Exercises from the *HoTT Book*

John Dougherty

September 8, 2014

Introduction

The following are solutions to exercises from *Homotopy Type Theory: Univalent Foundations of Mathematics*. The Coq code given alongside the by-hand solutions requires the HoTT version of Coq, available at the HoTT github repository. It will be assumed throughout that it has been imported by

```
Require Export HoTT.
```

Contents

1 Type Theory

Exercise 1.1 (p. 56) Given functions $f : A \to B$ and $g : B \to C$, define their composite $g \circ f : A \to C$. Show that we have $h \circ (g \circ f) \equiv (h \circ g) \circ f$.

Solution Define $g \circ f :\equiv \lambda(x:A).\, g(f(x))$. Then if $h : C \to D$, we have

$$h \circ (g \circ f) \equiv \lambda(x:A).\, h((g \circ f)x) \equiv \lambda(x:A).\, h((\lambda(y:A).\, g(fy))x) \equiv \lambda(x:A).\, h(g(fx))$$

and

$$(h \circ g) \circ f \equiv \lambda(x:A).\, (h \circ g)(fx) \equiv \lambda(x:A).\, (\lambda(y:B).\, h(gy))(fx) \equiv \lambda(x:A).\, h(g(fx))$$

So $h \circ (g \circ f) \equiv (h \circ g) \circ f$. In Coq, we have

```
Definition compose' {A B C : Type} (g : B → C) (f : A → B) := fun x ⇒ g (f x).

Lemma compose'_assoc : ∀ (A B C D : Type) (f : A → B) (g : B→ C) (h : C → D),
  compose' h (compose' g f) = compose' (compose' h g) f.
Proof.
  reflexivity.
Defined.
```

Exercise 1.2 (p. 56) Derive the recursion principle for products $\mathsf{rec}_{A\times B}$ using only the projections, and verify that the definitional equalities are valid. Do the same for Σ-types.

Solution The recursion principle states that we can define a function $f : A \times B \to C$ by giving its value on pairs. Suppose that we have projection functions $\mathsf{pr}_1 : A \times B \to A$ and $\mathsf{pr}_2 : A \times B \to B$. Then we can define a function of type

$$\mathsf{rec}'_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to A \times B \to C$$

in terms of these projections as follows

$$\mathsf{rec}'_{A\times B}(C, g, p) :\equiv g(\mathsf{pr}_1 p)(\mathsf{pr}_2 p)$$

or, in Coq,

```
Section Exercise2a.
  Context {A B : Type}.
  Definition recprd (C : Type) (g : A → B → C) (p : A × B) := g (fst p) (snd p).
```

We must then show that

$$\mathsf{rec}'_{A\times B}(C, g, (a, b)) \equiv g(\mathsf{pr}_1(a, b))(\mathsf{pr}_2(a, b)) \equiv g(a)(b)$$

which in Coq amounts to showing that these this equality is a reflexivity.

```
  Theorem recprd_correct : ∀ (C : Type) (g : A → B → C) (a : A) (b : B),
    recprd C g (a, b) = g a b.
  Proof.
    reflexivity.
  Defined.
End Exercise2a.
Section Exercise2b.
  Context (A : Type).
  Context (B : A → Type).
```

Now for the Σ-types. Here we have a projection

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

and another

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

Define a function of type

$$\mathsf{rec}'_{\sum_{(x:A)} B(x)} : \prod_{C:\mathcal{U}} \left(\prod_{(x:A)} B(x) \to C\right) \to \left(\sum_{(x:A)} B(x)\right) \to C$$

by

$$\mathsf{rec}'_{\sum_{(x:A)} B(x)}(C, g, p) :\equiv g(\mathsf{pr}_1 p)(\mathsf{pr}_2 p)$$

```
  Definition recsm (C : Type) (g : ∀ (x : A), B x → C) (p : {x:A & B x}) :=
    g (p.1) (p.2).
```

We then verify that

$$\mathsf{rec}_{\sum_{(x:A)} B(x)}(C, g, (a, b)) \equiv g(\mathsf{pr}_1(a, b))(\mathsf{pr}_2(a, b)) \equiv g(a)(b)$$

```
Theorem recsm_correct : ∀ (C:Type) (g : ∀ x, B x → C) (a:A) (b:B a),
  recsm C g (a; b) = g a b.
Proof.
  reflexivity.
Defined.
End Exercise2b.
```

Exercise 1.3 (p. 56) Derive the induction principle for products $\mathsf{ind}_{A\times B}$ using only the projections and the propositional uniqueness principle uppt. Verify that the definitional equalities are valid. Generalize uppt to Σ-types, and do the same for Σ-types.

```
Section Exercise3a.
  Context {A B : Type}.
```

Solution The induction principle has type

$$\mathsf{ind}_{A\times B} : \prod_{C:A\times B\to\mathcal{U}} \left(\prod_{(x:A)} \prod_{(y:B)} C((x, y))\right) \to \prod_{z:A\times B} C(z)$$

For a first pass, we can define our new inductor as

$$\lambda C. \lambda g. \lambda z. g(\mathsf{pr}_1 z)(\mathsf{pr}_2 z)$$

However, we have $g(\mathsf{pr}_1 x)(\mathsf{pr}_2 x) : C((\mathsf{pr}_1 x, \mathsf{pr}_2 x))$, so the type of this $\mathsf{ind}'_{A\times B}$ is

$$\prod_{C:A\times B\to\mathcal{U}} \left(\prod_{(x:A)} \prod_{(y:B)} C((x, y))\right) \to \prod_{z:A\times B} C((\mathsf{pr}_1 z, \mathsf{pr}_2 z))$$

To define $\mathsf{ind}'_{A\times B}$ with the correct type, we need the $\mathsf{transport}$ operation from the next chapter. The uniqueness principle for product types is

$$\mathsf{uppt} : \prod_{x:A\times B} ((\mathsf{pr}_1 x, \mathsf{pr}_2 x) =_{A\times B} x)$$

By the transport principle, there is a function

$$(\mathsf{uppt}\, x)_* : C((\mathsf{pr}_1 x, \mathsf{pr}_2 x)) \to C(x)$$

so

$$\mathsf{ind}_{A\times B}(C, g, z) :\equiv (\mathsf{uppt}\, z)_*(g(\mathsf{pr}_1 z)(\mathsf{pr}_2 z))$$

has the right type. In Coq we use eta_prod, which is the name for uppt, then use it with transport to give our $\mathsf{ind}'_{A\times B}$.

```
  Definition indprd (C : A × B → Type) (g : ∀ (x:A) (y:B), C (x, y)) (z : A × B) :=
    (eta_prod z) # (g (fst z) (snd z)).
```

We now have to show that

$$\mathsf{ind}_{A\times B}(C, g, (a, b)) \equiv g(a)(b)$$

Unfolding the left gives

$$\begin{aligned}
\mathsf{ind}_{A\times B}(C,g,(a,b)) &\equiv (\mathsf{uppt}\,(a,b))_*(g(\mathsf{pr}_1(a,b))(\mathsf{pr}_2(a,b)))\\
&\equiv \mathsf{ind}_{=_{A\times B}}(D,d,(a,b),(a,b),\mathsf{uppt}((a,b)))(g(a)(b))\\
&\equiv \mathsf{ind}_{=_{A\times B}}(D,d,(a,b),(a,b),\mathsf{refl}_{(a,b)})(g(a)(b))\\
&\equiv \mathsf{id}_{C((a,b))}(g(a)(b))\\
&\equiv g(a)(b)
\end{aligned}$$

which was to be proved.

```
Theorem indprd_correct :
  ∀ (C : A × B → Type) (g : ∀ (x:A) (y:B), C (x, y)) (a:A) (b:B),
    indprd C g (a, b) = g a b.
Proof.
  reflexivity.
Defined.

End Exercise3a.

Section Exercise3b.
  Context {A : Type}.
  Context {B : A → Type}.
```

For Σ-types, we define

$$\mathsf{ind}'_{\sum_{(x:A)} B(x)} : \prod_{C:(\sum_{(x:A)} B(x))\to\mathcal{U}} \left(\textstyle\prod_{(a:A)}\prod_{(b:B(a))} C((a,b))\right) \to \prod_{p:\sum_{(x:A)} B(x)} C(p)$$

at first pass by

$$\lambda C.\,\lambda g.\,\lambda p.\,g(\mathsf{pr}_1 p)(\mathsf{pr}_2 p)$$

We encounter a similar problem as in the product case: this gives an output in $C((\mathsf{pr}_1 p, \mathsf{pr}_2 p))$, rather than $C(p)$. So we need a uniqueness principle for Σ-types, which would be a function

$$\mathsf{upst} : \prod_{p:\sum_{(x:A)} B(x)} \left((\mathsf{pr}_1 p, \mathsf{pr}_2 p) =_{\sum_{(x:A)} B(x)} p\right)$$

As for product types, we can define

$$\mathsf{upst}((a,b)) :\equiv \mathsf{refl}_{(a,b)}$$

which is well-typed, since $\mathsf{pr}_1(a,b) \equiv a$ and $\mathsf{pr}_2(a,b) \equiv b$. Thus, we can write

$$\mathsf{ind}'_{\sum_{(x:A)} B(x)}(C,g,p) :\equiv (\mathsf{upst}\,p)_*(g(\mathsf{pr}_1 p)(\mathsf{pr}_2 p)).$$

and in Coq,

```
Definition indsm (C : {x:A & B x} → Type) (g : ∀ (a:A) (b:B a), C (a; b)) (p : {x:A & B x}) :=
  (eta_sigma p) # (g (p.1) (p.2)).
```

Now we must verify that

$$\mathsf{ind}'_{\sum_{(x:A)} B(x)}(C,g,(a,b)) \equiv g(a)(b)$$

We have

$$
\begin{aligned}
\mathsf{ind}'_{\Sigma_{(x:A)} B(x)}(C, g, (a,b)) &\equiv (\mathsf{uppt}\,(a,b))_*(g(\mathsf{pr}_1(a,b))(\mathsf{pr}_2(a,b))) \\
&\equiv \mathsf{ind}'_{=_{\Sigma_{(x:A)} B(x)}}(D, d, (a,b), (a,b), \mathsf{uppt}\,(a,b))(g(a)(b)) \\
&\equiv \mathsf{ind}'_{=_{\Sigma_{(x:A)} B(x)}}(D, d, (a,b), (a,b), \mathsf{refl}_{(a,b)})(g(a)(b)) \\
&\equiv \mathsf{id}_{C((a,b))}(g(a)(b)) \\
&\equiv g(a)(b)
\end{aligned}
$$

```
  Theorem indsm_correct :
        ∀ (C : {x:A & B x} → Type)
              (g : ∀ (a:A) (b:B a), C (a; b)) (a : A) (b : B a),
          indsm C g (a; b) = g a b.
  Proof.
    reflexivity.
  Defined.
End Exercise3b.
```

