HoTT Chapter 2 Exercises

November 17, 2014

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

module Ch2 where

open import Base
open import Ch1
```

1

Lemma (2.1.2)**.** *For every type A and every $x, y, z : A$ there is a function $(x = y) \to (y = z) \to (x = z)$ written $p \to q \to p \cdot q$, such that $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$ for any $x : A$.*

We call $p \cdot q$ the concatenation or composite of p and q.

Exercise 2.1 Show that the three obvious proofs of Lemma 2.1.2 are pairwise equal.

Proof. (this justifies denoting "the" concatenation function as $\cdot$)

First, we need a type to inhabit. The type of any concatenation operator is

$$\prod_{x,y,z:A} \prod_{p:x=y} \prod_{q:y=z} (x = z)$$

Thus far, the only tool we have to inhabit such a type is path induction. So, we first write down a family

$$D_1(x, y, p) : \prod_{z:A} (y = z) \to (x = z)$$

That is, given $x, y : A$ and a path from x to y, we want a function that takes paths from y to z to paths from x to z.

Path induction dictates that we now need a

$$d_1 : \prod_{x:A} D(x, x, \mathsf{refl}_x)$$

hence

$$d_1(x) : \prod_{z:A} (x = z) \to (x = z)$$

So, given a path from x to z, we want a path from x to z. We'll take the easy way out on this one!

```
_•₁_ : ∀ {i} {A : Type i} {x y z : A} → (x == y) → (y == z) → (x == z)
_•₁_ {i} {A} {_} {_} {z} = ind== D d where
    D : (x y : A) → (p : x == y) → Type i
    D x y _ = y == z → x == z

    d : (x : A) → D x x refl
    d _ = λ q → q
```

For another construction, we do path induction in "the other direction". That is, we will define

$$D_2 : \prod_{y,z:A} \prod_{q:y=z} (x = y) \to (x = z)$$

In other words, given y and z and a path from y to z, we want a function that takes paths from x to y to paths from x to z.

Just like the previous proof, we need a

$$d_2(y) : (y = z) \to (y = z)$$

This is a bit trickier in Agda, because we really want to define a curried function

$$(\bullet_2\ q)\ p = p \bullet q$$

However, we also want the type to be exactly the same as the types of the other constructions. Hence, we will use a twist map.

```
_•₂_ : ∀ {i} {A : Type i} {x y z : A} → (x == y) → (y == z) → (x == z)
_•₂_ = twist concat2 where
    concat2 : ∀ {i} {A : Type i} {x y z : A} → (y == z) → (x == y) → (x == z)
    concat2 {i} {A} {x} {_} {_} = ind== D d where
      D : (y z : A) → (q : y == z) → Type i
      D y z _ = x == y → x == z

      d : (y : A) → D y y refl
      d _ = λ q → q
    twist : ∀ {i} {A : Type i} {x y z : A} → ((y == z) → (x == y) →
        (x == z)) → ((x == y) → (y == z) → (x == z))
    twist f = λ p → λ q → f q p
```

Note that these two constructions use path induction to reduce one side or the other to the "identity" path (in the first case refl_x and in the second case refl_y). We can also do double induction to reduce both p and q to the refl_x and refl_y.

We begin with the same type family as the first proof:

$$D_1 : \prod_{x,y:A} \prod_{p:x=y} (y = z) \to (x = z)$$

but we now wish to find a different inhabitant

$$d_1'(x) : (x = z) \to (x = z)$$

We will use path induction to construct d_1'. We introduce a family:

$$E : \prod_{x,z:A} \prod_{q:x=z} (x = z)$$

we now need

$$e(x) : (x = x)$$

which is gotten quite easily:

$$e(x) = \mathsf{refl}_x$$

```
_∙₃_ : ∀ {i} {A : Type i} {x y z : A} → x == y → y == z → x == z
_∙₃_ {i} {A} {_} {_} {z} = ind== D d where
    D : (x y : A) → (p : x == y) → Type i
    D x y _ = y == z → x == z

    d : (x₁ : A) → D x₁ x₁ refl
    d _ = ind== E e where
      E : (x z : A) (q : x == z) → Type i
      E x z _ = x == z

      e : (x : A) → E x x refl
      e _ = refl
```

We now want to show that these constructions are pairwise equal. By this, we mean "propositional equality" - hence we must find paths between each pair of constructions.

In each case, we perform a double induction on paths, first reducing p to refl, and then reducing q to refl.

```
▪₁=▪₂ : ∀ {i} {A : Type i} {x y z : A}
    (p : x == y) (q : y == z) → p ▪₁ q == p ▪₂ q
▪₁=▪₂ {i} {A} {_} {_} {z} = ind== D d where
    D : (x y : A) → x == y → Type i
    D _ y p = (q : y == z) →  p ▪₁ q == p ▪₂ q

    d : (x : A) → D x x refl
    d _ = ind== E e where
      E : (y₁ z₁ : A) → y₁ == z₁ → Type i
      E _ _ q = refl ▪₁ q == refl ▪₂ q

      e : (x₁ : A) → E x₁ x₁ refl
      e _ = refl

▪₂=▪₃ : ∀ {i} {A : Type i} {x y z : A}
    (p : x == y) (q : y == z) →  p ▪₂ q == p ▪₃ q
▪₂=▪₃ {i} {A} {x} {y} {z} = ind== D d where
    D : (x y : A) → x == y → Type i
    D _ y p = (q : y == z) → p ▪₂ q == p ▪₃ q

    d : (x : A) → D x x refl
    d x = ind== E e where
      E : (y₁ z₁ : A) → y₁ == z₁ → Type i
      E _ _ q = refl ▪₂ q == refl ▪₃ q

      e : (x₁ : A) → E x₁ x₁ refl
      e _ = refl -- : concat2' refl refl == concat3' refl refl


