid A → isGroupoid B
    -------------------
    → isGroupoid (A × B)
```

```
Prop-/-≡
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (P : A → hProp ℓ)
  → ∀ {a₀ a₁} p₀ p₁ {α : a₀ ≡ a₁}
  ------------------------------
  → p₀ ≡ p₁ [ (# ∘ P) / α ]

Prop-/-≡ P {a₀} p₀ p₁ {α = idp} = proj₂ (P a₀) p₀ p₁
```

H-levels actually are preserved by products, coproducts, pi-types and sigma-types.

```
id-contractible-from-set
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isSet A
  → {a a' : A}
  --------------------------
  → a ≡ a' → isContr (a ≡ a')

id-contractible-from-set iA {a}{.a} idp
  = idp , λ q → iA a a idp q
-- This is quite obvious from the hset definition.
-- But it's nice to spell it out fully.
```

Lemma 3.11.3: For any type A, `isContr A` is a mere proposition.

```
isContrIsProp
  : ∀ {ℓ : Level} {A : Type ℓ}
  --------------------
  → isProp (isContr A)

isContrIsProp {_} {A} (a , p) (b , q) =
```

(continues on next page)

(continued from previous page)

```
  Σ-bycomponents (inv (q a) , isProp-pi (AisSet b) _ q)
    where
      AisSet : isSet A
      AisSet = propIsSet (contrIsProp (a , p))

BookLemma3113 = isContrIsProp
```

Lemma 3.3.3 (HoTT-Book):

```
lemma333
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isProp A → isProp B
  → (A → B)  → (B → A)
  ----------------------
  → A ≃ B

lemma333 iA iB f g = qinv-≃ f (g , gf , fg)
  where
  private
    fg : (f :> g) ∼ id
    fg a = iA ((f :> g) a) a

    gf : (g :> f) ∼ id
    gf b = iB ((g :> f) b) b

BookLemma333 = lemma333
```

Lemma 3.3.2 (HoTT-Book):

```
prop-inhabited-≃𝟙
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isProp A
  → (a : A)
  ---------
  → A ≃ (𝟙 ℓ)

prop-inhabited-≃𝟙 iA a =
  lemma333 iA 𝟙-is-prop (λ _ → unit) (λ _ → a)

BookLemma332 = prop-inhabited-≃𝟙
```

From Exercise 3.5 (HoTT-Book):

```
isProp-≃-isContr
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isProp A ≃ (A → isContr A)

isProp-≃-isContr {A = A} =
  lemma333 isProp-isProp (pi-is-prop (λ a → isContrIsProp)) go back
  where
    private
      go : isProp A → (A → isContr A)
      go iA a = a , λ a' → iA a a'

      back : (A → isContr A) → isProp A
      back f = λ a a' → (! π₂ (f a) a) · (π₂ (f a) a')
```

Equivalence of two types is a proposition Moreover, equivalences preserve propositions.

Contractible maps are propositions:

```
isContrMapIsProp
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  -------------
  → isProp (isContrMap f)

isContrMapIsProp f = pi-is-prop (λ a → isContrIsProp)
```

```
isEquivIsProp
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → isProp (isEquiv f)

isEquivIsProp = isContrMapIsProp
```

```
is-equiv-is-prop = isEquivIsProp
```

Equality of same-morphism equivalences

```
sameEqv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → {α β : A ≃ B}
  → π₁ α == π₁ β
  ---------------
  →     α == β

sameEqv {α = (f , σ)} {(g , τ)} p = Σ-bycomponents (p , (isEquivIsProp g _ τ))
```

```
equiv-iff-hprop
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isProp A
  → isProp B
  -----------------
  → isProp (A ≃ B)

equiv-iff-hprop {A = A}{B} iA iB ef eg
  = sameEqv f≡g
  where
  private
    f≡g : (π₁ ef) ≡ (π₁ eg)
    f≡g = pi-is-prop (λ _ → iB) (π₁ ef) (π₁ eg)
```

```
propEqvIsprop
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → isProp A
  → isProp B
  -----------------
  → isProp (A == B)

propEqvIsprop iA iB p q =
  begin
    p
      ≡⟨ ! (ua-η p) ⟩
    ua (idtoeqv p)
      ≡⟨ ap ua (equiv-iff-hprop iA iB (idtoeqv p) (idtoeqv q)) ⟩
    ua (idtoeqv q)
      ≡⟨ ua-η q ⟩
    q
  ∎
```

Equivalences preserve propositions

```
isProp-≃
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (A ≃ B)
  → isProp A
  ----------
  → isProp B

isProp-≃ eq prop x y =
  begin
    x                        ==⟨ inv (lrmap-inverse eq) ⟩
    lemap eq ((remap eq) x) ==⟨ ap (λ u → lemap eq u) (prop _ _) ⟩
    lemap eq ((remap eq) y) ==⟨ lrmap-inverse eq ⟩
    y
  ■
```

```
is-set-equiv-to-set
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → A ≃ B
  → isSet A
  ---------
  → isSet B

is-set-equiv-to-set {A = A}{B} e iA
  x y  =  isProp-≃ aux (iA (!f x) (!f y))
  where
  private
   f : A → B
   f = lemap e

   !f : B → A
   !f = remap e

   aux : (remap e x ≡ remap e y) ≃ (x ≡ y)
   aux
     = qinv-≃ (λ p → ! (lrmap-inverse e) · ap f p · lrmap-inverse e)
              ((λ { idp → idp})
              , (λ { idp → H₁})
              , λ {p → iA (!f x) (!f y) _ p})
     where
     H₁ : (! lrmap-inverse e · idp) · lrmap-inverse e {x} == idp
     H₁ = begin
       (! lrmap-inverse e · idp) · lrmap-inverse e
         ≡⟨ ap (λ w → w · (lrmap-inverse e)) (! (·-runit _)) ⟩
       ! lrmap-inverse e · lrmap-inverse e
         ≡⟨ ·-linv (lrmap-inverse e) ⟩
       idp
       ■
equiv-with-a-set-is-set = is-set-equiv-to-set
≃-with-a-set-is-set = is-set-equiv-to-set
```

Above, we might want to use univalence to rewrite $x \equiv By$. Unfortunately, we can not because a universe level matters, at least for now. A first proof would have been saying $x \equiv Ay$ is a mere proposition and since $A \simeq B$, $x \equiv By$ is also a mere proposition. So, B is a set. Second proof is construct a term of 'isSet B' by using the inverse function from the equivalence and some path algebra. Not happy with this but it works.

```
≃-trans-inv
  : ∀ {ℓ} {A B : Type ℓ}
  → (α : A ≃ B)
  -----------------------------
  → ≃-trans α (≃-flip α) ≡ A≃A
```

(continues on next page)

(continued from previous page)

```
≃-trans-inv α = sameEqv (
  begin
    π₁ (≃-trans α (≃-sym α)) ==⟨ refl _ ⟩
    π₁ (≃-sym α) ∘ π₁ α      ==⟨ funext (rlmap-inverse-h α) ⟩
    id
  ■)
```

The following lemma is telling us, something we should probably knew already: Equivalence of propositions is the same logical equivalence.

```
twoprops-to-equiv-≃-⇔
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isProp A
  → isProp B
  -------------------
  → (A ≃ B) ≃ (A ⇔ B)

twoprops-to-equiv-≃-⇔ {A = A} {B} ispropA ispropB  = qinv-≃ f (g , H₁ , H₂)
 where
  f : (A ≃ B) → (A ⇔ B)
  f e = e •→ , e •←

  g : (A ⇔ B) → (A ≃ B)
  g (h→ , h←) = qinv-≃ h→ (h← , (λ b → ispropB (h→ (h← b)) b) , (λ a → ispropA (h←␣
↪(h→ a)) a))

  H₁ : f ∘ g ∼ id
  H₁ (h→ , h←) = idp

  H₂ : g ∘ f ∼ id
  H₂ e =
    begin
      g (e •→ , e •←)
        ==⟨⟩
      e •→ , _
        ==⟨ Σ-bycomponents (idp , isEquivIsProp (e •→) _ _) ⟩
      e
    ■
```

```
∑-prop
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}
  → isProp A
  → ((a : A) → isProp (B a))
  ------------------------
  → isProp (∑ A B)

∑-prop {B = B} iA λ-iB u v
  = pair= (α , β)
  where
    α : π₁ u ≡ π₁ v
    α = iA (π₁ u) (π₁ v)

    β : (π₂ u) ≡ (π₂ v) [ B / α ]
    β = λ-iB (π₁ v) (tr B α (π₂ u)) (π₂ v)

isProp-Σ = ∑-prop
isProp-∑  = ∑-prop
Σ-prop   = ∑-prop
```

```
pi-is-set
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}
  → ((a : A) → isSet (B a))
  -------------------------
  → isSet (∏ A B)

pi-is-set  setBa f g = b
  where
  a : isProp (f ∼ g)
  a h1 h2 = funext λ x → setBa x (f x) (g x) (h1 x) (h2 x)

  b : isProp (f ≡ g)
  b = isProp-≃ (≃-sym eqFunExt) (pi-is-prop λ a → setBa a (f a) (g a))
```

```
∏-set = pi-is-set
Π-set = pi-is-set
```

The following is a custom version useful to deal with functions with implicit parameters.