Exercise 1.4 (p. 56) Assuming as given only the *iterator* for natural numbers

$$\mathsf{iter} : \prod_{C:\mathcal{U}} C \to (C \to C) \to \mathbb{N} \to C$$

with the defining equations

$$
\begin{aligned}
\mathsf{iter}(C, c_0, c_s, 0) &:\equiv c_0, \\
\mathsf{iter}(C, c_0, c_s, \mathsf{succ}(n)) &:\equiv c_s(\mathsf{iter}(C, c_0, c_s, n)),
\end{aligned}
$$

derive a function having the type of the recursor $\mathsf{rec}_{\mathbb{N}}$. Show that the defining equations of the recursor hold propositionally for this function, using the induction principle for $\mathbb{N}$.

```
Section Exercise4.
  Variable C : Type.
  Variable c0 : C.
  Variable cs : nat → C → C.
```

Solution Fix some $C : \mathcal{U}$, $c_0 : C$, and $c_s : \mathbb{N} \to C \to C$. $\mathsf{iter}(C)$ allows for the n-fold application of a single function to a single input from C, whereas $\mathsf{rec}_{\mathbb{N}}$ allows each application to depend on n, as well. Since n just tracks how many applications we've done, we can construct n on the fly, iterating over elements of $\mathbb{N} \times C$. So we will use the iterator

$$\mathsf{iter}_{\mathbb{N}\times C} : \mathbb{N} \times C \to (\mathbb{N} \times C \to \mathbb{N} \times C) \to \mathbb{N} \to \mathbb{N} \times C$$

to derive a function

$$\Phi : \prod_{C:\mathcal{U}} C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$$

which has the same type as $\mathsf{rec}_{\mathbb{N}}$.

The first argument of $\mathsf{iter}_{\mathbb{N}\times C}$ is the starting point, which we'll make $(0, c_0)$. The second input takes an element of $\mathbb{N} \times C$ as an argument and uses c_s to construct a new element of $\mathbb{N} \times C$. We can use the first and second elements of the pair as arguments for c_s, and we'll use succ to advance the first argument, representing the number of steps taken. This gives the function

$$\lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)) : \mathbb{N} \times C \to \mathbb{N} \times C$$

for the second input to $\mathsf{iter}_{\mathbb{N}\times C}$. The third input is just n, which we can pass through. Plugging these in gives

$$\mathsf{iter}_{\mathbb{N}\times C}\big((0, c_0), \lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)), n\big) : \mathbb{N} \times C$$

from which we need to extract an element of C. This is easily done with the projection operator, so we have

$$\Phi(C, c_0, c_s, n) :\equiv \mathsf{pr}_2\Big(\mathsf{iter}_{\mathbb{N}\times C}\big((0, c_0), \lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)), n\big)\Big)$$

which has the same type as $\mathsf{rec}_{\mathbb{N}}$. In Coq we first define the iterator and then our alternative recursor:

```
Fixpoint iter (C : Type) (c0 : C) (cs : C → C) (n : nat) : C :=
  match n with
    | O ⇒ c0
    | S n' ⇒ cs(iter C c0 cs n')
  end.
Definition Phi (C : Type) (c0 : C) (cs: nat → C → C) (n : nat) :=
  snd (iter (nat × C)
            (O, c0)
            (fun x ⇒ (S (fst x), cs (fst x) (snd x)))
            n).
```

Now to show that the defining equations hold propositionally for Φ. For clarity of notation, define

$$\Phi'(n) = \mathsf{iter}_{\mathbb{N}\times C}\big((0, c_0), \lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)), n\big)$$

```
Definition Phi' (n : nat) :=
  iter (nat × C) (O, c0) (fun x ⇒ (S (fst x), cs (fst x) (snd x))) n.
```

So the propositional equalities can be written

$$\mathsf{pr}_2\Phi'(0) =_C c_0$$
$$\prod_{n:\mathbb{N}} \mathsf{pr}_2\Phi'(\mathsf{succ}(n)) =_C c_s(n, \mathsf{pr}_2\Phi'(n)).$$

The first is straightforward:

$$\mathsf{pr}_2\Phi'(0) \equiv \mathsf{pr}_2\mathsf{iter}_{\mathbb{N}\times C}\big((0, c_0), \lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)), 0\big) \equiv \mathsf{pr}_2(0, c_0) \equiv c_0$$

so $\mathsf{refl}_{c_0} : \mathsf{pr}_2\Phi'(0) =_C c_0$. To establish the second, we use induction on a strengthened hypothesis involving Φ'. We will establish that for all $n : \mathbb{N}$,

$$P(n) :\equiv \Phi'(\mathsf{succ}(n)) =_C (\mathsf{succ}(n), c_s(n, \mathsf{pr}_2\Phi'(n)))$$

is inhabited. For the base case, we have

$$\begin{aligned}
\Phi'(\mathsf{succ}(0)) &\equiv \mathsf{iter}_{\mathbb{N}\times C}\big((0, c_0), \lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)), \mathsf{succ}(0)\big) \\
&\equiv \Big(\lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x))\Big)\mathsf{iter}_{\mathbb{N}\times C}\big((0, c_0), \lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)), 0\big) \\
&\equiv \Big(\lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x))\Big)(0, c_0) \\
&\equiv (\mathsf{succ}(0), c_s(0, c_0)) \\
&\equiv (\mathsf{succ}(0), c_s(0, \mathsf{pr}_2\Phi'(0)))
\end{aligned}$$

using the derivation of the first propositional equality. So $P(0)$ is inhabited, or $p_0 : P(0)$. For the induction hypothesis, suppose that $n : \mathbb{N}$ and that $p_n : P(n)$. A little massaging gives

$$\begin{aligned}\Phi'(\mathsf{succ}(\mathsf{succ}(n))) &\equiv \mathsf{iter}_{\mathbb{N}\times C}\big((0, c_0), \lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x)), \mathsf{succ}(\mathsf{succ}(n))\big)\\ &\equiv \Big(\lambda x.\,(\mathsf{succ}(\mathsf{pr}_1 x), c_s(\mathsf{pr}_1 x, \mathsf{pr}_2 x))\Big)\Phi'(\mathsf{succ}(n))\\ &\equiv (\mathsf{succ}(\mathsf{pr}_1\Phi'(\mathsf{succ}(n))), c_s(\mathsf{pr}_1\Phi'(\mathsf{succ}(n)), \mathsf{pr}_2\Phi'(\mathsf{succ}(n))))\end{aligned}$$

We now apply based path induction using p_n. Consider the family

$$D : \prod_{z:\mathbb{N}\times C} (\Phi'(\mathsf{succ}(n)) = x) \to \mathcal{U}$$

given by

$$D(z) :\equiv \Big(\mathsf{succ}(\mathsf{pr}_1\Phi'(\mathsf{succ}(n))), c_s(\mathsf{pr}_1\Phi'(\mathsf{succ}(n)), \mathsf{pr}_2\Phi'(\mathsf{succ}(n)))\Big) = (\mathsf{succ}(\mathsf{pr}_1 z), c_s(\mathsf{pr}_1 z, \mathsf{pr}_2\Phi'(\mathsf{succ}(n))))$$

Clearly, we have

$$\mathsf{refl}_{\Phi'(\mathsf{succ}(\mathsf{succ}(n)))} : D(\Phi'(\mathsf{succ}(n)), \mathsf{refl}_{\Phi'(\mathsf{succ}(n))})$$

so by based path induction, there is an element

$$\begin{aligned}f((\mathsf{succ}(n), c_s(n, \mathsf{pr}_2\Phi'(n))), p_n) : \Big(&\mathsf{succ}(\mathsf{pr}_1\Phi'(\mathsf{succ}(n))), c_s(\mathsf{pr}_1\Phi'(\mathsf{succ}(n)), \mathsf{pr}_2\Phi'(\mathsf{succ}(n)))\Big)\\ &= (\mathsf{succ}(\mathsf{pr}_1(\mathsf{succ}(n), c_s(n, \mathsf{pr}_2\Phi'(n)))),\\ &\qquad c_s(\mathsf{pr}_1(\mathsf{succ}(n), c_s(n, \mathsf{pr}_2\Phi'(n))), \mathsf{pr}_2\Phi'(\mathsf{succ}(n))))\end{aligned}$$

Let $p_{n+1} :\equiv f((\mathsf{succ}(n), c_s(n, \mathsf{pr}_2\Phi'(n))))$. Our first bit of massaging allows us to replace the left hand side of this by $\Phi'(\mathsf{succ}(\mathsf{succ}(n))$. As for the right, applying the projections gives

$$p_{n+1} : \Phi'(\mathsf{succ}(\mathsf{succ}(n))) = (\mathsf{succ}(\mathsf{succ}(n)), c_s(\mathsf{succ}(n), \mathsf{pr}_2\Phi'(\mathsf{succ}(n)))) \equiv P(\mathsf{succ}(n))$$

Plugging all this into our induction principle for $\mathbb{N}$, we can discharge the assumption that $p_n : P(n)$ to obtain

$$q :\equiv \mathsf{ind}_{\mathbb{N}}(P, p_0, \lambda n.\,\lambda p_n.\,p_{n+1}, n) : P(n)$$

The propositional equality we're after is a consequence of this, which we again obtain by based path induction. Consider the family

$$E : \prod_{z:\mathbb{N}\times C} (\Phi'(n) = z) \to \mathcal{U}$$

given by

$$E(z, p) :\equiv \mathsf{pr}_2\Phi'(\mathsf{succ}(n)) = \mathsf{pr}_2 z$$

Again, it's clear that

$$\mathsf{refl}_{\mathsf{pr}_2\Phi'(\mathsf{succ}(n))} : E(\Phi'(\mathsf{succ}(n)), \mathsf{refl}_{\Phi'(\mathsf{succ}(n))})$$

So based path induction gives us a function

$$g((\mathsf{succ}(n), c_s(n, \mathsf{pr}_2\Phi'(n))), q) : \mathsf{pr}_2\Phi'(\mathsf{succ}(n)) = \mathsf{pr}_2(\mathsf{succ}(n), c_s(n, \mathsf{pr}_2\Phi'(n)))$$

and by applying the projection function on the right and discharging the assumption of n, we have shown that

$$\prod_{n:\mathbb{N}} \mathsf{pr}_2\Phi'(\mathsf{succ}(n)) = c_s(n, \mathsf{pr}_2\Phi'(n))$$

is inhabited. Next chapter we'll prove that functions are functors, and we won't have to do this based path induction every single time. It'll be great. Repeating it all in Coq, we have

```
  Theorem Phi'_correct1 : snd (Phi' 0) = c0.
  Proof.
    reflexivity.
  Defined.

  Theorem Phi'_correct2 : ∀ n, Phi'(S n) = (S n , cs n (snd (Phi' n))).
  Proof.
    intros. induction n. reflexivity.
    transitivity ((S (fst (Phi' (S n))), cs (fst (Phi' (S n))) (snd (Phi' (S n))))).
    reflexivity.
    rewrite IHn. reflexivity.
  Defined.

End Exercise4.
```

Exercise 1.5 (p. 56) Show that if we define $A + B :\equiv \sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$, then we can give a definition of ind_{A+B} for which the definitional equalities stated in §1.7 hold.