▪₁=▪₃ : ∀ {i} {A : Type i} {x y z : A} (p : x == y) (q : y == z) → p ▪₁ q == p ▪₃ q
▪₁=▪₃ {i} {A} {_} {_} {z} = ind== D d where
    D : (x y : A) → x == y → Type i
    D x y p = (q : y == z) →  p ▪₁ q == p ▪₃ q

    d : (x : A) → D x x refl
    d _ = ind== E e where
      E : (y z : A) → (q : y == z) → Type i
      E _ _ q =  refl ▪₁ q == refl ▪₃ q

      e : (y : A) → E y y refl
      e _ = refl -- : concat1' refl refl == concat3' refl refl
```

□

2

Lemma (2.2.1)**.** *The three equalities of proofs constructed in the previous exercise form a commutative triangle. In other words, if the three definitions of concatenation are denoted by $(p \bullet_1 q)$, $(p \bullet_2 q)$, and $(p \bullet_3 q)$, then the concatenated equality*

$$(p \bullet_1 q) = (p \bullet_2 q) = (p \bullet_3 q)$$

is equal to the equality

$$(p \bullet_1 q) = (p \bullet_3 q)$$

Proof. Despite the fact that we're working with the somewhat myserious type of "equalities of equalities", this remains a statement about the propositional equality of two paths. The only tool we have for establishing such an equality is path induction.

First, we fix the definition of concatenation:

```
_•_ = _•₁_
```

We must now show that, for all paths p, q, the proof that $p \bullet_1 q$ is equal to $p \bullet_2 q$ followed by the proof that $p \bullet_2 q$ is equal to $p \bullet_3 q$ is equal to the proof that $p \bullet_1 q$ is equal to $p \bullet_3 q$.

This is exactly expressed in the following type signature:

Since the theorem is quantified over two paths, we shall do double path induction. So, it really just boils down to the theorem being true when both p and q are the identity.

```
concat-commutative-triangle : ∀ {i} {A : Type i} {x y z : A} (p : x == y) (q : y == z) →
    (•₁=•₂ p q) • (•₂=•₃ p q) == •₁=•₃ p q

concat-commutative-triangle {i} {A} {_} {_} {z} = ind== D d where
    D : (x y : A) → x == y → Type i
    D _ y p = (q : y == z) →
      (•₁=•₂ p q) • (•₂=•₃ p q) == •₁=•₃ p q

    d : (x : A) → D x x refl
    d _ = ind== E e where
      E : (y z : A) → (q : y == z) → Type i
      E _ _ q =
        (•₁=•₂ refl q) • (•₂=•₃ refl q) == •₁=•₃ refl q

      e : (y : A) → E y y refl
      e _ = refl
```

$\square$

At this point, it might be helpful to review the definitions of the different concatenation functions. In particular, $\mathsf{refl} \cdot \mathsf{refl} \equiv \mathsf{refl}$ where $\cdot$ is any of $\cdot_1$, $\cdot_2$, or $\cdot_3$.

3

4

Let's try to do it one stage at a time:

- a 0-path is a point in A.
- a 1-path is a path between 0-paths.

```
0-path : ∀ {i} (A : Type i) -> Type i
0-path A = A

1-path : ∀ {i} (A : Type i) -> Type i
1-path A = Σ (0-path A) (λ a -> Σ (0-path A) (λ b -> (a == b)))

-- the boundary of a 1-path is a pair of 0-paths
δ1 : ∀ {i} {A : Type i} -> (p : 1-path A) -> (0-path A) × (0-path A)
δ1 (a , (b , p)) = (a , b)
```

The boundary of a 0-path is somewhat mysterious, so we shall leave it undefined.

We now would like to define a 2-path as a path between 1-paths. However, two arbitrary 1-paths look like this:

$$a \xrightarrow{p} b$$

$$a' \xrightarrow{q} b'$$

That is, p and q are paths between different points. Hence, a path between p and q doesn't make sense. That is, it's not well typed.

However, suppose we have paths x, y as follows:

$$\begin{tikzcd} a \arrow[r, "p"] \arrow[d, "x"'] & b \arrow[d, "y"] \\ a' \arrow[r, "q"] & b' \end{tikzcd}$$

It would certainly make sense to ask for a path of type $p \cdot y = x \cdot q$.

So, it seems that to define 2-paths, we need pairs of 1-paths together with vertical paths like x and y above. So we'll define it as a Σ-type:

$$\sum_{p,q} \sum_{x:src(p)=src(q)} \sum_{y:dst(p)=dst(q)} p \cdot y = x \cdot q$$

We will write down some helper functions and then formalize this:

```
-- Some convenience functions!
src : ∀ {i} {A : Type i} {a : A} {b : A} -> a == b -> A
src {_} {_} {a} {_} p = a

dst : ∀ {i} {A : Type i} {a : A} {b : A} -> a == b -> A
dst {_} {_} {_} {b} p = b

map : ∀ {i} {A : Type i} (p : (1-path A)) -> (fst p) == (fst (snd p))
map p = snd (snd p)

2-path : ∀ {i} (A : Type i) -> Type i
2-path A = Σ (1-path A) λ p -> Σ (1-path A) λ q →
  Σ ((src (map p)) == (src (map q))) λ x → Σ ((dst (map p)) == (dst (map q))) λ y →
    x • (map q) == (map p) • y