```
pi-is-prop-implicit
   : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}
   → ((a : A) → isProp (B a))
   -------------------------
   → isProp ({a : A} → B a)

pi-is-prop-implicit {A = A} {B} f = isProp-≃ explicit-≃-implicit (pi-is-prop f)
  where
   explicit-≃-implicit
     : ((a : A) → B a) ≃ ({a : A} → B a)
   explicit-≃-implicit = qinv-≃ go ((λ x x₁ → x) , (λ x → idp) , (λ x → idp))
     where
       go : ((a : A) → B a) → ({a : A} → B a)
       go f {a} = f a
```

```
0-is-set
  : ∀ {ℓ} → isSet (0 ℓ)
0-is-set = prop-is-set 0-is-prop
```

```
open HLevelLemmas public
```

```
postulate
  law-excluded-middle
    : ∀ {ℓ} {P : Type ℓ}
    → isProp P
    ------------
    → P + (¬ P)

LEM = law-excluded-middle
```

and the more general propositions-as-types formulation of the law of excluded middle is:

```
postulate
 LEM∞
   : ∀ {ℓ : Level} {A : Type ℓ}
   → A + (¬ A)
```

```
law-double-negation
 : ∀ {ℓ} {P : Type ℓ}
 → isProp P
 -----------
```

(continues on next page)

(continued from previous page)

```
 → (¬ (¬ P)) → P

law-double-negation iP with LEM iP
law-double-negation iP | inl x = λ _ → x
law-double-negation iP | inr x = λ p→⊥→⊥ → ⊥-elim (p→⊥→⊥ x)
```

Law excluded middle and law of double negation are both equivalent.

Weak extensionality principle:

```
WeakExtensionalityPrinciple
  : ∀ {ℓ : Level} {A : Type ℓ}  {P : A → Type ℓ}
  → ((x : A) → isContr (P x))
  -------------------------
  → isContr ( ∏ A P )

WeakExtensionalityPrinciple {A = A}{P} f =
  (fx , λ h →  ! funext (λ x → ! (π₂ (f x)) (h x)))
  where
   fx : ∏ A P
   fx = λ x → π₁ (f x)
```

```
open import SigmaEquivalence
```

```
isSet-Σ
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : A → Type ℓ₂}
  → isSet A → ((a : A) → isSet (B a))
  -------------------
  → isSet (Σ A B)

isSet-Σ {A = A}{B} iA f x y
  = isProp-≃
    (pair=Equiv A B)
    (∑-prop (iA (π₁ x) (π₁ y))
      (λ a → f _ (tr (λ x  → B x) a (π₂ x)) (π₂ y) ))
```

```
sigma-is-set = isSet-Σ
∑-set    = isSet-Σ
isSet-∑ = isSet-Σ
```

```
≃-is-set-from-sets
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isSet A
  → isSet B
  -------------
  → isSet (A ≃ B)

≃-is-set-from-sets {A = A}{B} ia ib
  = isSet-Σ (pi-is-set  (λ _ → ib)) (λ _ → prop-is-set (isEquivIsProp _))
```

```
≡-is-set-from-sets
  : ∀ {ℓ : Level} {A B : Type ℓ}
  → isSet A
  → isSet B
  -------------
  → isSet (A ≡ B)

≡-is-set-from-sets iA iB = equiv-with-a-set-is-set (≃-sym eqvUnivalence) (≃-is-set-from-sets␣
↪iA iB)
```

```
≡-set = ≡-is-set-from-sets
```

A handy result is that the two point type is a set. We know already that 𝟙 is indeed mere propositions and hence a set. The two point type ⊭ is in fact equivalent to the type $𝟙 + 𝟙$. The fact ⊭ is a set is used later to show $A + B$ is a set when both are sets.

```
𝟙-is-set : ∀ {ℓ : Level} → isSet (𝟙 ℓ)
𝟙-is-set = prop-is-set (𝟙-is-prop)
```

```
𝟙+𝟙-is-set : ∀ {ℓ : Level} → isSet (𝟙 ℓ + 𝟙 ℓ)
𝟙+𝟙-is-set (inl ∗) (inl ∗) idp idp = idp
𝟙+𝟙-is-set (inr ∗) (inr ∗) idp idp = idp
```

```
⊭-≃-𝟙+𝟙
  : ∀ {ℓ₁ ℓ₂ : Level}
  → ⊭ ℓ₁ ≃ 𝟙 ℓ₂ + 𝟙 ℓ₂

⊭-≃-𝟙+𝟙 {ℓ₁}{ℓ₂} = quasiinverse-to-≃ f (g ,
  (λ { (inl x) → ap inl idp ; (inr x) → ap inr idp}) ,
  λ { 0₂ → idp ; 𝟙₂ → idp})
  where
    f : ⊭ ℓ₁ → 𝟙 ℓ₂ + 𝟙 ℓ₂
    f 0₂ = inl ∗
    f 𝟙₂ = inr ∗

    g : ⊭ ℓ₁ ← 𝟙 ℓ₂ + 𝟙 ℓ₂
    g (inl x) = 0₂
    g (inr x) = 𝟙₂
```

```
⊭-is-set : ∀ {ℓ : Level} → isSet (⊭ ℓ)
⊭-is-set {ℓ} = ≃-with-a-set-is-set {ℓ}{lsuc ℓ} (≃-sym (⊭-≃-𝟙+𝟙 )) 𝟙+𝟙-is-set
```

Another fact we might use later is the fact, natural numbers forms a set. We can see $\mathbb{N}$ as a type is equivalent to $\sum (n : \mathbb{N})\ 𝟙$.

The coproduct $A + B$ is equivalent to the sigma type $\sum$ ⊭ P, where P is the type family that maps 0_2 to A and consequently, 𝟙_2 maps to B.

```
P⊭-to-A+B
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}
  → (A : Type ℓ₁)(B : Type ℓ₂)
  -----------------------
  → ⊭ ℓ₃ → Type (ℓ₁ ⊔ ℓ₂)

P⊭-to-A+B A B = λ { 0₂ → ↑ (level-of B) A ; 𝟙₂ → ↑ (level-of A) B}
```

```
+-≃-∑
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → A + B ≃ ∑ (⊭ ℓ₃) (P⊭-to-A+B A B)

+-≃-∑ {ℓ₁}{ℓ₂}{ℓ₃}{A}{B} = quasiinverse-to-≃ f (g
  , (λ { (0₂ , Lift lower₁) → idp ; (𝟙₂ , Lift lower₁) → idp})
  , λ { (inl x) → idp ; (inr x) → idp})
  where
  f : A + B → ∑ (⊭ ℓ₃) (P⊭-to-A+B A B)
  f (inl x) = 0₂ , Lift x
  f (inr x) = 𝟙₂ , Lift x

  g : A + B ← ∑ (⊭ ℓ₃) (P⊭-to-A+B A B)
  g (0₂ , Lift a) = inl a
  g (𝟙₂ , Lift b) = inr b
```

```
+-of-sets-is-set
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → isSet A → isSet B
  -------------------
  → isSet (A + B)

+-of-sets-is-set {ℓ₁}{ℓ₂}{A}{B} iA iB
  = ≃-with-a-set-is-set (≃-sym (+-≃-∑ {ℓ₃ = ℓ₂}{A = A}{B}))
    (∑-set ⊭-is-set λ { 0₂ → fact₁ ; 𝟙₂ → fact₂})
  where
  open import BasicEquivalences
  abstract
    fact₁ : isSet (P⊭-to-A+B {ℓ₃ = ℓ₂} A B 0₂)
    fact₁ = ≃-with-a-set-is-set (lifting-equivalence A) iA

    fact₂ : isSet (P⊭-to-A+B {ℓ₃ = ℓ₂} A B 𝟙₂)
    fact₂ = ≃-with-a-set-is-set (lifting-equivalence B) iB
```

```
+-set = +-of-sets-is-set
```

```
⟦⟧₂-is-set
  : ∀ {ℓ : Level} {n : ℕ}
  ---------------
  → isSet {ℓ} (⟦ n ⟧₂)

⟦⟧₂-is-set {ℓ}{0} = 0-is-set {ℓ}
⟦⟧₂-is-set {ℓ}{succ n} = +-of-sets-is-set 𝟙-is-set ⟦⟧₂-is-set
```

```
∑-≃-base
  : ∀ {ℓ₁ ℓ₂ : Level}
  → {A : Type ℓ₁}{B : A → Type ℓ₂}
  → ((a : A) → isContr (B a))
  ---------------------------
  → ∑ A B ≃ A

∑-≃-base {A = A}{B} discrete-base
  = quasiinverse-to-≃ f (g , (H₁ , H₂))
  where
  private
   f : ∑ A B → A
   f (a , b) = a

   g : ∑ A B ← A
   g a = (a ,  π₁ (discrete-base a))

   H₁ : f ∘ g ∼ id
   H₁ x = idp

   H₂ : g ∘ f ∼ id
   H₂ x = pair= (idp , contrIsProp (discrete-base (π₁ x)) _ _)
```

```
set-is-groupoid
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isSet A
  → isGroupoid A

set-is-groupoid A-is-set a b = prop-is-set (A-is-set a b)
```

Another device to remember this fact (set-is-groupoid) is to see that any simple graph can be seen as a multigraph. Here, the graph represents the path structure of the type in question.