Solution Define $A + B$ as stated. We need to define a function of type

$$\mathsf{ind}'_{A+B} : \prod_{C:(A+B)\to\mathcal{U}} \left(\prod_{(a:A)} C(\mathsf{inl}(a))\right) \to \left(\prod_{(b:B)} C(\mathsf{inr}(b))\right) \to \prod_{(x:A+B)} C(x)$$

which means that we also need to define $\mathsf{inl}' : A \to A + B$ and $\mathsf{inr}' B \to A + B$; these are

$$\mathsf{inl}'(a) :\equiv (0_{\mathbf{2}}, a) \qquad\qquad \mathsf{inr}'(b) :\equiv (1_{\mathbf{2}}, b)$$

In Coq, we can use *sigT* to define coprd as a Σ-type:

```
Section Exercise5.
  Context {A B : Type}.
  Definition coprd := {x : Bool & if x then B else A}.
  Definition myinl (a : A) := existT (fun x:Bool ⇒ if x then B else A) false a.
  Definition myinr (b : B) := existT (fun x:Bool ⇒ if x then B else A) true b.
```

Suppose that $C : A + B \to \mathcal{U}$, $g_0 : \prod_{(a:A)} C(\mathsf{inl}'(a))$, $g_1 : \prod_{(b:B)} C(\mathsf{inr}'(b))$, and $x : A + B$; we're looking to define

$$\mathsf{ind}'_{A+B}(C, g_0, g_1, x)$$

We will use $\mathsf{ind}_{\sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)}$, and for notational convenience will write $\Phi :\equiv \sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$. ind_Φ has signature

$$\mathsf{ind}_\Phi : \prod_{C:\Phi\to\mathcal{U}} \left(\prod_{(x:\mathbf{2})}\prod_{(y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x))} C((x,y))\right) \to \prod_{(p:\Phi)} C(p)$$

So

$$\mathsf{ind}_\Phi(C) : \left(\prod_{(x:\mathbf{2})}\prod_{(y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x))} C((x,y))\right) \to \prod_{(p:\Phi)} C(p)$$

To obtain something of type $\prod_{(x:\mathbf{2})} \prod_{(y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x))} C((x,y))$ we'll have to use $\mathsf{ind}_{\mathbf{2}}$. In particular, for $B(x) :\equiv \prod_{(y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x))} C((x,y))$ we have

$$\mathsf{ind}_{\mathbf{2}}(B) : B(0_{\mathbf{2}}) \to B(1_{\mathbf{2}}) \to \prod_{x:\mathbf{2}} B(x)$$

along with

$$g_0 : \prod_{a:A} C(\mathsf{inl}'(a)) \equiv \prod_{a:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,0_{\mathbf{2}})} C((0_{\mathbf{2}},a)) \equiv B(0_{\mathbf{2}})$$

and similarly for g_1. So

$$\mathsf{ind}_{\mathbf{2}}(B, g_0, g_1) : \prod_{(x:\mathbf{2})} \prod_{(y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x))} C((x,y))$$

which is just what we needed for ind_{Φ}. So we define

$$\mathsf{ind}'_{A+B}(C, g_0, g_1, x) :\equiv \mathsf{ind}_{\sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} \left(C, \mathsf{ind}_{\mathbf{2}} \left(\prod_{y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} C((x,y)), g_0, g_1 \right), x \right)$$

and, in Coq, we use sigT_rect, which is the built-in $\mathsf{ind}_{\sum_{(x:A)} B(x)}$:

```
Definition indcoprd (C : coprd → Type) (g0 : ∀ a : A, C (myinl a)) (g1 : ∀ b : B, C (myinr b)) (x : coprd) :=
  sigT_rect C
            (Bool_rect (fun x:Bool ⇒ ∀ (y : if x then B else A), C (x; y))
                       g1
                       g0)
            x.
```

Now we must show that the definitional equalities

$$\mathsf{ind}'_{A+B}(C, g_0, g_1, \mathsf{inl}'(a)) \equiv g_0(a)$$
$$\mathsf{ind}'_{A+B}(C, g_0, g_1, \mathsf{inr}'(b)) \equiv g_1(b)$$

hold. For the first, we have

$$\begin{aligned}
\mathsf{ind}'_{A+B}(C, g_0, g_1, \mathsf{inl}'(a)) &\equiv \mathsf{ind}'_{A+B}(C, g_0, g_1, (0_{\mathbf{2}}, a)) \\
&\equiv \mathsf{ind}_{\sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} \left(C, \mathsf{ind}_{\mathbf{2}} \left(\prod_{y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} C((x,y)), g_0, g_1 \right), (0_{\mathbf{2}}, a) \right) \\
&\equiv \mathsf{ind}_{\mathbf{2}} \left(\prod_{y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} C((x,y)), g_0, g_1, 0_{\mathbf{2}} \right) (a) \\
&\equiv g_0(a)
\end{aligned}$$

and for the second,

$$\begin{aligned}
\mathsf{ind}'_{A+B}(C, g_0, g_1, \mathsf{inr}'(b)) &\equiv \mathsf{ind}'_{A+B}(C, g_0, g_1, (1_{\mathbf{2}}, b)) \\
&\equiv \mathsf{ind}_{\sum_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} \left(C, \mathsf{ind}_{\mathbf{2}} \left(\prod_{y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} C((x,y)), g_0, g_1 \right), (1_{\mathbf{2}}, b) \right) \\
&\equiv \mathsf{ind}_{\mathbf{2}} \left(\prod_{y:\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} C((x,y)), g_0, g_1, 1_{\mathbf{2}} \right) (b) \\
&\equiv g_1(b)
\end{aligned}$$

```
Theorem indcoprd_correct1 :
  ∀ C g0 g1 a, indcoprd C g0 g1 (myinl a) = g0 a.
Proof.
  reflexivity.
Qed.

Theorem indcoprd_correct2 :
  ∀ C g0 g1 a, indcoprd C g0 g1 (myinr a) = g1 a.
Proof.
  reflexivity.
Qed.

End Exercise5.
```

Exercise 1.6 (p. 56) Show that if we define $A \times B :\equiv \prod_{(x:\mathbf{2})} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$, then we can give a definition of $\mathsf{ind}_{A\times B}$ for which the definitional equalities stated in §1.5 hold propositionally (i.e. using equality types).

```
Section Exercise6.
  Context {A B : Type}.
```

Solution Define

$$A \times B :\equiv \prod_{x:\mathbf{2}} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)$$

Supposing that $a : A$ and $b : B$, we have an element $(a, b) : A \times B$ given by

$$(a, b) :\equiv \mathsf{ind}_{\mathbf{2}}(\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B), a, b)$$

Defining this type and constructor in Coq, we have

```
  Definition prd := ∀ x : Bool, if x then B else A.
  Definition mypair (a : A) (b : B) := Bool_rect (fun x : Bool ⇒ if x then B else A) b a.
```

An induction principle for $A \times B$ will, given a family $C : A \times B \to \mathcal{U}$ and a function

$$g : \prod_{(x:A)} \prod_{(y:B)} C((x, y)),$$

give a function $f : \prod_{(x:A\times B)} C(x)$ defined by

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

So suppose that we have such a C and g. Writing things out in terms of the definitions, we have

$$C : \left(\prod_{x:\mathbf{2}} \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, x)\right) \to \mathcal{U}$$
$$g : \prod_{(x:A)} \prod_{(y:B)} C(\mathsf{ind}_{\mathbf{2}}(\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B), x, y))$$

We can define projections by

$$\mathsf{pr}_1 p :\equiv p(0_{\mathbf{2}}) \qquad \mathsf{pr}_2 p :\equiv p(1_{\mathbf{2}})$$

Since p is an element of a dependent type, we have

$$p(0_{\mathbf{2}}) : \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, 0_{\mathbf{2}}) \equiv A$$
$$p(1_{\mathbf{2}}) : \mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B, 1_{\mathbf{2}}) \equiv B$$

```
Definition myfst (p : prd) := p false.
Definition mysnd (p : prd) := p true.
```

Then we have

$$g(\mathsf{pr}_1 p)(\mathsf{pr}_2 p) : C(\mathsf{ind}_{\mathbf{2}}(\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B), (\mathsf{pr}_1 p), (\mathsf{pr}_2 p))) \equiv C((p(0_{\mathbf{2}}), p(1_{\mathbf{2}})))$$

So we have defined a function

$$f' : \prod_{p:A\times B} C((p(0_{\mathbf{2}}), p(1_{\mathbf{2}})))$$

But we need one of the type

$$f : \prod_{p:A\times B} C(p)$$

To solve this problem, we need to appeal to function extensionality from §2.9. This implies that there is a function

$$\mathsf{funext} : \left(\prod_{x:\mathbf{2}} ((\mathsf{pr}_1 p, \mathsf{pr}_2 p)(x) =_{\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} p(x))\right) \to ((\mathsf{pr}_1 p, \mathsf{pr}_2 p) =_{A\times B} p)$$

We just need to show that the antecedent is inhabited, which we can do with $\mathsf{ind}_{\mathbf{2}}$. So consider the family

$$\begin{aligned} E &:\equiv \lambda(x:\mathbf{2}).\, ((p(0_{\mathbf{2}}), p(1_{\mathbf{2}}))(x) =_{\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} p(x))) \\ &\equiv \lambda(x:\mathbf{2}).\, (\mathsf{ind}_{\mathbf{2}}(\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B), p(0_{\mathbf{2}}), p(1_{\mathbf{2}}), x) =_{\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} p(x)) \end{aligned}$$

We have

$$\begin{aligned} E(0_{\mathbf{2}}) &\equiv (\mathsf{ind}_{\mathbf{2}}(\mathsf{rec}_{\mathbf{2}}(\mathcal{U}, A, B), p(0_{\mathbf{2}}), p(1_{\mathbf{2}}), 0_{\mathbf{2}}) =_{\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,0_{\mathbf{2}})} p(0_{\mathbf{2}})) \\ &\equiv (p(0_{\mathbf{2}}) =_{\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,0_{\mathbf{2}})} p(0_{\mathbf{2}})) \end{aligned}$$