-- The boundary of a 2 path as a pair of 1 paths
δ2 : ∀ {i} {A : Type i} -> 2-path A -> (1-path A) × (1-path A)
δ2 {i} {A} (p , (q , (x , (y , α)))) = p , q
```

A boundary of a 2-path can be thought of as a loop. We can formalize this:

```
-- Definition of inverses. This should be put somewhere else.
inverse : ∀ {i} {A : Type i} {a : A} {b : A} -> a == b -> b == a
inverse {i} {A} = ind== D d where
  D : (a b : A) (p : a == b) → Type i
  D a b _ = b == a
  d : (x : A) → D x x refl
  d _ = refl

-- Just to be cute - the boundary of a 2 path as a loop
δ2-loop : ∀ {i} {A : Type i} -> 2-path A -> 1-path A
δ2-loop (p , (q , (x , (y , α)))) =
  (src (map p) , (src (map p) ,
    ((x • (map q)) • (inverse y)) • (inverse (map p))))
```

If one tries to continue in this manner, the Σ-types will become rather large! So it would be nice to appeal to some kind of recursion at this point.

Luckily, it turns out that equality of inhabitants of Σ-types contain all the lower dimensional equalities to make this work!

```
n-path : ∀ {i} {A : Type i} -> ℕ -> Type i
n-path {i} {A} 0 = A
n-path {i} {A} (S n) = Σ (n-path {i} {A} n) λ p -> Σ (n-path n) λ q -> p == q

δ : ∀ {i} {A : Type i} -> (n : ℕ) ->
  (n-path {i} {A} (S n)) -> (n-path n) × (n-path n)
δ n p = fst p , fst (snd p)
```

This is not evidently geometric. To make the connection, we need to use some facts about equalities of sigma types.

```
Σ== : ∀ {i} {A : Type i} {P : A -> Type i}
  -> {w : Σ A P} -> {w' : Σ A P} -> w == w'
    -> (Σ ((fst w) == (fst w')) λ p
      -> (transport {i} {i} {A} P p (snd w)) == (snd w'))
Σ== {i} {A} {P} = ind== D d where
  D : (w : Σ A P) -> (w' : Σ A P) -> w == w' → Type i
  D w w' _ = (Σ ((fst w) == (fst w')) λ p
             -> (transport {i} {i} {A} P p (snd w)) == (snd w'))
  d : (w : Σ A P) → D w w refl
  d _ = refl , refl

Σ==-inv : ∀ {i} {A : Type i} {P : A -> Type i}
  -> {w : Σ A P} -> {w' : Σ A P}
    -> (Σ ((fst w) == (fst w')) λ p
      -> (transport {i} {i} {A} P p (snd w)) == (snd w'))
        -> w == w'
Σ==-inv {i} {A} {P} {w} {w'} α = p-ind (snd w) (snd w') q where
  p = fst α
  q = snd α
  p-ind = ind== D d p where
    D : (w_1 w'_1 : A) -> (p : w_1 == w'_1) -> Type i
    D w_1 w'_1 p = Π (P w_1) λ w_2 -> Π (P w'_1) λ w'_2
                -> (q : (transport P p w_2) == w'_2) -> (w_1 , w_2) == (w'_1 , w'_2)
    d : (x : A) -> D x x refl
    d x x_1 x_2 q = ind== E e q where
      E : (w_2 w_2' : (P x)) -> (q : (transport P refl w_2) == w_2') -> Type i
      E w_2 w_2' q = (x , w_2) == (x , w_2')
      e : (y : (P x)) -> E y y refl
      e _ = refl
```

Lemma. *If $n > 1$, then the boundary of an n-path is a closed $(n-1)$-path.*

Proof. The following code corresponds to this diagram:

$$\begin{tikzcd} a \arrow[r, "p"] \arrow[d, "x"] & b \\ a' \arrow[r, bend left, "p'"] \arrow[r, "q"'] & b' \end{tikzcd}$$

```
δ' : ∀ {i} {A : Type i} -> {n : ℕ}
    -> (n-path {i} {A} (S (S n))) -> (n-path (S n))
δ' {i} {_} {n} α = a' , ( a' , q • (inverse p') ) where
   -- first, unpack α
   a : n-path n
   a = fst (fst α)
   b : n-path n
   b = fst (snd (fst α))
   p : a == b
   p = snd (snd (fst α))

   a' : n-path n
   a' = fst (fst (snd α))
   b' : n-path n
   b' = fst (snd (fst (snd α)))
   q : a' == b'
   q = snd (snd (fst (snd α)))

   -- Apply first sigma equality
   P : n-path n -> Type i
   P p = Σ (n-path n) λ q -> p == q
   t : Σ (a == a') λ x -> (transport P x (b , p)) == (b' , q)
   t = Σ== (snd (snd α))
   x : a == a'
   x = fst t

   -- Apply second (nested) sigma equality
   P' : n-path n -> Type i
   P' c = a' == c
   y' : Σ (n-path n) λ c -> a' == c
   y' = transport P x (b , p)
   t' : y' == (b' , q)
   t' = snd t
   t'' : Σ ((fst y') == b') λ y -> (transport P' y (snd y')) == q
   t'' = Σ== t'
   y : fst y' == b'
   y = fst t''
   α' : (transport P' y (snd y')) == q
```