```
module _ {ℓ : Level}(A : Type ℓ) where
```

```
contr-is-set
  : A is-contr → A is-set

contr-is-set A-is-contr = prop-is-set (contr-is-prop A-is-contr)
```

```
≡-preserves-prop
  : ∀ {x y : A}
  → A is-prop
  ------------------
  → (x ≡ y) is-prop

≡-preserves-prop {x}{y} A-is-prop = prop-is-set A-is-prop x y
```

```
≡-preserves-set
  : {x y : A}
  → (A is-set
  -----------------
  → (x ≡ y) is-set)

≡-preserves-set {x}{y} A-is-set = set-is-groupoid A-is-set x y
```

Quite recurrent are the fixed $\sum$-types like $\sum(t : A)(t \equiv x)$. Such types are contractible as we show with the following lemmas.

```
pathto-is-contr
  : (x : A)
  ------------------------------
  → (Σ A (λ t → t ≡ x)) is-contr

pathto-is-contr x = (x , refl x) ,  λ {(a , idp) → idp}
```

```
∑≡x-contr = pathto-is-contr
```

```
pathfrom-is-contr
  : (x : A)
  ------------------------------
  → (Σ A (λ t → x ≡ t)) is-contr

pathfrom-is-contr x = (x , refl x) , λ {(a , idp) → idp}
```

```
∑x≡-contr = pathfrom-is-contr
```

Being contractible give you a section.

```
contr-has-section
  : ∀ {ℓ₂ : Level} {B : A → Type ℓ₂}
  → A is-contr → (a : A)
  ----------------------
  → (u : B a) → Π A B

contr-has-section {B = B} A-is-contr a u
  = λ a' → tr B (contr-connects A-is-contr a a') u
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

```

(continues on next page)

(continued from previous page)

```
open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import DecidableEquality
open import NaturalType
open import HLevelTypes
open import HLevelLemmas
open import HedbergLemmas
open import TruncationType
open import FibreType
open import CoproductIdentities
```

1.40 Types of Morphisms

```
module TypesofMorphisms where
```

1.40.1 Surjections

```
isSurjection
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

isSurjection {B = B} f = (b : B) → ∥ fib f b ∥

isSurjective   = isSurjection
isOnto         = isSurjection
_is-surjection = isSurjection
```

Do not confuse with the traditional logic notation for surjective functions which says f is surjective if $\forall(b : B)\exists(a : A).fa \equiv b$. This is stronger than merely know exists an $(a : A)$ as it was stated above with ∥ `fib f b` ∥.

Therefore, we define the concept of *split-surjection*:

```
isSplitSurjection
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

isSplitSurjection {B = B} f = (b  : B) → fib f b

_is-split-surjection = isSplitSurjection
```

Which is equivalent to say f is a **retraction**:

```
isRetraction
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

isRetraction {A = A}{B} f = ∑ (B → A) (λ g → (b : B) → f (g b) ≡ b)

_is-retraction = isRetraction
```

As a trivial example, we know the identity function is indeed a surjective function. Let us check this.

```
identity-is-surjective : ∀ {ℓ : Level}{A : Type ℓ} →  isSurjection {A = A} id
identity-is-surjective {A = A} b = ∥∥-intro (b , idp)
```

1.40.2 Embeddings

```
isEmbedding
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  ---------------
  → Type (ℓ₁ ⊔ ℓ₂)

isEmbedding {A = A} f = ∀ {x y : A} → isEquiv (ap f {x}{y})

_is-embedding = isEmbedding


is-embedding-has-prop-fibers
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → f is-embedding
  → B is-set
  → ∏[ x : B ] isProp (fib f x)

is-embedding-has-prop-fibers f f-is-emb B-is-set =
  λ b → λ {(a , p₁) (b , p₂) →
  pair= (((ap f , f-is-emb) ∙←) (p₁ · ! p₂)
  , B-is-set _ _ _ _)}
```

1.40.3 Injections

Discuss: should I demand for injective functions to have their domains and codomains as sets?

```
isInjective
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → Type (ℓ₁ ⊔ ℓ₂)

isInjective {A = A} f = ∀ {x y} → f x ≡ f y → x ≡ y

_is-injective = isInjective
```

As a trivial example, let us prove identity is an injective function:

```
identity-is-injective
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isInjective {A = A} id

identity-is-injective p = p
```

```
isInjectiveIsProp
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (iA : isSet A)
  → (f : A → B)
  -----------------------
  → isProp (isInjective f)

isInjectiveIsProp {A = A}{B} iA f i1 i2 =
  aux i1 i2
```

(continues on next page)

(continued from previous page)

```
  where
    private
      aux : isProp (∀ {x y} → (f x ≡ f y → x ≡ y))
      aux = pi-is-prop-implicit
              (λ x → pi-is-prop-implicit (λ y →
                pi-is-prop (λ p → iA x y)
              ))

injective-is-prop = isInjectiveIsProp
```

```
isSurjectionIsProp
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → isProp (isSurjection f)

isSurjectionIsProp f = pi-is-prop (λ b → truncated-is-prop {A = fib f b})

surjective-is-prop = isSurjectionIsProp
```

If the function $f : A \to B$ is a surjection, we are able to get a function $g : B \to A$ by the recursion principle of truncation.

```
fromSurjection
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → isSet B
  → isSurjection f
  → isInjective f
  --------------------------------
  → (b : B) → ∑ A (λ a → f a == b)

fromSurjection {A = A}{B} f iB f-is-onto f-is-injective b
  = trunc-rec (aux b) id (f-is-onto b)
  where
  private
    aux
      : (b : B)
      → isProp (fib f b)

    aux .(f x) (x , idp) (x' , p2) =
      ∑-≡
        (λ y → f y == f x)
        (f-is-injective (! p2))
        (iB (f x') (f x)
            (tr (λ z₁ → f z₁ == f x) (f-is-injective (! p2)) idp) p2)
```

```
preimage-function = fromSurjection
```

1.40.4 Bijections

Bijection is a concept from Set Theory, which meeans that if we want to define it in Univalent Type Theory, we must talk about only functions between (homotopy) sets. Thus, we will find these assumptions in the type for bijections, even though, we do not really need them ¬¬.

```
isBijection
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (f : A → B)
  → isSet A → isSet B
```

(continues on next page)

(continued from previous page)

```
  -------------------
  → Type (ℓ₁ ⊔ ℓ₂)

isBijection f iA iB = isInjective f × isSurjection f

_is-bijection = isBijection
```

Before to proceed to prove that *equivalence* and *bijection* are two logical equivalent concept when we talk about (homotopy) sets, let us give an example of a natural bijection, the idenitity function.

```
identity-is-bijection
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (A-is-set : isSet A)
  → isBijection id A-is-set A-is-set

identity-is-bijection {A} ia = identity-is-injective , identity-is-surjective
```

Discuss: we again see that the assumption of being a set for the domain is required in the way to check the funciton is injective or surjective. This must suggest, we must include this assumption in the Injective definition.

```
Bijection
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (iA : isSet A) → (iB : isSet B)
  → (f : A → B)
  → isBijection f iA iB
  ----------------------
  → A ≃ B

Bijection {A = A}{B} iA iB f (f-is-injective , f-is-onto)
  = qinv-≃ f (g , (H₁ , H₂))
  where
  aux : (b : B) → ∑ A (λ a → f a ≡ b)
  aux = fromSurjection f iB f-is-onto f-is-injective

  g : B → A
  g = π₁ ∘ aux

  H₁ : (b : B) → f (g b) == b
  H₁ b = π₂ (aux b)

  H₂ : (a : A) → g (f a) == a
  H₂ a = f-is-injective (H₁ (f a))
```

```
is-bijection-to-≃ =  Bijection
```

```
≃-to-bijection
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (iA : isSet A)
  → (iB : isSet B)
  ------------------------------------------
  → (e : A ≃ B) → (isBijection (e ∙→) iA iB)

≃-to-bijection iA iB e =
  (λ {x y} p  → ! (∙←∘∙→ e) · (ap (e ∙←) p)  · (∙←∘∙→ e) )  -- is injective
  , λ b → ∣ (e ∙←) b , ∙→∘∙← e ∣                            -- is surjective
  where open import EquivalenceType
```

Bijection and being equivalent are equivalent notions:

```
isBijection-≃-isEquiv
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (iA : isSet A) (iB : isSet B)
  → (f : A → B)
  -----------------------------------
  → isBijection f iA iB  ≃ isEquiv f

isBijection-≃-isEquiv {A = A}{B} iA iB f =
  qinv-≃
    (λ bij → π₂ (Bijection iA iB f bij))
    ((λ isEquivf → ≃-to-bijection iA iB (f , isEquivf))
    , h1 , h2)
  where
    h1 : (λ x → π₂ (Bijection iA iB f (≃-to-bijection iA iB (f , x)))) ∼ id
    h1 e = isContrMapIsProp f _ e

    h2 : (λ x → ≃-to-bijection iA iB (f , π₂ (Bijection iA iB f x))) ∼ id
    h2 bij = ×-is-prop (isInjectiveIsProp iA f) (isSurjectionIsProp f) _ bij

open import QuasiinverseLemmas
```

```
bijIsProp
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (iA : isSet A)(iB : isSet B)
  → (f : A → B)
  ------------------------------
  → isProp (isBijection f iA iB)

bijIsProp iA iB f = isProp-≃ (≃-sym (isBijection-≃-isEquiv iA iB f)) (isEquivIsProp f)
bijection-is-prop = bijIsProp
```

For some reasons, we might need to have the inverse, the actual function, of a bijection. One way I see now is to recover such a function from the equivalence, using `remap`. Let's see this:

```
inverse-of-bijection
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (iA : isSet A) → (iB : isSet B)
  → (f : A → B)
  → isBijection f iA iB
  ------------------------------
  → B → A

inverse-of-bijection iA iB f isBij
  = remap (Bijection iA iB f isBij)

inv-of-bij = inverse-of-bijection
```

```
∘-bijective-and-its-inverse-l
    : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
    → (A-is-set : isSet A) → (B-is-set : isSet B)
    → (f : A → B) → (f-is-bij : isBijection f A-is-set B-is-set)
    ---------------------------------------------------------
    → f ∘ inverse-of-bijection A-is-set B-is-set f f-is-bij ∼ id

∘-bijective-and-its-inverse-l A-is-set B-is-set f f-is-bij =
  lrmap-inverse-h (is-bijection-to-≃ A-is-set B-is-set f f-is-bij)
```

```
∘-bijective-and-its-inverse-r
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → (A-is-set : isSet A)(B-is-set : isSet B)
  → (f : A → B) → (f-is-bij : isBijection f A-is-set B-is-set)
```

(continues on next page)

(continued from previous page)

```
  -------------------------------------------------------------
  → (inverse-of-bijection A-is-set B-is-set f f-is-bij) ∘ f ∼ id

∘-bijective-and-its-inverse-r A-is-set B-is-set f f-is-bij =
  rlmap-inverse-h (is-bijection-to-≃ A-is-set B-is-set f f-is-bij)
```

The inverse of a bijection is clearly a bijection as well.

```
inverse-of-bijection-is-bijection
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁}{B : Type ℓ₂}
  → (A-is-set : isSet A) → (B-is-set : isSet B)
  → (f : A → B) → (f-is-bij : isBijection f A-is-set B-is-set)
  -------------------------------------------------------------
  → isBijection (inverse-of-bijection A-is-set B-is-set f f-is-bij) B-is-set A-is-set

inverse-of-bijection-is-bijection A-is-set B-is-set f f-is-bij
  = inv-f-is-inj , inv-f-is-sur
  where

  inv-f-is-inj : isInjective (inverse-of-bijection A-is-set B-is-set f f-is-bij)
  inv-f-is-inj {x = x}{y} p =
    ! ∘-bijective-and-its-inverse-l A-is-set B-is-set f f-is-bij x
    ·  ap f p
    · ∘-bijective-and-its-inverse-l A-is-set B-is-set f f-is-bij y

  inv-f-is-sur : isSurjection (inverse-of-bijection A-is-set B-is-set f f-is-bij)
  inv-f-is-sur a = ∣ f a , ∘-bijective-and-its-inverse-r A-is-set B-is-set f f-is-bij a ∣
```

```
∘-injectives-is-injective
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level }
  → {A : Type ℓ₁} {B : Type ℓ₂} {C : Type ℓ₃}
  → (f : A → B) → isInjective f
  → (g : B → C) → isInjective g
  ----------------------------
  → isInjective (g ∘ f)

∘-injectives-is-injective f f-is-injective g g-is-injective
  p = f-is-injective (g-is-injective p)
```

As we expect, composition of surjections is also surjections. However, this fact is not trivial and it is a good exercise to understand better propositional truncation.

```
∘-surjection-is-surjection
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level }
  → {A : Type ℓ₁} {B : Type ℓ₂} {C : Type ℓ₃}
  → (f : A → B) → isSurjection f
  → (g : B → C) → isSurjection g
  ----------------------------
  → isSurjection (g ∘ f)

∘-surjection-is-surjection {A = A}{B}{C} f f-is-surjection g g-is-surjection
  c = step₁ (g-is-surjection c)
  where
  step₁ : ∥ ∑ B (λ b → g b ≡ c) ∥ → ∥ ∑ A (λ a → (f :> g) a ≡ c) ∥
  step₁ = trunc-rec ∥∥-is-prop step₂
    where
    step₂ : ∑ B (λ b → g b ≡ c) → ∥ ∑ A (λ a → (f :> g) a ≡ c) ∥
    step₂ (b , p₁) = step₃ b p₁ (f-is-surjection b)
      where
      step₃ : (b : B) → g b ≡ c
        → ∥ ∑ A (λ a → f a ≡ b) ∥ → ∥ ∑ A (λ a → (f :> g) a ≡ c) ∥
```

(continues on next page)

(continued from previous page)

```
      step₃ b p₁ = trunc-rec ∥∥-is-prop step₄
         where
         step₄ : ∑ A (λ a → f a ≡ b) → ∥ ∑ A (λ a → (f :> (λ {a = a₁} → g)) a ≡ c) ∥
         step₄ (a , p) = | a , ((ap g p) · p₁) |
```

Lastly, bijections is also closed by compositions but its proof is just application of the lemmas proved above. Notice the extra requirement which is the domain and codomains need to be sets. This was not stated above for the related lemmas, but it the condition to talk about the concept of bijection.

```
∘-bijections-is-bijection
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}
  → {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
  → (A-is-set : isSet A) (B-is-set : isSet B)(C-is-set : isSet C)
  → (f : A → B) → isBijection f A-is-set B-is-set
  → (g : B → C) → isBijection g B-is-set C-is-set
  ----------------------------------------------
  → isBijection (g ∘ f) A-is-set C-is-set

∘-bijections-is-bijection {A = A}{B}{C}
  A-is-set B-is-set iC
    f (f-is-injective , f-is-surjection)
    g (g-is-injective , g-is-surjection)
  = ∘-injectives-is-injective f f-is-injective g g-is-injective
  , ∘-surjection-is-surjection f f-is-surjection g g-is-surjection
```

Other theorems about +-map

```
inj-from-⊕-injective
  : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level}
  → {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃} {D : Type ℓ₄}
  → {f : A + B → C + D}
  → (isInjective f)
  --------------------------
  → (g : B → D) → ((b : B) → inr (g b) ≡ f (inr b))
  → isInjective g

inj-from-⊕-injective {C = C} f⊕g-is-inj g g-is-f {x} {y} p  =
   inr-is-injective
    (f⊕g-is-inj {x = inr x} (! g-is-f x · ap inr p · g-is-f y))
```

```
right-is-injective-of-⊕-injective
  : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level}
  → {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃} {D : Type ℓ₄}
  → {f : A → C} {g : B → D}
  → (isInjective ⟨ f ⊕ g ⟩)
  --------------------------
  → isInjective g

right-is-injective-of-⊕-injective {C = C} f⊕g-is-inj {x} {y} p =
  inr-is-injective
    (f⊕g-is-inj {x = inr x}
      $ ap (λ w → inr {A = C} w) p)
```

```
left-is-injective-of-⊕-injective
  : ∀ {ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level}
  → {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃} {D : Type ℓ₄}
  → {f : A → C} {g : B → D}
  → (isInjective ⟨ f ⊕ g ⟩)
  --------------------------
  → isInjective f
```

(continues on next page)

(continued from previous page)

```
left-is-injective-of-⊕-injective {C = C} f⊕g-is-inj {x} {y} p =
  inl-is-injective
    (f⊕g-is-inj {x = inl x}
      $ ap (λ w → inl {A = C} w) p)
```

```
:>-is-injective-is-inj
  : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Type ℓ₁}{B : Type ℓ₂}{C : Type ℓ₃}
  → {f : A → B} {g : B → C }
  → isInjective (f :> g)
  ----------------------
  → isInjective f

:>-is-injective-is-inj {f = f} {g} :>-is-inj p = :>-is-inj (ap g p)
```

1.41 Rewriting

We can define Higher inductive types in Agda by rewriting method, same approach as in [HoTT-Agda]. However, this is unsafe.

```
{-# OPTIONS --without-K --exact-split --rewriting #-}
module Rewriting where
```

```
open import BasicTypes
```

{: .axiom}

```
infix 30 _↦_
postulate
  _↦_
    : ∀ {ℓ : Level} {A : Type ℓ}
    → A → A
    -------
    → Type ℓ

{-# BUILTIN REWRITE _↦_ #-}
```

```
{-# OPTIONS --without-K --exact-split --rewriting  #-}

open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import DecidableEquality
open import NaturalType
open import HLevelTypes
open import HLevelLemmas
open import HedbergLemmas
open import Rewriting
```

1.42 Set truncations

```
module SetTruncationType where
```

(continues on next page)

(continued from previous page)

```
module _ {ℓ : Level} (A : Type ℓ) where
  postulate
    ||_||₀ : Type ℓ → Type ℓ
    |_|₀ :  A → || A ||₀
    |||₀-set : || A ||₀ is-set
```

Recursion principle

```
module
  Rec
    {ℓ : Level} {B : Type ℓ}
    (f : A → B)
    (B-is-set : B is-set)
  where
  postulate
    |||₀-rec : || A ||₀ → B
    |||₀-rec-points : (x : A) → (|||₀-rec (| x |₀)) ↦ f x
    {-# REWRITE |||₀-rec-points #-}

    -- |||₀-rec-set
    --    : (x y : A)
    --    → (p q : | x |₀ ≡ | y |₀)
    --    → (r : p ≡ q)
    --    -- → {!!} ↦ B-is-set ()
    -- -- {-# REWRITE |||₀-rec-set #-}