Thus $\mathsf{refl}_{p(0_{\mathbf{2}})} : E(0_{\mathbf{2}})$. The same argument goes through to show that $\mathsf{refl}_{p(1_{\mathbf{2}})} : E(1_{\mathbf{2}})$. This means that

$$h :\equiv \mathsf{ind}_{\mathbf{2}}(E, \mathsf{refl}_{p(0_{\mathbf{2}})}, \mathsf{refl}_{p(1_{\mathbf{2}})}) : \prod_{x:\mathbf{2}} ((\mathsf{pr}_1 p, \mathsf{pr}_2 p)(x) =_{\mathsf{rec}_{\mathbf{2}}(\mathcal{U},A,B,x)} p(x))$$

and thus

$$\mathsf{funext}(h) : (p(0_{\mathbf{2}}), p(1_{\mathbf{2}})) =_{A\times B} p$$

This allows us to define the uniqueness principle for products:

$$\mathsf{uppt} :\equiv \lambda p.\, \mathsf{funext}(h) : \prod_{p:A\times B} (\mathsf{pr}_1 p, \mathsf{pr}_2 p) =_{A\times B} p$$

Now we can define $\mathsf{ind}_{A\times B}$ as

$$\mathsf{ind}_{A\times B}(C, g, p) :\equiv (\mathsf{uppt}\ p)_*(g(\mathsf{pr}_1 p)(\mathsf{pr}_2 p))$$

In Coq we can repeat this construction using Funext.

```
Definition myuppt `{Funext} (p : prd) : mypair (myfst p) (mysnd p) = p.
  apply path_forall.
  unfold pointwise_paths; apply Bool_rect; reflexivity.
Defined.
```

```
Definition indprd' (C : prd → Type) (g : ∀ (x:A) (y:B), C (mypair x y)) (z : prd) :=
  (myuppt z) # (g (myfst z) (mysnd z)).
```

Now, we must show that the definitional equality holds propositionally. That is, we must show that the type

$$\mathsf{ind}_{A\times B}(C, g, (a, b)) =_{C((a,b))} g(a)(b)$$

is inhabited. Unfolding the left gives

$$\begin{aligned}\mathsf{ind}_{A\times B}(C, g, (a, b)) &\equiv (\mathsf{uppt}\,(a, b))_*(g(\mathsf{pr}_1(a, b))(\mathsf{pr}_2(a, b))) \\ &\equiv \mathsf{ind}_{=_{C((a,b))}}(D, d, (a, b), (a, b), \mathsf{uppt}\,(a, b))(g(a)(b))\end{aligned}$$

where $D : \prod_{(x,y:A\times B)}(x = y) \to \mathcal{U}$ is given by $D(x, y, p) :\equiv C(x) \to C(y)$ and

$$d :\equiv \lambda x.\, \mathsf{id}_{C(x)} : \prod_{x:A\times B} D(x, x, \mathsf{refl}_x)$$

Now,

$$\mathsf{uppt}\,(a, b) \equiv \mathsf{funext}(h) : (a, b) =_{A\times B} (a, b)$$

and, in particular, we have $h : x \mapsto \mathsf{refl}_{(a,b)(x)}$, so $\mathsf{funext}(h) = \mathsf{refl}_{(a,b)}$. Plugging this into $\mathsf{ind}_{=_{C((a,b))}}$ and applying its defining equality gives

$$\begin{aligned}\mathsf{ind}_{A\times B}(C, g, (a, b)) &= \mathsf{ind}_{=_{C((a,b))}}(D, d, (a, b), (a, b), \mathsf{refl}_{(a,b)})(g(a)(b)) \\ &= d((a, b))(g(a)(b)) \\ &= \mathsf{id}_{C((a,b))}(g(a)(b)) \\ &= g(a)(b)\end{aligned}$$

Verifying that the definitional equality holds propositionally. The reason we can only get propositional equality, not judgemental equality, is that $\mathsf{funext}(h) = \mathsf{refl}_{(a,b)}$ is just a propositional equality. Understanding this better requires stuff from next chapter.

```
Lemma lemma1_6 : ∀ a b, myuppt (mypair a b) = 1.
Proof.
  intros. unfold myuppt.
  transitivity (path_forall (mypair (myfst (mypair a b)) (mysnd (mypair a b)))
                            _
                            (fun _ ⇒ 1)).
  f_ap. by_extensionality x. destruct x; reflexivity.
  unfold mypair, myfst, mysnd. simpl.
  apply path_forall_1 with (f:=Bool_rect (fun x : Bool ⇒ if x then B else A) b a).
Qed.

Goal ∀ C g a b, indprd' C g (mypair a b) = g a b.
Proof.
  intros. unfold indprd'. unfold transport.
  rewrite lemma1_6.
  unfold mypair, myfst, mysnd. reflexivity.
Qed.

End Exercise6.
```

Exercise 1.7 (p. 56) Give an alternative derivation of $\mathsf{ind}'_{=_A}$ from $\mathsf{ind}_{=_A}$ which avoids the use of universes.

Solution To avoid universes, we follow the plan from p. 53 of the text: show that $\mathsf{ind}_{=_A}$ entails Lemmas 2.3.1 and 3.11.8, and that these two principles imply $\mathsf{ind}'_{=_A}$ directly.

First we have Lemma 2.3.1, which states that for any type family P over A and $p : x =_A y$, there is a function $p_* : P(x) \to P(y)$. The proof for this can be taken directly from the text. Consider the type family

$$D : \prod_{x,y:A} (x = y) \to \mathcal{U}, \qquad D(x, y, p) :\equiv P(x) \to P(y)$$

which exists, since $P(x) : \mathcal{U}$ for all $x : A$ and these can be used to form function types. We also have

$$d :\equiv \lambda x.\, \mathsf{id}_{P(x)} : \prod_{x:A} D(x, x, \mathsf{refl}_x) \equiv \prod_{x:A} P(x) \to P(x)$$

We now apply $\mathsf{ind}_{=_A}$ to obtain

$$p_* :\equiv \mathsf{ind}_{=_A}(D, d, x, y, p) : P(x) \to P(y)$$

establishing the Lemma.

Next we have Lemma 3.11.8, which states that for any A and any $a : A$, the type $\sum_{(x:A)}(a = x)$ is contractible; that is, there is some $w : \sum_{(x:A)}(a = x)$ such that $w = w'$ for all $w' : \sum_{(x:A)}(a = x)$. Consider the point $(a, \mathsf{refl}_a) : \sum_{(a:A)}(a = x)$ and the family $C : \prod_{(x,y:A)}(x = y) \to \mathcal{U}$ given by

$$C(x, y, p) :\equiv ((x, \mathsf{refl}_x) =_{\sum_{(z:A)}(x=z)} (y, p))$$

Take also the function

$$\mathsf{refl}_{(x,\mathsf{refl}_x)} : \prod_{x:A} ((x, \mathsf{refl}_x) =_{\sum_{(x:A)}(x=z)} (x, \mathsf{refl}_x))$$

By path induction, then, we have a function

$$g : \prod_{(x,y:A)} \prod_{(p:x=_A y)} ((x, \mathsf{refl}_x) =_{\sum_{(z:A)}(x=z)} (y, p))$$

such that $g(x, x, \mathsf{refl}_x) :\equiv \mathsf{refl}_{(x,\mathsf{refl}_x)}$. This allows us to construct

$$\lambda p.\, g(a, \mathsf{pr}_1 p, \mathsf{pr}_2 p) : \prod_{p:\sum_{(x:A)}(a=x)} (a, \mathsf{refl}_a) =_{\sum_{(z:A)}(a=z)} (\mathsf{pr}_1 p, \mathsf{pr}_2 p)$$

And upst lets us transport this, using the first lemma, to the statement that $\sum_{(x:A)}(a = x)$ is contractible:

$$\mathsf{contr} :\equiv \lambda p.\, \Big((\mathsf{upst}\, p)_* g(a, \mathsf{pr}_1 p, \mathsf{pr}_2 p)\Big) : \prod_{p:\sum_{(x:A)}(a=x)} (a, \mathsf{refl}_a) =_{\sum_{(z:A)}(a=z)} p$$

With these two lemmas we can derive based path induction. Fix some $a : A$ and suppose we have a family

$$C : \prod_{x:A} (a = x) \to \mathcal{U}$$

and an element

$$c : C(a, \mathsf{refl}_a).$$

Suppose we have $x : A$ and $p : a = x$. Then we have $(x, p) : \sum_{(x:A)}(a = x)$, and because this type is contractible, an element $\mathsf{contr}_{(x,p)} : (a, \mathsf{refl}_a) = (x, p)$. So for any type family P over $\sum_{(x:A)}(a = x)$, we have the function $(\mathsf{contr}_{(x,p)})_* : P((a, \mathsf{refl}_a)) \to P((x, p))$. In particular, we have the type family

$$\tilde{C} :\equiv \lambda p.\, C(\mathsf{pr}_1 p, \mathsf{pr}_2 p)$$

so

$$(\mathsf{contr}_{(x,p)})_* : \tilde{C}((a, \mathsf{refl}_a)) \to \tilde{C}((x, p)) \equiv C(a, \mathsf{refl}_a) \to C(x, p).$$

thus

$$(\mathsf{contr}_{(x,p)})_*(c) : C(x, p)$$

or, abstracting out the x and p,

$$f :\equiv \lambda x.\, \lambda p.\, (\mathsf{contr}_{(x,p)})_*(c) : \prod_{(x:A)} \prod_{(p:x=y)} C(x, p).$$

We also have

$$\begin{aligned}
f(a, \mathsf{refl}_a) &\equiv (\mathsf{contr}_{(a,\mathsf{refl}_a)})_*(c) \\
&\equiv ((\mathsf{upst}\ (a, \mathsf{refl}_a))_* g(a, a, \mathsf{refl}_a))_*(c) \\
&\equiv ((\mathsf{upst}\ (a, \mathsf{refl}_a))_* \mathsf{refl}_{(a,\mathsf{refl}_a)})_*(c) \\
&\equiv (\mathsf{ind}_=(\lambda x.\, ((a, \mathsf{refl}_a) = x), \lambda x.\, \mathsf{id}_{(a,\mathsf{refl}_a)=x}, (a, \mathsf{refl}_a), (a, \mathsf{refl}_a), \mathsf{refl}_{(a,\mathsf{refl}_a)})\mathsf{refl}_{(a,\mathsf{refl}_a)})_*(c) \\
&\equiv (\mathsf{id}_{(a,\mathsf{refl}_a)=(a,\mathsf{refl}_a)} \mathsf{refl}_{(a,\mathsf{refl}_a)})_*(c) \\
&\equiv (\mathsf{refl}_{(a,\mathsf{refl}_a)})_*(c) \\
&\equiv \mathsf{ind}_=(\tilde{C}, \lambda x.\, \mathsf{id}_{\tilde{C}(x)}, (a, \mathsf{refl}_a), (a, \mathsf{refl}_a), \mathsf{refl}_{(a,\mathsf{refl}_a)})(c) \\
&\equiv \mathsf{id}_{\tilde{C}((a,\mathsf{refl}_a))}(c) \\
&\equiv \mathsf{id}_{C(a,\mathsf{refl}_a)}(c) \\
&\equiv c
\end{aligned}$$