```
α' = snd t''
p' : a' == b'
p' = src α'
```

□

5

6

Lemma. *Let $p : x = y$ in A. Then for all $z : A$, the function $p \cdot - : (y = z) \to (x = z)$ is an equivalence.*

Proof. An obvious candidate for a quasi-inverse would be a map that concatinates with inverse(p).

We need to use the groupoid laws for ·, as well as whiskering over higher paths.

Here are some groupoid laws:

```
·-inv-l : ∀ {i} {A : Type i} {a b : A} -> (p : a == b) -> ((inverse p) · p) == refl
·-inv-l {i} {A} {a} {b} = ind== D d where
  D : (a b : A) -> (p : a == b) → Type i
  D _ _ p = ((inverse p) · p) == refl
  d : (x : A) → D x x refl
  d _ = refl

·-inv-r : ∀ {i} {A : Type i} {a b : A} -> (p : a == b) ->  p · (inverse p) == refl
·-inv-r {i} {A} {a} {b} = ind== D d where
  D : (a b : A) -> (p : a == b) → Type i
  D _ _ p = p · (inverse p) == refl
  d : (x : A) → D x x refl
  d _ = refl

·-func : ∀ {i} {A : Type i} {a : A} {b : A} {c : A} -> (p : a == b) -> (b == c) -> (a == c)
·-func p q = p · q

·-id-l : ∀ {i} {A : Type i} {a b : A} -> (p : a == b) -> refl · p == p
·-id-l {i} {A} {_} {_} = ind== D d where
  D : (a b : A) -> (p : a == b) → Type i
  D _ _ p = (refl · p) == p
  d : (x : A) → D x x refl
  d _ = refl

·-id-r : ∀ {i} {A : Type i} {a b : A} -> (p : a == b) -> p · refl == p
```

```
•-id-r {i} {A} {_} {_} = ind== D d where
  D : (a b : A) -> (p : a == b) → Type i
  D _ _ p = p • refl == p
  d : (x : A) → D x x refl
  d _ = refl

•-assoc : ∀ {i} {A : Type i} {w x y z : A}
  -> (p : w == x) -> (q : x == y) -> (r : y == z)
    -> p • (q • r) == (p • q) • r
•-assoc {i} {A} {_} {_} {y} {z} = ind== D d where
  D : (w x : A) -> (p : w == x) -> Type i
  D _ x p = (q : x == y) -> (r : y == z) -> p • (q • r) == (p • q) • r
  d : (x : A) -> D x x refl
  d _ = ind== E e where
    E : (x y : A) -> (q : x == y) -> Type i
    E _ y q = (r : y == z) -> refl • (q • r) == (refl • q) • r
    e : (x : A) -> E x x refl
    e _ = ind== F f where
      F : (y z : A) -> (r : y == z) -> Type i
      F _ z r = refl • (refl • r) == (refl • refl) • r
      f : (x : A) -> F x x refl
      f _ = refl
```

And whiskering for 2-paths (really, $n+2$ paths...)

```
whisk-r : ∀ {i} {A : Type i} {x y z : A} {p p' : x == y} -> (q : y == z)
  -> (p == p') -> ((p • q) == (p' • q))
whisk-r {i} {_} {x} {y} {z} {_} {_} q = ind== D d where
  D : (p p' : x == y) -> (α : p == p') -> Type i
  D p p' α =  ((p • q) == (p' • q))
  d : (p : x == y) -> D p p refl
  d _ = refl

whisk-l : ∀ {i} {A : Type i} {x y z : A} {q q' : y == z} -> (p : x == y)
  -> (q == q') -> ((p • q) == (p • q'))
whisk-l {i} {_} {x} {y} {z} {_} {_} p = ind== D d where
  D : (q q' : y == z) -> (β : q == q') -> Type i
  D q q' β =  ((p • q) == (p • q'))
  d : (q : y == z) -> D q q refl
  d _ = refl
```

We now define the quasi inverse to $p \cdot -$ as $p^{-1} \cdot -$. To do this, we need homotopies from $(p \cdot -) \circ (p^{-1} \cdot -) \sim id$ and $(p^{-1} \cdot -) \circ (p \cdot -) \sim id$. By definition, $(p \cdot -) \circ (p^{-1} \cdot -) \equiv (p \cdot p^{-1} \cdot -)$, so we really just need a 2-path $p \cdot p^{1} = \mathsf{refl}$ (and a 2-path for the symmetric case). This follows from the groupoid laws above:

```
∙-qinv : ∀ {i} {A : Type i} {x y z : A}
  -> (p : x == y) -> (q-inv (_∙_ {i} {A} {x} {y} {z} p))
∙-qinv p = _∙_ (inverse p) ,
  ( (λ q -> ((∙-assoc p (inverse p) q) ∙ whisk-r q (∙-inv-r p)) ∙ (∙-id-l q))
     , (λ q →  ((∙-assoc (inverse p) p q) ∙ whisk-r q (∙-inv-l p)) ∙  (∙-id-l q)))
```

Now, we simply observe that every quasi-inverse is an equivalence.