--   : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{P : Type ℓ₂}
--   → isSet P
--   → (A → P)
--   ------------
--   → || A ||₀ → P

-- strunc-rec _ f !| x |₀ = f x
```

Induction principle

```
-- strunc-ind
--   : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {B : || A ||₀ → Type ℓ₂}
--   → ((a : || A ||₀) → isSet (B a))
--   → (g : (a : A) → B | a |₀)
--   ------------------------------
--   → (a : || A ||₀) → B a

-- strunc-ind _ g !| x |₀ = g x
```

1.43 Groupoid truncations

```
{-# OPTIONS --without-K --exact-split #-}
open import BasicTypes
open import BasicFunctions
open import Transport
```

1.44 Product identities

```
module
  ProductIdentities
  where
```

```
prodComponentwise
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {x y : A × B}
  → (x == y)
  --------------------------------
  → (π₁ x == π₁ y) × (π₂ x == π₂ y)

prodComponentwise {x = x} idp = refl (π₁ x) , refl (π₂ x)
```

```
prodByComponents
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {x y : A × B}
  → (π₁ x == π₁ y) × (π₂ x == π₂ y)
  --------------------------------
  → (x == y)

prodByComponents {x = a , b} (idp , idp) = refl (a , b)
```

```
prodCompInverse
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂} {x y : A × B}
  → (b : (π₁ x == π₁ y) × (π₂ x == π₂ y))
  --------------------------------------------
  → prodComponentwise (prodByComponents b) == b

prodCompInverse {x = x} (idp , idp) = refl (refl (π₁ x) , refl (π₂ x))
```

```
prodByCompInverse
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}  {x y : A × B}
  → (b : x == y)
  --------------------------------------------
  → prodByComponents (prodComponentwise b) == b

prodByCompInverse {x = x} idp = refl (refl x)
```

```
×-≡
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{B : Type ℓ₂}
  → {ab ab' : A × B}
  → (p : π₁ ab ≡ π₁ ab') → (π₂ ab) ≡ π₂ ab'
  → ab ≡ ab'

×-≡ idp idp = idp
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import DecidableEquality
open import NaturalType
open import HLevelTypes
open import HLevelLemmas
open import HedbergLemmas
```

1.45 Suspensions

```
module SuspensionType where

  module S where

  private
    data Suspₚ {ℓ : Level} (A : Type ℓ) : Type ℓ where
      Nₚ : Suspₚ A
      Sₚ : Suspₚ A

    data Suspₓ {ℓ} (A : Type ℓ) : Type ℓ where
      mkSusp : Suspₚ A → (𝟙 ℓ → 𝟙 ℓ) → Suspₓ A

  Susp = Suspₓ
```

- point-constructors

```
North : ∀ {ℓ} {A : Type ℓ} → Susp A
North = mkSusp Nₚ _

South : ∀ {ℓ} {A : Type ℓ} → Susp A
South = mkSusp Sₚ _
```

- path-constructors

```
postulate
  merid : ∀ {ℓ} {A : Type ℓ}
        → A
        → Path {ℓ}{Susp A} North South
```

Recursion principle on points

```
Susp-rec
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{C : Type ℓ₂}
  → (cₙ cₛ  : C)
  → (merid' : A → cₙ == cₛ)
  ------------------------
  → (Susp A → C)

Susp-rec cₙ _ _ (mkSusp Nₚ _) = cₙ
Susp-rec _ cₛ _ (mkSusp Sₚ _) = cₛ
```

Recursion principle on paths

```
postulate
  Susp-βrec
    : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{C : Type ℓ₂}
    → {cₙ cₛ : C} {mer : A → cₙ == cₛ}
    → {a : A}
    -------------------------------------------
    → ap (Susp-rec cₙ cₛ mer) (merid a) == mer a
```

Induction principle on points

```
Susp-ind
  : ∀ {ℓ : Level} {A : Type ℓ} (C : Susp A → Type ℓ)
  → (N' : C North)
  → (S' : C South)
  → (merid' : (x : A) → N' == S' [ C ↓ (merid x) ])
  --------------------------------------------------
  → ((x : Susp A) → C x)
```

(continues on next page)

(continued from previous page)

```
Susp-ind _ N' S' _ (mkSusp Nₚ _) = N'
Susp-ind _ N' S' _ (mkSusp Sₚ _) = S'
```

Induction principle on paths

```
postulate
  Susp-βind
    : ∀ {ℓ} {A : Type ℓ} (C : Susp A → Type ℓ)
    → (N' : C North)
    → (S' : C South)
    → (merid' : (x : A) → N' == S' [ C ↓ (merid x)]) {x : A}
    --------------------------------------------------------
    → apd (Susp-ind C N' S' merid') (merid x) == merid' x
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import HLevelTypes
```

1.46 Intervals

Interval. An interval is defined by taking two points (two elements) and a path between them (an element of the equality type). This path is nontrivial.

```
module IntervalType where

  private

    data !I : Type₀ where
      !Izero : !I
      !Ione : !I

  I : Type₀
  I = !I

  Izero : I
  Izero = !Izero

  Ione : I
  Ione = !Ione

  postulate
    seg : Izero == Ione
```

Recursion principle on points.

```
I-rec
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (a b : A)
  → (p : a == b)
  --------------
  → (I → A)

I-rec a _ _ !Izero = a
I-rec _ b _ !Ione  = b
```

Recursion principle on paths.

```
postulate
  I-βrec
    : ∀ {ℓ : Level} (A : Type ℓ)
    → (a b : A)
    → (p : a == b)
    --------------------------
    → ap (I-rec a b p) seg == p
```

```
{-# OPTIONS --without-K --exact-split --rewriting #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import DecidableEquality
open import NaturalType
open import HLevelTypes
open import HLevelLemmas
open import HedbergLemmas
```

1.47 Propositional truncation

Propositional truncation (or reflection) is the universal solution to the problem of mapping X to a proposition P:

Notes:

- It's possible to extend MLTT to get truncations for all types. (Such as resizing + funext, or higher inductive types.)

For a different way of formalising trucation see: agda-premises[10].

```
module
  TruncationType
  where
```

```
private
  data
    !∥_∥ {ℓ} (A : Type ℓ)
      : Type ℓ
      where
      !|_| : A → !∥ A ∥
```

```
∥_∥
  : ∀ {ℓ : Level} (A : Type ℓ)
  → Type ℓ

∥ A ∥ = !∥ A ∥

prop-trunc = ∥_∥
```

```
|_|
  : ∀ {ℓ : Level} {X : Type ℓ}
  ------------
  → X → ∥ X ∥
```

(continues on next page)

[10] https://hub.darcs.net/gylterud/agda-premises/browse/Premises/Truncation.agda

(continued from previous page)

```
| x | = !| x |
```

```
||||-intro = |_|
```

Any two elements of the truncated type are equal

{: .axiom}

```
postulate
  trunc
    : ∀ {ℓ : Level} {A : Type ℓ}
    → isProp ∥ A ∥
```

Recursion principle

```
trunc-rec
  :  ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}{P : Type ℓ₂}
  → isProp P
  → (A → P)
  -----------
  → ∥ A ∥ → P

trunc-rec _ f !| x | = f x
```

```
trunc-elim = trunc-rec
||||-rec     = trunc-rec
```

There exists the possibility to charactherize, propositional truncation using an impredicative approach, which means, our definition will lay on a larger universe as on the right-hand side in the following formulation.

$$\parallel X \parallel \Leftrightarrow \prod (P : \mathsf{Type}), \mathsf{isProp}(P) \to (X \to P) \to P$$

Remarks:

- rhs is a propositon assuming funext
- this equation tells us an important relation (a pattern) between the type (in this case, prop-trunc) and its elimination principle (trunc-rec)
- *Impredicative* means in this context: "it is defined in terms of quantification over a family which the thing we are defining is a member of." (Gylterud).

1.47.1 Truncation Lemmas

```
truncated-is-prop
  : ∀ {ℓ : Level} {A : Type ℓ}
  → isProp (∥ A ∥)

truncated-is-prop = trunc
```

```
||||-is-prop     = truncated-is-prop
trunc-is-prop = truncated-is-prop
```

```
trunc-≃-prop
  : ∀ {ℓ : Level} {A : Type ℓ}
  → A is-prop
  -----------
  → ∥ A ∥ ≃ A
```

(continues on next page)

(continued from previous page)

```
trunc-≃-prop pA = lemma333 trunc pA (trunc-rec pA id) |_|
```

A relation between double implication and the truncation of a type:

```
postulate
 trunc-⇔-¬¬
   : ∀ {ℓ} {X : Type ℓ}
   → ∥ X ∥ ⇔ (¬ (¬ X))
```

Using propositional truncation, we are able to define properly the logical disjunction and existence as follows.

```
_∨_
  : ∀ {ℓ₁ ℓ₂ : Level}
  → (p : hProp ℓ₁) (q : hProp ℓ₂)
  → Type (ℓ₁ ⊔ ℓ₂)
(P , _) ∨ (Q , _) = ∥ P + Q ∥

infix 2 _∨_
```

Conjunction is the product of two mere propositons.