So we have derived based path induction.

```
Definition ind {A} : ∀ (C : ∀ (x y : A), x = y → Type),
                         (∀ (x:A), C x x 1) →
                         ∀ (x y : A) (p : x = y), C x y p.
  path_induction. apply X.
Defined.

Definition Lemma231 {A} (P : A → Type) (x y : A) (p : x = y) : P(x) → P(y).
  intro. rewrite ← p. apply X.
Defined.

Definition Lemma3118 {A} : ∀ (a:A), Contr {x:A & a=x}.
  intro a. ∃ (a; 1).
  intro x. destruct x as [x p]. path_induction. reflexivity.
Defined.

Definition ind' {A} : ∀ (a : A) (C : ∀ (x:A), a = x → Type),
                          C a 1 → ∀ (x:A) (p:a=x), C x p.
  intros.
  assert (Contr {x:A & a=x}) as H. apply Lemma3118.
  change (C x p) with ((fun c ⇒ C c.1 c.2) (x; p)).
  apply (Lemma231 _ (a; 1) (x; p)).
  transitivity (center {x : A & a = x}). destruct H as [[a' p'] z]. simpl.
  rewrite ← p'. reflexivity.
  destruct H as [[a' p'] z]. simpl. rewrite ← p'. rewrite ← p. reflexivity.
  apply X.
Defined.
```

Exercise 1.8 (p. 56) Define multiplication and exponentiation using $\mathsf{rec}_{\mathbb{N}}$. Verify that $(\mathbb{N}, +, 0, \times, 1)$ is a semiring using only $\mathsf{ind}_{\mathbb{N}}$.

```
Local Open Scope nat_scope.
```

Solution For multiplication, we need to construct a function $\mathsf{mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Defined with pattern-matching, we would have

$$\begin{aligned} \mathsf{mult}(0, m) &:\equiv 0 \\ \mathsf{mult}(\mathsf{succ}(n), m) &:\equiv m + \mathsf{mult}(n, m) \end{aligned}$$

so in terms of $\mathsf{rec}_{\mathbb{N}}$ we have

$$\mathsf{mult} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \lambda n.\, 0, \lambda n.\, \lambda g.\, \lambda m.\, \mathsf{add}(m, g(m)))$$

For exponentiation, we have the function $\mathsf{exp} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the intention that $\mathsf{exp}(e, b) = b^e$. In terms of pattern matching,

$$\begin{aligned} \mathsf{exp}(0, b) &:\equiv 1 \\ \mathsf{exp}(\mathsf{succ}(e), b) &:\equiv \mathsf{mult}(b, \mathsf{exp}(e, b)) \end{aligned}$$

or, in terms of $\mathsf{rec}_{\mathbb{N}}$,

$$\mathsf{exp} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \lambda n.\, 1, \lambda n.\, \lambda g.\, \lambda m.\, \mathsf{mult}(m, g(m)))$$

In Coq, we can define these by

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | 0 ⇒ m
    | S p ⇒ S (p + m)
  end

    where "n + m" := (plus n m) : nat_scope.
Fixpoint mult (n m : nat) : nat :=
  match n with
    | 0 ⇒ 0
    | S p ⇒ m + (p × m)
  end