```
∙-equiv : ∀ {i} {A : Type i} {x y z : A}
  -> (p : x == y) -> (is-equiv'  (_∙_ {i} {A} {x} {y} {z} p))
∙-equiv p = q-inv-to-equiv' (_∙_ p) (∙-qinv p)
```

□

7

8

9

10

11

Oh boy! Homotopy pullbacks!

First, let's define a homotopy commutative diagram. We will stick to the notation used in the book.

The following Σ-type is the type of all pullback squares given types P, A, B, C.

$$\begin{tikzcd} P \arrow[r, "h"] \arrow[d, "k"] & A \arrow[d, "f"] \\ B \arrow[r, "g"] & C \end{tikzcd}$$

```
com-sq : ∀ {i} {A B C : Type i} -> (f : A -> C) -> (g : B -> C)
  -> (P : Type i) -> Type i
com-sq {_} {A} {B} {_} f g P =
  Σ (P -> A) λ h -> Σ (P -> B) λ k -> (f ∘ h) == (g ∘ k)
```

A pullback square is a commutative square together with a certain equivalence. The book defines this in terms of a "canonical pullback" that is defined in terms of composition. This is analogous to defining pullbacks in a category $\mathcal{C}$ in terms of presheaves over $\mathcal{C}$. A diagram is a pullback square if the upper left corner represents a functor that is equivalent to the pullback of the diagram.

```
open import FunExt

precomp : ∀ {i} {A B C : Type i} -> (A -> B) -> (B -> C) -> (A -> C)
precomp f g = g ∘ f

precomp-happly : ∀ {i} {A B X : Type i} -> (f : X -> A) -> (g g' : A -> B)
  -> (α : (g == g'))
    -> (x : X)
      -> (happly (g ∘ f) (g' ∘ f) (ap (precomp f) α) x) == (happly g g' α) (f x)
precomp-happly {i} {A} {B} {X} f g g' α = ind== D d α where
  D : (g g' : A -> B) -> (g == g') -> Type i
  D g g' α = (x : X)
    -> (happly (g ∘ f) (g' ∘ f) (ap (precomp f) α) x) == (happly g g' α) (f x)
  d : (g : A -> B) -> D g g refl
  d g x = refl

homotopy-square : ∀ {i} {A B : Type i} -> (f g : A -> B) -> (H : f ~ g)
  -> (x y : A) -> (p : x == y) -> ((H x) • (ap g p)) == ((ap f p) • (H y))
homotopy-square {i} {A} {B} f g H x y = ind== D d where
  D : (x y : A) -> (p : x == y) -> Type i
  D x y p = ((H x) • (ap g p)) == ((ap f p) • (H y))
  d : (x : A) -> D x x refl
  d x = •-id-r (H x)

ap-id : ∀ {i} {A : Type i} {x y : A} -> (p : x == y) -> ap id p == p
ap-id {i} {A} p = ind== D d p where
  D : (x y : A) -> (x == y) -> Type i
  D _ _ p = ap id p == p
  d : (x : A) -> D x x refl
  d _ = refl

homotopy-equiv-square : ∀ {i} {A : Type i} -> (f : A -> A) -> (H : f ~ id)
  -> (x : A) -> H (f x) == ap f (H x)
homotopy-equiv-square f H x = (inverse (•-id-r (H (f x)))
                              • (inverse (whisk-l (H (f x)) (•-inv-r (H x)))
                              • (•-assoc (H (f x)) (H x) (inverse (H x))
                              • whisk-r (inverse (H x))
                                  (inverse (whisk-l (H (f x)) (ap-id (H x)))
                              • homotopy-square f id H (f x) x (H x)))))
                              • (inverse (•-assoc (ap f (H x)) (H x) (inverse (H x)))
                              • (whisk-l (ap f (H x)) (•-inv-r (H x))
```

```
                          · ·-id-r (ap f (H x))))

∘-app : ∀ {i} {A B C : Type i} {x y : A} -> (p : x == y) -> (f : A -> B) -> (g : B -> C)
  -> (ap g (ap f p)) == ap (g ∘ f) p
∘-app {i} {A} {_} {_} {x} {y} p f g = ind== D d p where
  D : (x y : A) -> (x == y) -> Type i
  D x y p = (ap g (ap f p)) == ap (g ∘ f) p
  d : (x : A) -> D x x refl
  d x = refl

-- Theorem 2.4.3 from the book
q-inv-to-equiv : ∀ {i} {A B : Type i} -> (f : A -> B)
  -> (q-inv f) -> (is-equiv f)
q-inv-to-equiv {i} {A} {B} f (g , (ε , η)) =
  record { g = g ; ε = ε' ; η = η ; τ = λ a -> (inverse (τ a)) } where
  ε' : (b : B) -> f (g b) == b
  ε' b = (inverse (ε (f (g b))) ) · (ap f (η (g b)) · ε b)
  η-ε-square : (a : A) ->
    ap f (η (g (f a))) · (ε (f a))  == ε (f (g (f a))) · (ap f (η a))
  η-ε-square a =
    whisk-r (ε (f a)) ((ap (ap f) (homotopy-equiv-square {i} {A} (g ∘ f) η a))
                      · (∘-app (η a) (g ∘ f) (f)) )
    · (whisk-r (ε (f a)) (inverse (∘-app (η a) f (f ∘ g)))
    · (inverse (homotopy-square (f ∘ g) id ε (f (g (f a))) (f a) (ap f (η a)))
    · whisk-l (ε (f (g (f a)))) (ap-id (ap f (η a)))  ))
  τ : (a : A) -> ε' (f a) == ap f (η a)
  τ a = whisk-l (inverse (ε (f (g (f a))))) (η-ε-square a)
        · (·-assoc (inverse (ε (f (g (f a))))) (ε (f (g (f a)))) (ap f (η a))
        · (whisk-r (ap f (η a)) (·-inv-l (ε (f (g (f a)))))
        · ·-id-l (ap f (η a))))