```
_∧_
  : ∀ {ℓ₁ ℓ₂ : Level}
  → (p : hProp ℓ₁) (q : hProp ℓ₂)
  → Type (ℓ₁ ⊔ ℓ₂)

(P , _) ∧ (Q , _) = P × Q

infix  2 _∧_
```

```
∃[_]_
  : ∀ {ℓ : Level}
  → (T : Type ℓ) → (P : T → hProp ℓ)
  → Type ℓ

∃[ T ] P = ∥ ∑ T (λ x → π₁ (P x)) ∥
```

Another use of propositional truncation is to say a type A is non-empty. In this case, we have an element of $\| A \|$

```
_is-non-empty
  : ∀ {ℓ : Level}
  → (A : Type ℓ)
  → Type ℓ
A is-non-empty = ∥ A ∥

infixl 100 _is-non-empty
```

```
is-non-empty-is-prop
  : ∀ {ℓ : Level}{A : Type ℓ}
  → isProp (A is-non-empty)

is-non-empty-is-prop = ∥∥-is-prop
```

For any type A and a term $a : A$, we shall say the connected commponent of a is all the terms in A “connected” with a.

```
connected-component
  : ∀ {ℓ : Level} {A : Type ℓ}
  → (a : A)
  → Type ℓ

connected-component {A = A} a = ∑ A (λ x → ∥ a ≡ x ∥ )
```

Consequently, two terms appear to be in the same component whenever there is an element in ∥ x ≡ y ∥.

```
_is-in-the-same-component-of_
  : ∀ {ℓ : Level}{A : Type ℓ}
  → (x y : A) → Type ℓ

x is-in-the-same-component-of y = ∥ x ≡ y ∥

infix 100 _is-in-the-same-component-of_
```

```
_is-connected
  : ∀ {ℓ : Level} (A : Type ℓ)
  → Type ℓ

A is-connected =
    (A is-non-empty)
  × ((x y : A) → (x is-in-the-same-component-of y))
```

```
is-connected-is-prop
  : ∀ {ℓ : Level} {A : Type ℓ}
  --------------------------
  → isProp (A is-connected)

is-connected-is-prop =
  ×-is-prop
    is-non-empty-is-prop
    (pi-is-prop (λ x → pi-is-prop λ y → trunc-is-prop))
```

1.48 Quotient type

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

open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import DecidableEquality
open import NaturalType
open import HLevelTypes
open import HLevelLemmas
open import HedbergLemmas
```

```
module QuotientType where

  record QRel {ℓ : Level} (A : Type ℓ) : Type (lsuc ℓ) where
    field
      R      : A → A → Type ℓ
      Aset   : isSet A
```

(continues on next page)

(continued from previous page)

```
    Rprop : (a b : A) → isProp (R a b)

open QRel {{...}} public

private
  data _!/_ {ℓ : Level} (A : Type ℓ)  (r : QRel A)
    : Type (lsuc ℓ)
    where
    ![_] : A → (A !/ r)

_/_
  : {ℓ : Level} (A : Type ℓ)  (r : QRel A)
  → Type (lsuc ℓ)

A / r = (A !/ r)

[_]
  : ∀ {ℓ : Level} {A : Type ℓ}
  → A → {r : QRel A}
  ---------
  → (A / r)

[ a ] = ![ a ]

-- Equalities induced by the relation
postulate
  Req
    : ∀ {ℓ : Level} {A : Type ℓ}  {r : QRel A}
    → {a b : A}
    → R {{r}} a b
    --------------------
    → [ a ] {r} == [ b ]

-- The quotient of a set is again a set
postulate
  Rtrunc
    : ∀ {ℓ : Level} {A : Type ℓ}  {r : QRel A}
    ---------------
    → isSet (A / r)
```

Recursion principle

```
QRel-rec
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁}  {r : QRel A} {B : Type ℓ₂}
  → (f : A → B)
  → ((x y : A) → R {{r}} x y → f x == f y)
  ---------------------------------------
  → A / r → B

QRel-rec f p ![ x ] = f x
```

Induction principle

```
QRel-ind
  : ∀ {ℓ₁ ℓ₂ : Level} {A : Type ℓ₁} {r : QRel A} {B : A / r → Type ℓ₂}
  → (f : ((a : A) → B [ a ]))
  → ((x y : A) → (o : R {{r}} x y) → (transport B (Req o) (f x)) == f y)
  -------------------
  → (z : A / r) → B z

QRel-ind f p ![ x ] = f x
```

Recursion in two arguments

```
QRel-rec-bi
  : ∀ {ℓ₁ ℓ₂ : Level}{A : Type ℓ₁} {r : QRel A} {B : Type ℓ₂}
  → (f : A → A → B)
  → ((x y z t : A) → R {{r}} x y → R {{r}} z t → f x z == f y t)
  -------------------
  → A / r → A / r → B

QRel-rec-bi f p ![ x ] ![ y ] = f x y
```

```
Qrel-prod
  : ∀ {ℓ : Level}{A : Type ℓ}
  → (r : QRel A)
  --------------
  → QRel (A × A)

Qrel-prod r =
  record { R = λ { (a , b) (c , d) → (R {{r}} a c) × (R {{r}} b d) }
         ; Aset = isSet-prod (Aset {{r}}) (Aset {{r}})
         ; Rprop = λ { (x , y) (q , w) → isProp-prod (Rprop {{r}} x q) (Rprop {{r}} y w)} }
```

```
{-# OPTIONS --without-K --exact-split  --rewriting #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import HLevelTypes
```

1.49 Circles

The circle type is constructed by postulating a type with a single element (base) and a nontrivial path (loop).

```
module CircleType {ℓ : Level} where
  open import Rewriting
```

```
postulate
  S¹ : Type ℓ
  base : S¹
  loop : base ≡ base
```

Recursion principle on points

```
postulate
  S¹-rec
    : ∀ {ℓ : Level}
    → (A : Type ℓ)
    → (a : A)
    → (p : a ≡ a)
    --------------
    → (S¹ → A)
```

Recursion principle on paths

```
postulate
  S¹-βrec-base
    : ∀ {ℓ : Level}
```

(continues on next page)

(continued from previous page)

```
    → (A : Type ℓ)
    → (a : A)
    → (p : a ≡ a)
    ------------------------------
    → S¹-rec A a p base ↦ a

{-# REWRITE S¹-βrec-base #-}
```

```
postulate
  S¹-βrec-loop
    : ∀ {ℓ : Level}
    → (A : Type ℓ)
    → (a : A)
    → (p : a ≡ a)
    ------------------------------
    → ap (S¹-rec A a p) loop ↦ p

{-# REWRITE S¹-βrec-loop #-}
```

Induction principle on points

```
postulate
  S¹-ind
    : ∀ {ℓ : Level}
    → {P : S¹ → Type ℓ}
    → (x : P base)
    → (x == x [ P ↓ loop ])
    -----------------------
    → ((t : S¹) → P t)
```

```
postulate
  S¹-βind-base
    : ∀ {ℓ : Level}
    → {P : S¹ → Type ℓ}
    → (x : P base)
    → (p : x == x [ P ↓ loop ])
    ---------------------------
    → S¹-ind x p base ↦ x

{-# REWRITE S¹-βind-base #-}
```

Induction principle on paths

```
postulate
  S¹-βind-loop
    :  ∀ {ℓ : Level}
    → {P : S¹ → Type ℓ}
    → (x : P base)
    → (p : x == x [ P ↓ loop ])
    → apd (S¹-ind x p) loop ↦ p

{-# REWRITE S¹-βind-loop #-}
```

```
this-proofs-rewriting
    :  ∀ {ℓ : Level}
    → {P : S¹ → Type ℓ}
    → (x : P base)
    → (p : x == x [ P ↓ loop ])
    → apd (S¹-ind x p) loop ≡ p
```

(continues on next page)

(continued from previous page)

```
this-proofs-rewriting _ _ = idp
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import HLevelTypes
open import CircleType
open import GroupType
```

1.50 Fundamental group

Definition of the fundamental group of a type. Let `a:A` be one point of the type. The fundamental group on `a` is the group given by proofs of the equality (`a=a`).

```
module FundamentalGroupType where
```

Definition of the fundamental group.

```
Ω
  : ∀ {ℓ : Level}(A : Type ℓ)
  ------------------
  → (a : A) → Type ℓ

Ω A a = (a == a)
```

```
Ω-gr
  : ∀ {ℓ : Level} (A : hSet ℓ)
  → (a : π₁ A) → Group {ℓ}
Ω-gr (A , A-is-set) a =
  monoid
    (Ω A a)
    (refl a)
    _·_
    (set-is-groupoid A-is-set a a)
    (λ (x : Ω A a) → refl x)
    (λ (x : Ω A a) → ! (·-runit x))
    (λ x y z → ! (·-assoc x y z)  )
  , inv , λ x → ·-rinv x , ·-linv x
  where
    open import MonoidType
    open import HLevelLemmas
```

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

open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import HLevelTypes
```

1.51 Monoids

A **monoid** is a algebraic structure on a set with a featured object called the *unit* and an associative binary operation (also called the multiplication) that fulfills certain properties described below.

Before monoids, we could define instead a much simpler structure, the magma. A **magma** is basically a set and an binary operation. Then, any monoid is also magma, but magmas are not so interesting as the monoids.