    where "n * m" := (mult n m) : nat_scope.
Fixpoint myexp (e b : nat) :=
  match e with
    | 0 ⇒ S 0
    | S e' ⇒ b × (myexp e' b)
  end.
```

To verify that $(\mathbb{N}, +, 0, \times, 1)$ is a semiring, we need stuff from Chapter 2. In particular, we need the following properties of the identity. First, for all types A and $x, y : A$, we have the inversion mapping, with type

$$p \mapsto p^{-1} : (x = y) \to (y = x)$$

and such that $\mathsf{refl}_x^{-1} \equiv \mathsf{refl}_x$ for each $x : A$. Second, for $x, y, z : A$ we have concatenation:

$$p \mapsto q \mapsto p \cdot q : (x = y) \to (y = z) \to (x = z)$$

such that $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$ for any $x : A$. To show that $(\mathbb{N}, +, 0, \times, 1)$ is a semiring, we need to verify that for all $n, m, k : \mathbb{N}$,

(i) $\prod_{(n:\mathbb{N})} 0 + n = n = n + 0$
(ii) $\prod_{(n:\mathbb{N})} 0 \times n = 0 = n \times 0$.
(iii) $\prod_{(n:\mathbb{N})} 1 \times n = n = n \times 1$
(iv) $\prod_{(n,m:\mathbb{N})} n + m = m + n$
(v) $\prod_{(n,m,k:\mathbb{N})} (n + m) + k = n + (m + k)$
(vi) $\prod_{(n,m,k:\mathbb{N})} (n \times m) \times k = n \times (m \times k)$
(vii) $\prod_{(n,m,k:\mathbb{N})} n \times (m + k) = (n \times m) + (n \times k)$
(viii) $\prod_{(n,m,k:\mathbb{N})} (n + m) \times k = (n \times k) + (m \times k)$

For (i)–(iii), we show each equality separately and then use concatenation to show the implicit third equality. We dream of next chapter, where we obtain the function ap.

(i) For all $n : \mathbb{N}$, we have

$$0 + n \equiv \mathsf{add}(0, n) \equiv n$$

so $\mathsf{refl} : \prod_{(n:\mathbb{N})} 0 + n = n$. For the other equality we'll need induction on n. For the base case, we have

$$0 + 0 \equiv \mathsf{add}(0, 0) \equiv 0.$$

so $\mathsf{refl}_0 : 0 = 0 + 0$. Fix n and suppose for the induction step that $p_n : n = n + 0$. Then we have

$$\mathsf{succ}(n) + 0 \equiv \mathsf{add}(\mathsf{succ}(n), 0) \equiv \mathsf{succ}(\mathsf{add}(n, 0))$$

so we turn again to based path induction, with the family

$$C : \prod_{m:\mathbb{N}} (n = m) \to \mathcal{U} \qquad\qquad C(m, p) :\equiv (\mathsf{succ}(n) = \mathsf{succ}(m))$$

and the element $\mathsf{refl}_{\mathsf{succ}(n)} : C(n, \mathsf{refl}_n)$. So we have

$$\mathsf{ind}'_{=}(n, C, \mathsf{refl}_{\mathsf{succ}(n)}, \mathsf{refl}_n, \mathsf{add}(n, 0), p_n) : \mathsf{succ}(n) = \mathsf{succ}(\mathsf{add}(n, 0))$$

and discharging our induction step gives

$$q :\equiv \mathsf{ind}_{\mathbb{N}}(\lambda n.\, (n = n + 0), \mathsf{refl}_0, \lambda n.\, \mathsf{ind}'_{=}(n, C, \mathsf{refl}_{\mathsf{succ}(n)}, \mathsf{refl}_n, \mathsf{add}(n, 0))) : \prod_{n:\mathbb{N}} (n = n + 0)$$

For the final equality, we use concatenation. From $\mathsf{refl}_n : 0 + n = n$ and $q_n : n = n + 0$, we have $\mathsf{refl}_n \cdot q_n : 0 + n = n + 0$.

(ii) For all $n : \mathbb{N}$,

$$0 \times n \equiv \mathsf{mult}(0, n) \equiv 0$$

so $\lambda n.\, \mathsf{refl}_0 : \prod_{(n:\mathbb{N})} 0 \times n = 0$. For the other direction, induction on n. The base case is

$$0 \times 0 = \mathsf{mult}(0, 0) = 0$$

so $\mathsf{refl}_0 : 0 = 0 \times 0$. Fixing n and supposing for the induction step that $p_n : 0 = n \times 0$, we have

$$\mathsf{mult}(\mathsf{succ}(n), 0) \equiv 0 + \mathsf{mult}(n, 0) \equiv \mathsf{add}(0, \mathsf{mult}(n, 0)) \equiv \mathsf{mult}(n, 0)$$

so $p_n : 0 = \mathsf{succ}(n) \times 0$. Thus

$$q :\equiv \mathsf{ind}_{\mathbb{N}}(\lambda n.\, (0 = n \times 0), \mathsf{refl}_0, \lambda n.\, \mathsf{id}_{n=n\times 0}) : \prod_{n:\mathbb{N}} (n = n \times 0).$$

And again, $\mathsf{refl}_0 \cdot q_n : 0 \times n = n \times 0$ gives us the last equality.

(iii) For all $n : \mathbb{N}$,

$$1 \times n \equiv \mathsf{succ}(0) \times n \equiv n + (0 \times n) \equiv n + 0$$

so, recalling q_n from (i), we have $\mathsf{refl}_{1\times n} \cdot q_n^{-1} : 1 \times n = n$. For the other direction, we proceed by induction on n. For the base case we have

$$0 \times 1 \equiv \mathsf{mult}(0, 1) \equiv 0$$

so $\mathsf{refl}_0 : 0 = 0 \times 1$. Fixing n and supposing for induction that $p_n : n = n \times 1$, we have

$$\mathsf{mult}(\mathsf{succ}(n), 1) \equiv 1 + \mathsf{mult}(n, 1) \equiv \mathsf{succ}(0) + \mathsf{mult}(n, 1) \equiv \mathsf{succ}(n \times 1)$$

So we turn to based path induction again. Let $C(m) = \mathsf{succ}(n) = \mathsf{succ}(m)$; then

$$\mathsf{ind}'_{=}(n, C, \mathsf{refl}_{\mathsf{succ}(n)}, n \times 1, p_n) : \mathsf{succ}(n) = \mathsf{succ}(n \times 1)$$

and

$$r :\equiv \mathsf{ind}_{\mathbb{N}}(\lambda n.\, (n = n \times 1), \mathsf{refl}_0, \lambda n.\, \mathsf{ind}'_{=}(n, C, \mathsf{refl}_{\mathsf{succ}(n)}, n \times 1)) : \prod_{n:\mathbb{N}} (n = n \times 1)$$

For the third equality, finally, $\mathsf{refl}_{1\times n} \cdot q_n^{-1} \cdot r_n : 1 \times n = n \times 1$.

(iv) We first prove an auxiliary lemma by induction: $\prod_{(n,m:\mathbb{N})} \mathsf{succ}(n + m) = n + \mathsf{succ}(m)$. For the base case, we have $\mathsf{succ}(0 + m) \equiv \mathsf{succ}(m) \equiv 0 + \mathsf{succ}(m)$, so $\mathsf{refl}_{\mathsf{succ}(m)} : \mathsf{succ}(0 + m) = 0 + \mathsf{succ}(m)$. Fix $n : \mathbb{N}$, and suppose for induction that $p_n : \mathsf{succ}(n + m) = n + \mathsf{succ}(m)$. Then

$$\mathsf{succ}(\mathsf{succ}(n) + m) \equiv \mathsf{succ}(\mathsf{succ}(n + m))$$

and based path induction on $C(m) :\equiv \mathsf{succ}(\mathsf{succ}(n + m)) = \mathsf{succ}(m)$ gives

$$\mathsf{ind}'_{=}(\mathsf{succ}(n + m), C, \mathsf{refl}_{\mathsf{succ}(\mathsf{succ}(n+m))}, n + \mathsf{succ}(m), p_n) : \mathsf{succ}(\mathsf{succ}(n + m)) = \mathsf{succ}(n + \mathsf{succ}(m))$$

so letting $D(n) :\equiv \prod_{(m:\mathbb{N})} (\mathsf{succ}(n + m) = n + \mathsf{succ}(m))$,

$$r :\equiv \mathsf{ind}_{\mathbb{N}}(D, \mathsf{refl}_{\mathsf{succ}(m)}, \lambda n.\, \mathsf{ind}'_{=}(\mathsf{succ}(n + m), C, \mathsf{refl}_{\mathsf{succ}(\mathsf{succ}(n+m))}, n + \mathsf{succ}(m))) : \prod_{n:\mathbb{N}} D(n)$$

We now proceed by induction on n to show (iv). For the base case, recalling q_n from (i), we have $\mathsf{refl}_m \cdot q_m : 0 + m = m + 0$. Fixing n and supposing for induction that $p_n : n + m = m + n$, we have

$$\mathsf{succ}(n) + m \equiv \mathsf{succ}(n + m)$$

We then apply based path induction on $E(k) :\equiv \mathsf{succ}(n + m) = \mathsf{succ}(k)$ to obtain

$$\begin{aligned} \mathsf{ind}'_{=}(n + m, E, \mathsf{refl}_{\mathsf{succ}(n+m)}, m + n, p_n) &: \mathsf{succ}(n) + m = \mathsf{succ}(m + n) \\ \mathsf{ind}'_{=}(n + m, E, \mathsf{refl}_{\mathsf{succ}(n+m)}, m + n, p_n) \cdot r_{m,n} &: \mathsf{succ}(n) + m = m + \mathsf{succ}(n) \end{aligned}$$

and, finally, for the family $F(n) = n + m = m + n$,

$$\mathsf{ind}_{\mathbb{N}}(F, \mathsf{refl}_m \cdot q_m, \lambda n.\, \lambda p.\, (\mathsf{ind}'_{=}(n + m, E, \mathsf{refl}_{\mathsf{succ}(n+m)}, m + n, p) \cdot r_{m,n})) : \prod_{n:\mathbb{M}} n + m = m + m$$

Abstracting out the m gives us (iv).

(v) Fix m and k. We proceed by induction on n. For the base case,

$$(0+m)+k \equiv m+k \equiv 0+(m+k)$$

By the definition of add. Fix n, and suppose that $p_n : (n+m)+k = n+(m+k)$. We have

$$(\mathsf{succ}(n)+m)+k \equiv \mathsf{succ}(n+m)+k \equiv \mathsf{succ}((n+m)+k)$$

So based path induction on $C(\ell) = \mathsf{succ}((n+m)+k) = \mathsf{succ}(\ell)$ gives

$$\mathsf{ind}'_{=}((n+m)+k, C, \mathsf{refl}_{\mathsf{succ}((n+m)+k)}, n+(m+k), p_n) : \mathsf{succ}((n+m)+k) = \mathsf{succ}(n+(m+k))$$

which is equivalently the type $(\mathsf{succ}(n)+m)+k = \mathsf{succ}(n)+(m+k)$. So induction over $D(n) = (n+m)+k = n+(m+k)$ gives

$$\mathsf{ind}_{\mathbb{N}}(D, \mathsf{refl}_{(0+m)+k}, \lambda n.\, \lambda p.\, \mathsf{ind}'_{=}((n+m)+k, C, \mathsf{refl}_{\mathsf{succ}((n+m)+k)}, n+(m+k), p)) : \prod_{n:\mathbb{N}} D(n)$$

and abstracting out the m and k gives us (v).

(vi) Fix m and k. First an auxiliary lemma; we show that $(n+m)\times k = (n\times k)+(m\times k)$ by induction on n. For the base case,

$$(0+m)\times k \equiv m\times k \equiv 0+(m\times k) \equiv (0\times k)+(m\times k)$$

Now fix n and suppose that $p_n : (n+m)\times k = n\times k + m\times k$.

$$(\mathsf{succ}(n)+m)\times k \equiv \mathsf{succ}(n+m)\times k \equiv k+(n+m)\times k$$

and

$$\mathsf{succ}(n)\times k + m\times k \equiv (k+n\times k)+m\times k$$

Using based path induction over $C(\ell) :\equiv k+(n+m)\times k = k+\ell$, we get

$$\mathsf{ind}'_{=}((n+m)\times k, C, \mathsf{refl}_{k+(n+m)\times k}, n\times k+m\times k, p_n) : k+(n+m)\times k = k+(n\times k+m\times k)$$

We established in (v) that addition is associative, so we have some

$$r^{-1}_{k,n\times k,m\times k} : k+(n\times k+m\times k) = (k+n\times k)+m\times k$$

and concatenating this with the result of the based path induction gives something of type

$$k+(n+m)\times k = (k+n\times k)+m\times k$$

Our two strings of judgemental equalities mean that this is the same as the type

$$(\mathsf{succ}(n)+m)\times k = \mathsf{succ}(n)\times k+m\times k.$$

So we can now perform the induction over $D(\ell) = (n+m)\times k = n\times k+m\times k$ to obtain

$$\mathsf{ind}_{\mathbb{N}}(D, \mathsf{refl}_{(0+m)\times k}, \lambda n.\, \lambda p.\, (\mathsf{ind}'_{=}((n+m)\times k, C, \mathsf{refl}_{k+(n+m)\times k}, n\times k+m\times k, p_n) \cdot r^{-1}_{k,n\times k,m\times k}))$$

which is of type

$$\prod_{n:\mathbb{N}} (n+m)\times k = n\times k+m\times k$$

abstracting out the m and k give the final result (i.e., that multiplication on the right distributes over addition).

Now, for (vi). As always, it's induction on n. For the base case

$$(0 \times m) \times k \equiv 0 \times k \equiv 0 \equiv 0 \times (m \times k)$$

Now fix n and assume that $p_n : (n \times m) \times k = n \times (m \times k)$. We have

$$(\mathsf{succ}(n) \times m) \times k \equiv (m + n \times m) \times k$$

and

$$\mathsf{succ}(n) \times (m \times k) \equiv m \times k + n \times (m \times k)$$

From our lemma, then, there is a function

$$q : \prod_{n:\mathbb{N}} (\mathsf{succ}(n) \times m) \times k = m \times k + (n \times m) \times k$$

we use based path induction over $E(\ell) :\equiv m \times k + \ell$ to obtain

$$\mathsf{ind}'_{=}((n \times m) \times k, E, \mathsf{refl}_{m \times k + (n \times m) \times k}, n \times (m \times k), p_n) : m \times k + (n \times m) \times k = m \times k + n \times (m \times k)$$

which, concatenated with q_n and altered by the second judgemental equality, gives something of type

$$(\mathsf{succ}(n) \times m) \times k = \mathsf{succ}(n) \times (m \times k)$$

So our induction principle over $F(\ell) :\equiv (n \times m) \times k = n \times (m \times k)$ gives

$$\mathsf{ind}_{\mathbb{N}}(F, \mathsf{refl}_{(0 \times m) \times k}, \lambda n. \lambda p. (q_n \cdot \mathsf{ind}'_{=}((n \times m) \times k, E, \mathsf{refl}_{m \times k + (n \times m) \times k}, n \times (m \times k), p_n)))$$

of type

$$\prod_{n:\mathbb{N}} (n \times m) \times k = n \times (m \times k)$$

and abstracting out the m and k gives (vi).

(vii) Fix m and k. We proceed by induction on n. For the base case we have

$$0 \times (m + k) \equiv 0 \equiv 0 + 0 \equiv (0 \times m) + (0 \times k)$$

So fix $n : \mathbb{N}$ and suppose that $p_n : n \times (m + k) = (n \times m) + (n \times k)$. We have

$$\mathsf{succ}(n) \times (m + k) \equiv (m + k) + n \times (m + k)$$

and

$$(\mathsf{succ}(n) \times m) + (\mathsf{succ}(n) \times k) \equiv (m + n \times m) + (k + n \times k)$$

Now by (iv) and (v) we have the following two functions

$$q : \prod_{n,m:\mathbb{N}} n + m = m + n \qquad r : \prod_{n,m,k:\mathbb{N}} (n + m) + k = n + (m + k)$$

A long chain of based path inductions allows us to construct an object of type

$$(\mathsf{succ}(n) \times m) + (\mathsf{succ}(n) \times k) = (m + k) + (n \times m + n \times k)$$

In the interest of masochism, I'll do them explicitly. We start with

$$r_1 :\equiv r_{m, n \times m, k + n \times k} : (m + n \times m) + (k + n \times k) = m + (n \times m + (k + n \times k))$$

Based path induction over $C_1(\ell) :\equiv m + (n \times m + (k + n \times k)) = m + \ell$ and using

$$r_2 :\equiv r_{n\times m,k,n\times k} : n \times m + (k + n \times k) = (n \times m + k) + n \times k$$

gives

$$\langle r_2 \rangle :\equiv \mathsf{ind}'_{=}(n \times m + (k + n \times k), C_1, \mathsf{refl}_{m+(n\times m+(k+n\times k))}, (n \times m + k) + n \times k, r_2)$$

which results in

$$r_1 \cdot \langle r_2 \rangle : (m + n \times m) + (k + n \times k) = m + ((n \times m + k) + n \times k)$$

Next consider

$$q_1 :\equiv q_{n\times m,k} : n \times m + k = k + n \times m$$

which is passed through a based path induction on $C_2(\ell) :\equiv m + ((n \times m + k) + n \times k) = m + (\ell + n \times k)$ to get

$$\langle q_1 \rangle :\equiv \mathsf{ind}'_{=}(n \times m + k, C_2, \mathsf{refl}_{m+((n\times m+k)+n\times k)}, k + n \times m, q_1)$$

which adds to our chain, giving

$$r_1 \cdot \langle r_2 \rangle \cdot \langle q_1 \rangle : (m + n \times m) + (k + n \times k) = m + ((k + n \times m) + n \times k)$$

Now just two applications of associativity are left. We have

$$r_3 :\equiv r_{k,n\times m,n\times k} : (k + n \times m) + n \times k = k + (n \times m + n \times k)$$

so for $C_3(\ell) :\equiv m + ((k + n \times m) + n \times k) = m + \ell$, we have

$$\langle r_3 \rangle :\equiv \mathsf{ind}'_{=}((k + n \times m) + n \times k, C_3, \mathsf{refl}_{m+((k+n\times m)+n\times k)}, k + (n \times m + n \times k), r_3)$$

making our chain of type

$$r_1 \cdot \langle r_2 \rangle \cdot \langle q_1 \rangle \cdot \langle r_3 \rangle : (m + n \times m) + (k + n \times k) = m + (k + (n \times m + n \times k))$$

Finally, take

$$r_4 :\equiv r^{-1}_{m,k,n\times m+n\times k} : m + (k + (n \times m + n \times k)) = (m + k) + (n \times m + n \times k)$$

so after applying the last judgemental equality above, we have

$$f :\equiv r_1 \cdot \langle r_2 \rangle \cdot \langle q_1 \rangle \cdot \langle r_3 \rangle \cdot r_4 : (\mathsf{succ}(n) \times m) + (\mathsf{succ}(n) \times k) = (m + k) + (n \times m + n \times k)$$

Now, consider the family $D(\ell) :\equiv (m + k) + n \times (m + k) = (m + k) + \ell$. Based path induction once more gives us

$$\mathsf{ind}'_{=}(n \times (m + k), D, \mathsf{refl}_{(m+k)+n\times(m+k)}, n \times m + n \times k, p_n) \cdot f^{-1}$$

which, after application of our judgemental equalities, is of type

$$\mathsf{succ}(n) \times (m + k) = (\mathsf{succ}(n) \times m) + (\mathsf{succ}(n) \times k)$$

So we can at last apply induction over $\mathbb{N}$, using the family $E(n) : n \times (m + k) = (n \times m) + (n \times k)$, giving

$$\mathsf{ind}_{\mathbb{N}}(E, \mathsf{refl}_{0\times(m+k)}, \lambda n. \lambda p. (\mathsf{ind}'_{=}(n \times (m + k), D, \mathsf{refl}_{(m+k)+n\times(m+k)}, n \times m + n \times k, p) \cdot f^{-1}))$$

which is of type

$$\prod_{n:\mathbb{N}} n \times (m + k) = (n \times m) + (n \times k)$$

and m and k may be abstracted out to give (vii).

(viii) This was shown as a lemma in proving (vi).

In Coq we'll do things a touch out of order, so as to appeal to (viii) in the proof of (vi).