∘-functor : ∀ {i} {A B C : Type i}
  -> (f : A -> B) -> (g : B -> C) -> (g' : B -> C)
    -> (g == g') -> (g ∘ f) == (g' ∘ f)
∘-functor f g g' = ap (precomp f)

happly-path : ∀ {i} {A B : Type i}
  -> (f g : A -> B) -> (α : f == g) -> (β : f == g)
    -> (happly f g α) == (happly f g β) -> α == β
happly-path f g α β ψ = (inverse (h-inv-h f g α)) · (ψ' · h-inv-h f g β)  where
    ψ' : (funext f g (happly f g α)) == (funext f g (happly f g β))
    ψ' = (ap (funext f g)) ψ

p-map : ∀ {i} {A B C : Type i} -> (f : A -> C) -> (g : B -> C)
  -> (P : Type i) -> (X : Type i) -> (com-sq f g P)
    -> (X -> P) -> (com-sq f g X)
```

```
p-map f g P X sq l = h ∘ l , (k ∘ l , ap (precomp l) α ) where
    h = fst sq
    k = fst (snd sq)
    α = snd (snd sq)

open import Agda.Primitive using (lsuc)

-- A square (P, _, _) over f,g is a pullback if for all types X,
-- the induced function from maps from X to P to commutative squares
-- over f,g is an equivalence.
is-pullback : ∀ {i} {A B C : Type i}
  -> (f : A -> C) -> (g : B -> C) -> (P : Type i)
    -> (com-sq f g P) -> Type (lsuc i)
is-pullback {i} f g P α = Π (Type i) λ X -> is-equiv (p-map f g P X α)

-- pullback type of f, g
pullback : ∀ {i} {A B C : Type i} -> (f : A -> C) -> (g : B -> C) -> Type i
pullback {i} {A} {B} f g = Σ A λ a -> Σ B λ b -> (f a) == (g b)

-- pullback type together with projection maps
pullback-sq : ∀ {i} {A B C : Type i} -> (f : A -> C) -> (g : B -> C)
  -> (com-sq f g (pullback f g))
-- construct a homotopy and use function extensionality
pullback-sq {_} {A} {B} f g = h , (k , (funext  (f ∘ h) (g ∘ k) α )) where
  P = pullback f g
  h : P -> A
  h = fst
  k : P -> B
  k p = fst (snd p)
  α : Π P λ p -> (f (h p)) == (g (k p))
  α = λ p → snd (snd p)

-- We need to factor maps from X to f,g through P
factor : ∀ {i} {A B C : Type i} {f : A -> C} {g : B -> C} {X : Type i}
  -> (com-sq f g X) -> (X -> (pullback f g))
factor {_} {_} {_} {_} {f} {g} {_} (h' , (k' , α')) x =
  h' x , (k' x , (happly (f ∘ h') (g ∘ k') α') x)

pullback-is-pullback : ∀ {i} {A B C : Type i} -> (f : A -> C) -> (g : B -> C)
  -> (is-pullback f g (pullback f g) (pullback-sq f g))
pullback-is-pullback {_} {A} {B} f g X =
  (q-inv-to-equiv (p-map f g P X P-sq) p-map-q-inv) where

   P = pullback f g
   P-sq = pullback-sq f g
   h = fst P-sq
```

```
k = fst (snd P-sq)
α = snd (snd P-sq)
α' : Π P λ p -> (f (h p)) == (g (k p))
α' = λ p → snd (snd p)

p-map-q-inv : q-inv (p-map f g (pullback f g) X (pullback-sq f g))
p-map-q-inv = factor , (ε , η) where
  -- components of the quasi-inverse
  ε : (sq : (com-sq f g X)) -> (p-map f g P X P-sq (factor sq) == sq)
  ε sq = Σ==-inv (refl , (Σ==-inv (refl ,
           (happly-path (f ∘ h') (g ∘ k') (snd (snd sq')) (snd (snd sq)) β ) ))) where
    l : X -> P
    l = factor sq
    h' = fst sq
    k' = fst (snd sq)
    sq' = p-map f g P X P-sq l

    ψ : (x : X) -> ((happly (f ∘ h') (g ∘ k') (snd (snd sq'))) x == (happly (f ∘ h) (g ∘
    ψ = precomp-happly l (f ∘ h) (g ∘ k) α

    φ : (x : X) -> ((happly (f ∘ h) (g ∘ k) α) (l x)) == α' (l x)
    φ x = (happly (happly (f ∘ h) (g ∘ k) α) α' (h-h-inv (f ∘ h) (g ∘ k) α')) (l x)

    β : (happly (f ∘ h') (g ∘ k') (snd (snd sq'))) == (happly (f ∘ h') (g ∘ k') (snd (snd
    β = funext
        (happly (f ∘ h') (g ∘ k') (snd (snd sq')))
        (happly (f ∘ h') (g ∘ k') (snd (snd sq)))
        λ x -> ψ x · φ x

  η : (l : X -> P) -> factor (p-map f g P X P-sq l) == l
  η l = funext l' l β where
    l' = factor (p-map f g P X P-sq l)
    γ' : Π X λ x -> (f (h (l x))) == (g (k (l x)))
    γ' x = snd (snd (l' x))
    γ : Π X λ x -> (f (h (l x))) == (g (k (l x)))
    γ x = snd (snd (l x))
    ψ : (x : X) -> (γ' x == (happly (f ∘ h) (g ∘ k) α) (l x))
    ψ = precomp-happly l (f ∘ h) (g ∘ k) α

    φ : (x : X) -> (happly (f ∘ h) (g ∘ k) α) (l x) == γ x
    φ x = (happly (happly (f ∘ h) (g ∘ k) α)
                (λ p → snd (snd p))
                  (h-h-inv (f ∘ h) (g ∘ k) (λ p -> (snd (snd p))))) (l x)

    β : Π X λ x -> l' x == l x
    β = λ x → Σ==-inv ( refl , (Σ==-inv ( refl , ψ x · φ x )))
```

12

13

Show that $(2 \simeq 2) \simeq 2$.

First, we must define equivalence of types.