```
module
  MonoidType
    where
```

```
record
  Monoid {ℓ : Level}
    : Type (lsuc ℓ)
    where
  constructor monoid
  field
    M    : Type ℓ         -- The carrier
    e    : M              -- Unit element (at least one element, this one)
    _⊕_ : M → M → M     -- Operation

    M-is-set : isSet M   -- the carrier is a set

    -- axioms:
    l-unit : (x : M) → (e ⊕ x) == x
    r-unit : (x : M) → (x ⊕ e) == x
    assoc  : (x y z : M) → (x ⊕ (y ⊕ z)) == ((x ⊕ y) ⊕ z)
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType
open import HLevelTypes
```

1.52 Relation

```
module RelationType where

  record Rel {ℓ : Level} (A : Type ℓ) : Type (lsuc ℓ) where
    field
      R     : A → A → Type ℓ
      Rprop : (a b : A) → isProp (R a b)
  open Rel {{...}} public
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import DecidableEquality
open import NaturalType
open import HLevelTypes
open import HedbergLemmas
```

1.53 Integers

```
module IntegerType where

  data ℤ : Type₀ where
    zer : ℤ
    pos : ℕ → ℤ
    neg : ℕ → ℤ

  -- Inclusion of the natural numbers into the integers
  NtoZ : ℕ → ℤ
  NtoZ zero     = zer
  NtoZ (succ n) = pos n

  -- Successor function
  zsucc : ℤ → ℤ
  zsucc zer             = pos 0
  zsucc (pos x)         = pos (succ x)
  zsucc (neg zero)      = zer
  zsucc (neg (succ x)) = neg x

  -- Predecessor function
  zpred : ℤ → ℤ
  zpred zer             = neg 0
  zpred (pos zero)      = zer
  zpred (pos (succ x)) = pos x
  zpred (neg x)         = neg (succ x)

  -- Successor and predecessor
  zsuccpred-id : (n : ℤ) → zsucc (zpred n) == n
  zsuccpred-id zer             = refl _
  zsuccpred-id (pos zero)      = refl _
  zsuccpred-id (pos (succ n)) = refl _
  zsuccpred-id (neg n)         = refl _

  zpredsucc-id : (n : ℤ) → zpred (zsucc n) == n
  zpredsucc-id zer             = refl _
  zpredsucc-id (pos n)         = refl _
  zpredsucc-id (neg zero)      = refl _
  zpredsucc-id (neg (succ n)) = refl _

  zpredsucc-succpred : (n : ℤ) → zpred (zsucc n) == zsucc (zpred n)
  zpredsucc-succpred zer = refl zer
  zpredsucc-succpred (pos zero) = refl (pos zero)
  zpredsucc-succpred (pos (succ x)) = refl (pos (succ x))
  zpredsucc-succpred (neg zero) = refl (neg zero)
  zpredsucc-succpred (neg (succ x)) = refl (neg (succ x))

  zsuccpred-predsucc : (n : ℤ) → zsucc (zpred n) == zpred (zsucc n)
  zsuccpred-predsucc n = inv (zpredsucc-succpred n)

  -- Equivalence given by successor and predecessor
  zequiv-succ : ℤ ≃ ℤ
  zequiv-succ = qinv-≃ zsucc (zpred , (zsuccpred-id , zpredsucc-id))

  -- Negation
  private
    - : ℤ → ℤ
    - zer     = zer
    - (pos x) = neg x
    - (neg x) = pos x
```

(continues on next page)

(continued from previous page)

```
double- : (z : ℤ) → - (- z) == z
double- zer = refl _
double- (pos x) = refl _
double- (neg x) = refl _

zequiv- : ℤ ≃ ℤ
zequiv- = qinv-≃ - (- , (double- , double-))

-- Addition on integers
infixl 60 _+ᶻ_
_+ᶻ_ : ℤ → ℤ → ℤ
zer +ᶻ m = m
pos zero +ᶻ m = zsucc m
pos (succ x) +ᶻ m = zsucc (pos x +ᶻ m)
neg zero +ᶻ m = zpred m
neg (succ x) +ᶻ m = zpred (neg x +ᶻ m)

-- s on addition
+ᶻ-lunit : (n : ℤ) → n == n +ᶻ zer
+ᶻ-lunit zer             = refl _
+ᶻ-lunit (pos zero)      = refl _
+ᶻ-lunit (pos (succ x)) = ap zsucc (+ᶻ-lunit (pos x))
+ᶻ-lunit (neg zero)      = refl _
+ᶻ-lunit (neg (succ x)) = ap zpred (+ᶻ-lunit (neg x))


+ᶻ-unit-zsucc : (n : ℤ) → zsucc n == (n +ᶻ pos zero)
+ᶻ-unit-zsucc zer = refl _
+ᶻ-unit-zsucc (pos zero) = refl _
+ᶻ-unit-zsucc (pos (succ x)) = ap zsucc (+ᶻ-unit-zsucc (pos x))
+ᶻ-unit-zsucc (neg zero) = refl _
+ᶻ-unit-zsucc (neg (succ x)) = inv (zpredsucc-id (neg x)) · ap zpred (+ᶻ-unit-zsucc (neg x))

+ᶻ-unit-zpred : (n : ℤ) → zpred n == (n +ᶻ neg zero)
+ᶻ-unit-zpred zer = refl _
+ᶻ-unit-zpred (pos zero) = refl zer
+ᶻ-unit-zpred (pos (succ x)) = inv (zsuccpred-id (pos x)) · ap zsucc (+ᶻ-unit-zpred (pos x))
+ᶻ-unit-zpred (neg zero) = refl _
+ᶻ-unit-zpred (neg (succ x)) = ap zpred (+ᶻ-unit-zpred (neg x))


+ᶻ-pos-zsucc : (n : ℤ) → (x : ℕ) → zsucc (n +ᶻ pos x) == n +ᶻ pos (succ x)
+ᶻ-pos-zsucc zer x = refl _
+ᶻ-pos-zsucc (pos zero) x = refl _
+ᶻ-pos-zsucc (pos (succ n)) x = ap zsucc (+ᶻ-pos-zsucc (pos n) x)
+ᶻ-pos-zsucc (neg zero) x = zsuccpred-id (pos x)
+ᶻ-pos-zsucc (neg (succ n)) x = zsuccpred-predsucc (neg n +ᶻ pos x) · ap zpred (+ᶻ-pos-zsucc (neg n) x)

+ᶻ-neg-zpred : (n : ℤ) → (x : ℕ) → zpred (n +ᶻ neg x) == n +ᶻ neg (succ x)
+ᶻ-neg-zpred zer x = refl _
+ᶻ-neg-zpred (pos zero) x = zpredsucc-id (neg x)
+ᶻ-neg-zpred (pos (succ n)) x = zpredsucc-succpred (pos n +ᶻ neg x) · ap zsucc (+ᶻ-neg-zpred (pos n) x)
+ᶻ-neg-zpred (neg zero) x = refl _
+ᶻ-neg-zpred (neg (succ n)) x = ap zpred (+ᶻ-neg-zpred (neg n) x)

+ᶻ-comm : (n m : ℤ) → n +ᶻ m == m +ᶻ n
+ᶻ-comm zer m = +ᶻ-lunit m
+ᶻ-comm (pos zero) m = +ᶻ-unit-zsucc m
+ᶻ-comm (pos (succ x)) m = ap zsucc (+ᶻ-comm (pos x) m) · +ᶻ-pos-zsucc m x
```

(continues on next page)

(continued from previous page)

```
  +ᶻ-comm (neg zero) m = +ᶻ-unit-zpred m
  +ᶻ-comm (neg (succ x)) m = ap zpred (+ᶻ-comm (neg x) m) · +ᶻ-neg-zpred m x



  -- Decidable equality
  pos-inj : {n m : ℕ} → pos n == pos m → n == m
  pos-inj {n} {.n} idp = refl n

  neg-inj : {n m : ℕ} → neg n == neg m → n == m
  neg-inj {n} {.n} idp = refl n

  z-decEq : decEq ℤ
  z-decEq zer zer = inl (refl zer)
  z-decEq zer (pos x) = inr (λ ())
  z-decEq zer (neg x) = inr (λ ())
  z-decEq (pos x) zer = inr (λ ())
  z-decEq (pos n) (pos m) with (nat-decEq n m)
  z-decEq (pos n) (pos m) | inl p = inl (ap pos p)
  z-decEq (pos n) (pos m) | inr f = inr (f ∘ pos-inj)
  z-decEq (pos n) (neg m) = inr (λ ())
  z-decEq (neg n) zer = inr (λ ())
  z-decEq (neg n) (pos x₁) = inr (λ ())
  z-decEq (neg n) (neg m) with (nat-decEq n m)
  z-decEq (neg n) (neg m) | inl p = inl (ap neg p)
  z-decEq (neg n) (neg m) | inr f = inr (f ∘ neg-inj)

  z-isSet : isSet ℤ
  z-isSet = hedberg z-decEq


  -- Multiplication
  infixl 60 _*ᶻ_
  _*ᶻ_ : ℤ → ℤ → ℤ
  zer *ᶻ m = zer
  pos zero *ᶻ m = m
  pos (succ x) *ᶻ m = (pos x *ᶻ m) +ᶻ m
  neg zero *ᶻ m = - m
  neg (succ x) *ᶻ m = (neg x *ᶻ m) +ᶻ (- m)