```
Theorem plus_0_r : ∀ (n : nat), n = plus n 0.
Proof.
  induction n; [| simpl; rewrite ← IHn]; reflexivity.
Qed.

Theorem ex1_8_i : ∀ (n : nat),
                     (0 + n = n) ∧ (n = n + 0) ∧ (0 + n = n + 0).
Proof.
  split; [| split; rewrite ← plus_0_r]; reflexivity.
Qed.

Theorem mult_0_r : ∀ (n : nat), 0 = n × 0.
Proof.
  induction n; [| simpl; rewrite ← IHn]; reflexivity.
Qed.

Theorem ex1_8_ii : ∀ (n : nat),
                     (0 × n = 0) ∧ (0 = n × 0) ∧ (0 × n = n × 0).
Proof.
  split; [| split; rewrite ← mult_0_r]; reflexivity.
Qed.

Theorem mult_1_r : ∀ (n : nat), n = n × 1.
Proof.
  induction n; [| simpl; rewrite ← IHn]; reflexivity.
Qed.

Theorem mult_1_l : ∀ (n : nat), 1 × n = n.
Proof.
  simpl; intro n; rewrite ← plus_0_r; reflexivity.
Qed.

Theorem ex1_8_iii : ∀ (n : nat),
                      (1 × n = n) ∧ (n = n × 1) ∧ (1 × n = n × 1).
Proof.
  split; [rewrite mult_1_l
         | split; rewrite ← mult_1_r;
           [| rewrite mult_1_l]];
  reflexivity.
Qed.

Theorem plus_n_Sm : ∀ (n m : nat), S (n + m) = n + (S m).
Proof.
  intros n m.
  induction n; [| simpl; rewrite IHn]; reflexivity.
Qed.

Theorem ex1_8_iv : ∀ (n m : nat), n + m = m + n.
Proof.
  intros n m.
  induction n; [ rewrite ← plus_0_r
               | simpl; rewrite ← plus_n_Sm; rewrite IHn];
  reflexivity.
Qed.
```

```
Definition plus_comm := ex1_8_iv.

Theorem ex1_8_v : ∀ (n m k : nat),
                    (n + m) + k = n + (m + k).
Proof.
  intros n m k.
  induction n; [| simpl; rewrite IHn]; reflexivity.
Qed.

Theorem ex1_8_viii : ∀ (n m k : nat),
                       (n + m) × k = (n × k) + (m × k).
Proof.
  intros n m k.
  induction n; [| simpl; rewrite IHn; rewrite ex1_8_v]; reflexivity.
Qed.

Theorem ex1_8_vi : ∀ (n m k : nat),
                     (n × m) × k = n × (m × k).
Proof.
  intros n m k.
  induction n; [| simpl; rewrite ← IHn; rewrite ← ex1_8_viii]; reflexivity.
Qed.

(*  Yikes.  This would be easier if replace were available *)
Theorem ex1_8_vii : ∀ (n m k : nat),
                      n × (m + k) = (n × m) + (n × k).
Proof.
  intros n m k.
  induction n; [reflexivity |].
  simpl. rewrite IHn. rewrite ← ex1_8_v. rewrite ← ex1_8_v.
  cut (m + n × m + k = m + k + n × m). intro H. rewrite H. reflexivity.
  rewrite ex1_8_v.
  cut (n × m + k = k + n × m). intro H. rewrite H. rewrite ← ex1_8_v. reflexivity.
  rewrite ex1_8_iv. reflexivity.
Qed.

Local Close Scope nat_scope.
```

Exercise 1.9 (p. 56) Define the type family $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$ mentioned at the end of §1.3, and the dependent function $\mathsf{fmax} : \prod_{(n:\mathbb{N})} \mathsf{Fin}(n+1)$ mentioned in §1.4.

```
Local Open Scope nat_scope.
```

Solution $\mathsf{Fin}(n)$ is a type with exactly n elements. Consider $\mathsf{Fin}(n)$ from the types-as-propositions point of view: $\mathsf{Fin}(n)$ is a predicate that applies to exactly n elements. Recalling that $\sum_{(m:\mathbb{N})}(m < n)$ may be regarded as "the type of all elements $m : \mathbb{N}$ such that $(m < n)$", we note that there are n such elements, and define

$$\mathsf{Fin}(n) :\equiv \sum_{m:\mathbb{N}} (m < n) \equiv \sum_{m:\mathbb{N}} \sum_{k:\mathbb{N}} (m + \mathsf{succ}(k) = n)$$

And in Coq,

```
Definition le (n m : nat) : Type := {k:nat & n + k = m}.
Notation "n <= m" := (le n m)%nat (at level 70) : nat_scope.

Definition lt (n m : nat) : Type := {k : nat & n + S k = m}.
Notation "n < m" := (lt n m)%nat (at level 70) : nat_scope.
```

```
Definition Fin (n:nat) : Type := {m : nat & m < n}.
```

To define fmax, note that one can think of an element of $\mathsf{Fin}(n)$ as a tuple $(m,(k,p))$, where $p : m + \mathsf{succ}(k) = n$. The maximum element of $\mathsf{Fin}(n+1)$ will have the greatest value in the first slot, so

$$\mathsf{fmax}(n) :\equiv n_{n+1} :\equiv (n, (0, \mathsf{refl}_{n+1})) : \sum_{(m:\mathbb{N})} \sum_{(k:\mathbb{N})} (m + \mathsf{succ}(k) = n+1) \equiv \mathsf{Fin}(n+1)$$

```
Definition fmax (n:nat) : Fin(n+1) := (n; (0; idpath)).
```

To verify that this definition is correct, we need to show that

$$\prod_{(n:\mathbb{N})} \prod_{(m_{n+1}:\mathsf{Fin}(n+1))} (\mathsf{pr}_1(m_{n+1}) \leq \mathsf{pr}_1(\mathsf{fmax}(n)))$$

is inhabited. Unfolding this a bit, we get

$$\prod_{(n:\mathbb{N})} \prod_{(m_{n+1}:\mathsf{Fin}(n+1))} (m \leq n) \equiv \prod_{(n:\mathbb{N})} \prod_{(m_{n+1}:\mathsf{Fin}(n+1))} \sum_{(k:\mathbb{N})} (m + k = n)$$

Fix some such n and m_{n+1}. By the induction principle for Σ-types, we can write $m_{n+1} = (m^1, (m^2, m^3))$, where $m^3 : m^1 + \mathsf{succ}(m^2) = n + 1$. Using the results of the previous exercise, we can obtain from m^3 a proof $p : m^1 + m^2 = n$. So (m^2, p) is a witness to our result.

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S n′ ⇒ n′
  end.

Theorem S_inj : ∀ (n m : nat), S n = S m → n = m.
Proof.
  intros.
  cut (pred (S n) = pred (S m)). intros.
  simpl in X. apply X.
  rewrite H. reflexivity.
Qed.

Theorem plus_1_r : ∀ n, S n = n + 1.
Proof.
  intros. rewrite plus_comm. reflexivity.
Qed.

Theorem fmax_correct : ∀ (n:nat) (m:Fin(n+1)),
                            m .1 ≤ (fmax n) .1.
Proof.
  unfold Fin, lt, le. intros. simpl.
  ∃ m.2.1.
  apply S_inj. rewrite plus_n_Sm.
  rewrite plus_1_r.
  apply m.2.2.
Qed.

Local Close Scope nat_scope.
```

Exercise 1.10 (p. 56) Show that the Ackermann function $\mathsf{ack} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, satisfying the following equations

$$\begin{aligned}
\mathsf{ack}(0, n) &\equiv \mathsf{succ}(n), \\
\mathsf{ack}(\mathsf{succ}(m), 0) &\equiv \mathsf{ack}(m, 1), \\
\mathsf{ack}(\mathsf{succ}(m), \mathsf{succ}(n)) &\equiv \mathsf{ack}(m, \mathsf{ack}(\mathsf{succ}(m), n)),
\end{aligned}$$

is definable using only $\mathsf{rec}_{\mathbb{N}}$.