```
_≃_ : ∀ {i} -> (A B : Type i) -> Type i
_≃_ A B = Σ (A -> B) λ f -> is-equiv(f)
```

I guess we should define **2** as well:

```
data Two : Type₀ where
  0₂ : Two
  1₂ : Two

rec₂ : ∀ {i} {C : Type i} -> C -> C -> Two -> C
rec₂ c₀ c₁ 0₂ = c₀
rec₂ c₀ c₁ 1₂ = c₁

ind₂ : ∀ {j} -> (C : Two -> Type j) -> C 0₂ -> C 1₂ -> Π Two C
ind₂ C c₀ c₁ 0₂ = c₀
ind₂ C c₀ c₁ 1₂ = c₁
```

While we're at it, I suppose we should prove that anything in **2** is equal to 0_2 or 1_2:

```
elems-of-two : (x : Two) -> Coprod (x == 0₂) (x == 1₂)
elems-of-two x = ind₂ D (inl refl) (inr refl) x where
  D : Two -> Type _
  D x = Coprod (x == 0₂) (x == 1₂)
```

Also, we would like to know that $0 \neq 1$. With the above theorem, this would imply that **2** has two distinct path components.

To do this, we will show that if $0 = 1$ then the empty type is inhabited. We will need the encode-decode method for **2**. (Actually, only encode is necessary, but I'll do both for the same of completeness.)

```
--encode-decode for Two
code₂ : Two -> Two -> Type₀
```

```
code₂ 0₂ 0₂ = Unit
code₂ 1₂ 1₂ = Unit
code₂ 0₂ 1₂ = Empty
code₂ 1₂ 0₂ = Empty

r₂ : (x : Two) -> code₂ x x
r₂ 0₂ = unit
r₂ 1₂ = unit

encode₂ : {x y : Two} → (x == y) -> code₂ x y
encode₂ {x} {y} p = transport (code₂ x) p (r₂ x)

decode₂ : {x y : Two} → code₂ x y -> x == y
decode₂ {0₂} {0₂} =  λ _ → refl
decode₂ {0₂} {1₂} =  λ ()
decode₂ {1₂} {0₂} =  λ ()
decode₂ {1₂} {1₂} =  λ _ → refl

Two-distinct : 0₂ == 1₂ → Empty
Two-distinct p = encode₂ p
```

Now, let's start picking apart automorphisms of **2**.

We must define a map $\mathbf{2} \to \mathbf{2}^{\mathbf{2}}$ and show that it is an equivalence.

Or, by univalence, we could find a path in the universe $\mathbf{2} = \mathbf{2}^{\mathbf{2}}$.

Well, I know how to construct a function, but I'm not sure how to construct a path in the universe (other than using the univalence axiom itself).

First, we'll show that automorphisms cannot send 0 and 1 to equal inhabitants. This is accomplished by using the structure of the equivalence to show that such a path would imply $0 = 1$.

```
Two-auto-distinct : {f : Two ≃ Two}
  -> ((fst f) 0₂) == ((fst f) 1₂) -> Empty
Two-auto-distinct {f , (equiv g η ε τ)} p =
  Two-distinct (inverse (η 0₂) • (ap g p • η 1₂))
```

Now we will define automorphisms of **2** corresponding to 0 and 1. As you might imagine, 0 will correspond to the identity and 1 will correspond to the twist map.

```
Two-auto : Two -> (Two ≃ Two)
Two-auto 0₂ = f , q-inv-to-equiv f f-q-inv where
  f : Two -> Two
  f = rec₂ 0₂ 1₂
```

```
  f-homotopy : (f ∘ f) ~ id
  f-homotopy x = is-one-or-other p where
    p : Coprod (x == 0₂) (x == 1₂)
    p = (elems-of-two x)
    is-one-or-other : Coprod (x == 0₂) (x == 1₂) → (f (f x)) == x
    is-one-or-other (inl p0) = (ap (f ∘ f) p0) • inverse p0
    is-one-or-other (inr p1) = ap (f ∘ f) p1 • inverse p1
  f-q-inv : q-inv f
  f-q-inv = f , (f-homotopy , f-homotopy)

Two-auto 1₂ = f , q-inv-to-equiv f f-q-inv where
  f : Two -> Two
  f = rec₂ 1₂ 0₂
  f-homotopy : (f ∘ f) ~ id
  f-homotopy x =  is-one-or-other p where
    p : Coprod (x == 0₂) (x == 1₂)
    p = (elems-of-two x)
    is-one-or-other : Coprod (x == 0₂) (x == 1₂) → (f (f x)) == x
    is-one-or-other (inl p0) = (ap (f ∘ f) p0) • inverse p0
    is-one-or-other (inr p1) = ap (f ∘ f) p1 • inverse p1
  f-q-inv : q-inv f
  f-q-inv = f , (f-homotopy , f-homotopy)
```

We would like to construct a quasi inverse to this map to show that it is an equivalence. The inverse will take an automorphism to its evaluation on 0.