  -- Ordering
  _<ᶻ_ : ℤ → ℤ → Type₀
  zer <ᶻ zer = ⊥ lzero
  zer <ᶻ pos x = ⊤ lzero
  zer <ᶻ neg x = ⊥ lzero
  pos x <ᶻ zer = ⊥ lzero
  pos x <ᶻ pos y = x <ₙ y
  pos x <ᶻ neg y = ⊥ lzero
  neg x <ᶻ zer = ⊤ lzero
  neg x <ᶻ pos y = ⊤ lzero
  neg x <ᶻ neg y = y <ₙ x
```

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import HLevelTypes
open import MonoidType
```

1.54 Groups

```
module
  GroupType
    where
```

A group G is a monoid with something else, *inverses* for each element. This means, there is a function $\mathsf{inverse} : G \to G$ such that:

$$\forall (x : G) \to \mathsf{inverse}(x) <> x \equiv e \text{ and } x <> \mathsf{inverse}(x) \equiv e,$$

where G is the group, e the unit and $<>$ the operation from the underlined monoid. This is the following type for groups:

```
Group
  : ∀ {ℓ : Level} → Type (lsuc ℓ)

Group
  = ∑ (Monoid) (λ {(monoid G e _<>_ GisSet lunit runit assoc)
    → ∑ (G → G) (λ inverse →
      ∏ G (λ x →
        -- properties:
        (   (x <> inverse x) == e)
        × ( (inverse x <> x) == e ))
        )})
```

1.55 The type of natural numbers

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

open import TransportLemmas
open import EquivalenceType

open import ProductIdentities
open import CoproductIdentities

open import HomotopyType
open import FunExtAxiom
open import QuasiinverseType
open import DecidableEquality
open import HLevelTypes
open import HedbergLemmas
open import MonoidType

module NaturalType where
```

```
private
  code : ℕ → ℕ → Type₀
  code 0        0        = ⊤ lzero
  code 0        (succ m) = ⊥ lzero
  code (succ n) 0        = ⊥ lzero
  code (succ n) (succ m) = code n m

crefl : (n : ℕ) → code n n
crefl zero     = ⋆
crefl (succ n) = crefl n

private
  encode : (n m : ℕ) → (n == m) → code n m
```

(continues on next page)

(continued from previous page)

```
  encode n m p = transport (code n) p (crefl n)

  decode : (n m : ℕ) → code n m → n == m
  decode zero zero c = refl zero
  decode zero (succ m) ()
  decode (succ n) zero ()
  decode (succ n) (succ m) c = ap succ (decode n m c)
```

```
zero-not-succ
  : (n : ℕ)
  → ¬ (succ n == zero)

zero-not-succ n = encode (succ n) 0
```

The successor function is injective.

```
succ-inj
  : {n m : ℕ}
  → (succ n ≡ succ m)
  → n ≡ m

succ-inj {n} {m} p = decode n m (encode (succ n) (succ m) p)
```

```
+-inj
  : (k : ℕ) {n m : ℕ}
  → (k +ₙ n == k +ₙ m)
  → n == m

+-inj zero    p = p
+-inj (succ k) p = +-inj k (succ-inj p)
```

```
nat-decEq
  : decEq ℕ

nat-decEq zero zero = inl (refl zero)
nat-decEq zero (succ m) = inr (λ ())
nat-decEq (succ n) zero = inr (λ ())
nat-decEq (succ n) (succ m) with (nat-decEq n m)
nat-decEq (succ n) (succ m) | inl p = inl (ap succ p)
nat-decEq (succ n) (succ m) | inr f = inr λ p → f (succ-inj p)
```

```
natIsDec
  : (n m : ℕ)
  → (n ≡ m) + (¬ (n ≡ m))

natIsDec zero zero      = inl idp
natIsDec zero (succ m) = inr (λ ())
natIsDec (succ n) zero = inr (λ ())
natIsDec (succ n) (succ m) with natIsDec n m
... | inl p  = inl (ap succ p)
... | inr ¬p = inr λ sn=sm → ¬p (succ-inj sn=sm)
```

```
nat-is-set : isSet ℕ
nat-is-set = hedberg natIsDec
```

```
ℕ-is-set : isSet ℕ
ℕ-is-set = nat-is-set
nat-isSet = nat-is-set
```

```
ℕ-plus-monoid : Monoid
ℕ-plus-monoid = record
  { M = ℕ
  ; M-is-set = nat-isSet
  ; _⊕_      = plus
  ; e = zero
  ; l-unit = plus-lunit
  ; r-unit = plus-runit
  ; assoc = plus-assoc
  }
```

```
_<ₙ_ : ℕ → ℕ → Type₀
n <ₙ m = Σ ℕ (λ k → n +ₙ succ k == m)
```

```
<-isProp
  : (n m : ℕ)
  → isProp (n <ₙ m)

<-isProp n m (k1 , p1) (k2 , p2) =
  Σ-bycomponents (succ-inj (+-inj n (p1 · inv p2)) , nat-isSet _ _ _ _)
```

```
module _ {ℓ : Level} where
```

```
open ℕ-ordering ℓ
succ-<-inj
  : ∀ {n m : ℕ}
  → succ n < succ m
  → n < m
succ-<-inj {0} {succ m} ∗ = ∗
succ-<-inj {succ n} {succ m} p = succ-<-inj {n}{m} p
```

```
_≤ₙ_ : ℕ → ℕ → Type ℓ
0      ≤ₙ 0      = ⊤ ℓ
0      ≤ₙ succ b = ⊤ ℓ
succ a ≤ₙ 0      = ⊥ ℓ
succ a ≤ₙ succ b = a ≤ₙ b
```

We can express the property of being the minimum of some given predicate as follows (See the symmetry book).

```
_is-the-minimum-of_
  : ∀ {ℓ : Level}
  → (n : ℕ )
  → (P : ℕ → hProp ℓ)
  → Type ℓ
n is-the-minimum-of P = π₁ (P n) × ((m : ℕ) → π₁ (P m) → n <ₙ (m +ₙ 1))
  -- where open ℕ-< {level-of (π₁ (P n))}
```

```
_is-the-maximum-of_
  : {ℓ : Level}
  → (n : ℕ )
  → (P : ℕ → hProp ℓ)
  → Type ℓ

n is-the-maximum-of P = π₁ (P n) × ((m : ℕ) → π₁ (P m) →  (m +ₙ 1) <ₙ n )
```

Move this somewhere else:

‘‘‘agda(m $+_n$ 1) Max : ∀ {ℓ : Level} → (P : ℕ → hProp ℓ) → Type ℓ

Max P = $\sum$ ℕ (λ a → a is-the-maximum-of P) ‘‘‘

```
{-# OPTIONS --without-K --exact-split #-}
open import TransportLemmas
open import EquivalenceType

open import HomotopyType
open import FunExtAxiom
open import HLevelTypes

open import SetTruncationType
```

1.56 Connectedness

1.56.1 0-connected type

```
{-
_is-0-connected
  : ∀ {ℓ : Level}
  → (A : Type ℓ)
  → Type ℓ

A is-0-connected = isContr (∥ A ∥₀)
-}
```

```
{-# OPTIONS --without-K --exact-split #-}
open import BasicTypes

open import HLevelTypes
open import TruncationType
```

1.57 The Axiom Of Choice

```
module
  TheAxiomOfChoice
  where
```

Let's write in type theory, the following logic formulation of of the axiom of choice:

$$(\forall(b : B).\exists(a : Ab).P(b, a)) \Rightarrow \exists(g : (b : B) \to Ab).\forall(b : B).P(b, g(b)).$$

```
postulate
  Choice
    : ∀ {ℓ₁ ℓ₂ ℓ₃ : Level}
    -- Asummption 1:
      (B : Type ℓ₁)
    → isSet B
    -- Asummption 2:
    → (A : B → Type ℓ₂)
    → ∀ (b : B) → isSet (A b)
    -- Asummption 3:
    → (P : (b : B) → (A b) → Type ℓ₃)
    → ∀ (b : B)(a : A b) → isProp (P b a)
    ----------------------------------
    → (∀ b → ∥ ∑ (A b) (λ a → P b a) ∥)
    → ∥ ∑ (Π B A) (λ g → P b (g b)) ∥
```

Contributors Collaborations are always welcomed. At the moment, me, Jonathan Prieto-Cubides, I'm using the library to type-check my proofs for my research project at the University of Bergen[11].

- Jonathan Prieto-Cubides[12]

People involved in making this library better, although not directly involved are:

- Håkon Robbestad Gylterud[13]
- Marc Bezem[14]
- People from the Agda mailing list[15]

References

- Theory
  - The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics[16]. 2013.
  - The Homotopy Type Theory and Univalent Foundations CAS project. Symmetry Book[17]. 2020.
  - Escardo, M. Introduction to Univalent Foundations of Mathematics with Agda[18]. 2019.
  - Rikje, E. Introduction to Homotopy Type Theory[19]. 2019.
- Implementation:
  - Agda-HoTT[20]
  - Agda-premises[21]
  - HoTT-Agda[22]
  - Agda-Base[23]

https://www.uib.no/
jonathan.cubides@uib.no
https://hakon.gylterud.net
https://cas.oslo.no/fellows/marc-bezem-article2086-828.html
https://lists.chalmers.se/mailman/listinfo/agda
http://homotopytypetheory.org
https://github.com/UniMath/SymmetryBook
https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/
https://github.com/EgbertRijke/HoTT-Intro
https://mroman42.github.io/ctlc/agda-hott/Total.html
https://hub.darcs.net/gylterud/agda-premises
https://hott.github.io/HoTT-Agda/
https://github.com/pcapriotti/agda-base