```
Two-auto-inv : (Two ≃ Two) -> Two
Two-auto-inv (f , _) = f 0₂
```

Now comes the more difficult part. We need two homotopies to demonstrate that this is indeed a quasi-inverse.

One is quite easily obtained by induction over **2**.

```
Two-η : (x : Two) -> (Two-auto-inv (Two-auto x)) == x
Two-η = ind₂ D refl refl where
  D : (x : Two) -> Type₀
  D x = (Two-auto-inv (Two-auto x)) == x
```

The other direction is harder for a few reasons. We essentially need to do case analysis on $(f0)$ where f is an automorphism. To this, need to prove some lemmas of the form "if x is equal to 0, then the above functions applied to x behave as if they were applied to 0".

```
open import Agda.Primitive using (lzero)
```

```
Two-auto-paths : (f : Two ≃ Two) → (fst f) 0₂ == 0₂ → (fst f) 1₂ == 1₂
Two-auto-paths (f , ψ) p = derp image-1₂ where
  image-1₂ = elems-of-two (f 1₂)
  derp : Coprod ((f 1₂) == 0₂) ((f 1₂) == 1₂) → f 1₂ == 1₂
  derp (inl p0) = Empty-elim {lzero} {λ x → f 1₂ == 1₂} (Two-auto-distinct {(f , ψ)} (p • i
  derp (inr p1) = p1

Two-auto-path-0₂ : (x : Two) → (x == 0₂) → (fst (Two-auto x)) 0₂ == 0₂
Two-auto-path-0₂ x p = happly (fst (Two-auto x)) (fst (Two-auto 0₂))
                      (fst (Σ== (ap Two-auto p))) 0₂

Two-auto-path-1₂ : (x : Two) → (x == 1₂) → (fst (Two-auto x)) 0₂ == 1₂
Two-auto-path-1₂ x p = happly (fst (Two-auto x)) (fst (Two-auto 1₂))
                      (fst (Σ== (ap Two-auto p))) 0₂

Two-auto-path-0₂-1₂ : (x : Two) → (x == 0₂) → (fst (Two-auto x)) 1₂ == 1₂
Two-auto-path-0₂-1₂ x p = happly (fst (Two-auto x)) (fst (Two-auto 0₂))
                      (fst (Σ== (ap Two-auto p))) 1₂

Two-auto-path-1₂-1₂ : (x : Two) → (x == 1₂) → (fst (Two-auto x)) 1₂ == 0₂
Two-auto-path-1₂-1₂ x p = happly (fst (Two-auto x)) (fst (Two-auto 1₂))
                      (fst (Σ== (ap Two-auto p))) 1₂
```

Finally, we construct the homotopies of the quasi equivalence:

```
Two-ε : (f : (Two ≃ Two)) -> (Two-auto (Two-auto-inv f)) == f
Two-ε (f , ψ) = Σ==-inv (funext (fst (Two-auto (Two-auto-inv (f , ψ)))) f H , {!!}) where
  H : (x : Two) -> (fst (Two-auto (Two-auto-inv (f , ψ))) x) == (f x)
  H 0₂ = pick-one image-0₂ where
    image-0₂ = elems-of-two (f 0₂)
    pick-one : Coprod ((f 0₂) == 0₂) ((f 0₂) == 1₂)
      → fst (Two-auto (Two-auto-inv (f , ψ))) 0₂ == f 0₂
    pick-one (inl p0) = Two-auto-path-0₂ (f 0₂) p0 • inverse p0
    pick-one (inr p1) = Two-auto-path-1₂ (f 0₂) p1 • inverse p1
  H 1₂ = pick-one image-1₂ where
    image-0₂ = elems-of-two (f 0₂)
    image-1₂ = elems-of-two (f 1₂)
    pick-one : Coprod ((f 1₂) == 0₂) ((f 1₂) == 1₂)
      → fst (Two-auto (Two-auto-inv (f , ψ))) 1₂ == f 1₂
    pick-one (inl p0) = pick-another image-0₂ where
        pick-another : Coprod ((f 0₂) == 0₂) ((f 0₂) == 1₂)
          → fst (Two-auto (Two-auto-inv (f , ψ))) 1₂ == f 1₂
        pick-another (inl q0) = Empty-elim {lzero}
```

```
          {λ x → fst (Two-auto (Two-auto-inv (f , ψ))) 1₂ == f 1₂}
          (Two-auto-distinct {f , ψ }(q0 • inverse p0))
        pick-another (inr q1) = Two-auto-path-1₂-1₂ (f 0₂) q1 • inverse p0
    pick-one (inr p1) = pick-another image-0₂ where
        pick-another : Coprod ((f 0₂) == 0₂) ((f 0₂) == 1₂)
          → fst (Two-auto (Two-auto-inv (f , ψ))) 1₂ == f 1₂
        pick-another (inl q0) = Two-auto-path-0₂-1₂ (f 0₂) q0 • inverse p1
        pick-another (inr q1) = Empty-elim {lzero}
          {λ x → fst (Two-auto (Two-auto-inv (f , ψ))) 1₂ == f 1₂}
          (Two-auto-distinct {f , ψ} (q1 • inverse p1))

  τ :  transport is-equiv
                 (funext (fst (Two-auto (Two-auto-inv (f , ψ)))) f H)
                 (snd (Two-auto (f 0₂)))
       == ψ
  τ = {!!}

Two-auto-is-equiv : is-equiv Two-auto
Two-auto-is-equiv = q-inv-to-equiv Two-auto (Two-auto-inv ,  (Two-ε , Two-η))
```

14

Suppose the equality reflection rule holds for a type A. Now show that A is a set.