Solution ack must be of the form

$$\mathsf{ack} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \Phi, \Psi)$$

with

$$\Phi : \mathbb{N} \to \mathbb{N} \qquad \Psi : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$

which we can determine by their intended behaviour. We have

$$\mathsf{ack}(0, n) \equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \Phi, \Psi, 0)(n) \equiv \Phi(n)$$

So we must have $\Phi :\equiv \mathsf{succ}$, which is of the correct type. The next equation gives us

$$\begin{aligned}
\mathsf{ack}(\mathsf{succ}(m), 0) &\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \Psi, \mathsf{succ}(m))(0) \\
&\equiv \Psi(m, \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \Psi, m))(0) \\
&\equiv \Psi(m, \mathsf{ack}(m, -), 0)
\end{aligned}$$

Suppose that Ψ is also defined in terms of $\mathsf{rec}_{\mathbb{N}}$. We know its signature, giving the first arg, and this second equation gives its behavior on 0, the second arg. So it must be of the form

$$\Psi = \lambda m. \lambda r. \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, r(1), \Theta(m, r)) \qquad \Theta : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

The final equation fixes Θ:

$$\begin{aligned}
&\mathsf{ack}(\mathsf{succ}(m), \mathsf{succ}(n)) \\
&\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \lambda m. \lambda r. \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, r(1), \Theta(m, r)), \mathsf{succ}(m))(\mathsf{succ}(n)) \\
&\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, \mathsf{ack}(m, 1), \Theta(m, \mathsf{ack}(m, -)), \mathsf{succ}(n)) \\
&\equiv \Theta(m, \mathsf{ack}(m, -), n, \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, \mathsf{ack}(m, 1), \Theta(m, \mathsf{ack}(m, -)), n)) \\
&\equiv \Theta(m, \mathsf{ack}(m, -), n, \Psi(m, \mathsf{ack}(m, -), n))
\end{aligned}$$

Looking at the second equation again suggests that the final argument to Θ is really $\mathsf{ack}(\mathsf{succ}(m), n)$. Supposing this is true,

$$\Theta :\equiv \lambda m. \lambda r. \lambda n. \lambda s. r(s)$$

should work. Putting it all together, we have

$$\mathsf{ack} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \lambda m. \lambda r. \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, r(1), \lambda n. \lambda s. r(s)))$$

In Coq, we define

```
Definition ack : nat → nat → nat :=
  nat_rect (fun _ ⇒ nat → nat)
           S
           (fun m r ⇒ nat_rect (fun _ ⇒ nat)
```

```
(r (S 0))
(fun n s ⇒ (r s))).
```

Now, to show that the three equations hold, we just calculate

$$\mathsf{ack}(0,n) \equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \lambda m.\, \lambda r.\, \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, r(1), \lambda n.\, \lambda s.\, r(s)), 0)(n) \equiv \mathsf{succ}(n)$$

for the first,

$$\begin{aligned} \mathsf{ack}(\mathsf{succ}(m),0) &\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \lambda m.\, \lambda r.\, \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, r(1), \lambda n.\, \lambda s.\, r(s)), \mathsf{succ}(m))(0) \\ &\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, \mathsf{ack}(m,1), \lambda n.\, \lambda s.\, \mathsf{ack}(m,s), 0) \\ &\equiv \mathsf{ack}(m,1) \end{aligned}$$

for the second, and finally

$$\begin{aligned} \mathsf{ack}(\mathsf{succ}(m),\mathsf{succ}(n)) &\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \lambda m.\, \lambda r.\, \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, r(1), \lambda n.\, \lambda s.\, r(s)), \mathsf{succ}(m))(\mathsf{succ}(n)) \\ &\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, \mathsf{ack}(m,1), \lambda n.\, \lambda s.\, \mathsf{ack}(m,s), \mathsf{succ}(n)) \\ &\equiv \mathsf{ack}(m, \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, \mathsf{ack}(m,1), \lambda n.\, \lambda s.\, \mathsf{ack}(m,s), n)) \end{aligned}$$

Focus on the second argument of the outer ack. We have

$$\begin{aligned} \mathsf{ack}(\mathsf{succ}(m),n) &\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \mathsf{succ}, \lambda m.\, \lambda r.\, \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, r(1), \lambda n.\, \lambda s.\, r(s)), \mathsf{succ}(m))(n) \\ &\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, \mathsf{ack}(m,1), \lambda n.\, \lambda s.\, \mathsf{ack}(m,s), n) \end{aligned}$$

and so we may substitute it back in to get

$$\mathsf{ack}(\mathsf{succ}(m),\mathsf{succ}(n)) \equiv \mathsf{ack}(m, \mathsf{ack}(\mathsf{succ}(m),n))$$

which is the third equality. In Coq,

```
Goal ∀ n, ack 0 n = S n. auto. Qed.
Goal ∀ m, ack (S m) 0 = ack m (S 0). auto. Qed.
Goal ∀ m n, ack (S m) (S n) = ack m (ack (S m) n). auto. Qed.
```

```
Close Scope nat_scope.
```

Exercise 1.11 (p. 56) Show that for any type A, we have $\neg\neg\neg A \to \neg A$.

Solution Suppose that $\neg\neg\neg A$ and A. Supposing further that $\neg A$, we get a contradiction with the second assumption, so $\neg\neg A$. But this contradicts the first assumption that $\neg\neg\neg A$, so $\neg A$. Discharging the first assumption gives $\neg\neg\neg A \to \neg A$.

In type-theoretic terms, the first assumption is $x : ((A \to \mathbf{0}) \to \mathbf{0}) \to \mathbf{0}$, and the second is $a : A$. If we further assume that $h : A \to \mathbf{0}$, then $h(a) : \mathbf{0}$, so discharging the h gives

$$\lambda(h : A \to \mathbf{0}).\, h(a) : (A \to \mathbf{0}) \to \mathbf{0}$$

But then we have

$$x(\lambda(h : A \to \mathbf{0}).\, h(a)) : \mathbf{0}$$

so discharging the a gives

$$\lambda(a : A).\, x(\lambda(h : A \to \mathbf{0}).\, h(a)) : A \to \mathbf{0}$$

And discharging the first assumption gives

$$\lambda(x : ((A \to \mathbf{0}) \to \mathbf{0}) \to \mathbf{0}).\, \lambda(a : A).\, x(\lambda(h : A \to \mathbf{0}).\, h(a)) : (((A \to \mathbf{0}) \to \mathbf{0}) \to \mathbf{0}) \to (A \to \mathbf{0})$$

```
Goal ∀ A, ¬ ¬ ¬ A → ¬A. auto. Qed.
```

We can get a proof out of Coq by printing this `Goal`. It returns

```
fun (A : Type) (X : ¬ ¬ ¬ A) (X0 : A) ⇒ X (fun X1 : A → Empty ⇒ X1 X0)
: ∀ A : Type, ¬ ¬ ¬ A → ¬ A
```

which is just the function obtained by hand.

Exercise 1.12 (p. 56) Using the propositions as types interpretation, derive the following tautologies.

(i) If A, then (if B then A).
(ii) If A, then not (not A).
(iii) If (not A or not B), then not (A and B).

```
Section Exercise12.
  Context {A B : Type}.
```

Solution (i) Suppose that A and B; then A. Discharging the assumptions, $A \to B \to A$. That is, we have

$$\lambda(a:A).\,\lambda(b:B).\,a : A \to B \to A$$

and in Coq,

```
  Goal A → B → A. trivial. Qed.
```

(ii) Suppose that A. Supposing further that $\neg A$ gives a contradiction, so $\neg\neg A$. That is,

$$\lambda(a:A).\,\lambda(f:A \to \mathbf{0}).\,f(a) : A \to (A \to \mathbf{0}) \to \mathbf{0}$$

```
  Goal A → ¬ ¬ A. auto. Qed.
```

(iii) Finally, suppose $\neg A \vee \neg B$. Supposing further that $A \wedge B$ means that A and that B. There are two cases. If $\neg A$, then we have a contradiction; but also if $\neg B$ we have a contradiction. Thus $\neg(A \wedge B)$.

Type-theoretically, we assume that $x : (A \to \mathbf{0}) + (B \to \mathbf{0})$ and $z : A \times B$. Conjunction elimination gives $\mathsf{pr}_1 z : A$ and $\mathsf{pr}_2 z : B$. We can now perform a case analysis. Suppose that $x_A : A \to \mathbf{0}$; then $x_A(\mathsf{pr}_1 z) : \mathbf{0}$, a contradicton; if instead $x_B : B \to \mathbf{0}$, then $x_B(\mathsf{pr}_2 z) : \mathbf{0}$. By the recursion principle for the coproduct, then,

$$f(z) :\equiv \mathsf{rec}_{(A\to\mathbf{0})+(B\to\mathbf{0})}(\mathbf{0}, \lambda x.\,x(\mathsf{pr}_1 z), \lambda x.\,x(\mathsf{pr}_2 z)) : (A \to \mathbf{0}) + (B \to \mathbf{0}) \to \mathbf{0}$$

Discharging the assumption that $A \times B$ is inhabited, we have

$$f : A \times B \to (A \to \mathbf{0}) + (B \to \mathbf{0}) \to \mathbf{0}$$

So

$$\mathsf{swap}(A \times B, (A \to \mathbf{0}) + (B \to \mathbf{0}), \mathbf{0}, f) : (A \to \mathbf{0}) + (B \to \mathbf{0}) \to A \times B \to \mathbf{0}$$

```
  Goal (¬ A + ¬ B) → ¬ (A × B).
  Proof.
    unfold not.
    intros H x.
    apply H.
    destruct x.
    constructor.
    exact a.
  Qed.
End Exercise12.
```

Exercise 1.13 (p. 57) Using propositions-as-types, derive the double negation of the principle of excluded middle, i.e. prove *not (not (P or not P))*.

```
Section Exercise13.
  Context {P : Type}.
```

Solution Suppose that $\neg(P \vee \neg P)$. Then, assuming P, we have $P \vee \neg P$ by disjunction introduction, a contradiction. Hence $\neg P$. But disjunction introduction on this again gives $P \vee \neg P$, a contradiction. So we must reject the remaining assumption, giving $\neg\neg(P \vee \neg P)$.

In type-theoretic terms, the initial assumption is that $g : P + (P \to \mathbf{0}) \to \mathbf{0}$. Assuming $p : P$, disjunction introduction results in $\mathsf{inl}(p) : P + (P \to \mathbf{0})$. But then $g(\mathsf{inl}(p)) : \mathbf{0}$, so we discharge the assumption of $p : P$ to get

$$\lambda(p : P).\, g(\mathsf{inl}(p)) : P \to \mathbf{0}$$

Applying disjunction introduction again leads to contradiction, as

$$g(\mathsf{inr}(\lambda(p : P).\, g(\mathsf{inl}(p)))) : \mathbf{0}$$

So we must reject the assumption of $\neg(P \vee \neg P)$, giving the result:

$$\lambda(g : P + (P \to \mathbf{0}) \to \mathbf{0}).\, g(\mathsf{inr}(\lambda(p : P).\, g(\mathsf{inl}(p)))) : (P + (P \to \mathbf{0}) \to \mathbf{0}) \to \mathbf{0}$$

Finally, in Coq,

```
  Goal ¬ ¬ (P + ¬P).
  Proof.
    unfold not.
    intro H.
    apply H.
    right.
    intro p.
    apply H.
    left.
    apply p.
  Qed.
End Exercise13.
```

Exercise 1.14 (p. 57) Why do the induction principles for identity types not allow us to construct a function $f : \prod_{(x:A)} \prod_{(p:x=x)} (p = \mathsf{refl}_x)$ with the defining equation

$$f(x, \mathsf{refl}_x) :\equiv \mathsf{refl}_{\mathsf{refl}_x} \qquad ?$$

Solution To attempt to define this function by path induction, we'd need a family

$$C :\equiv \lambda x.\, \lambda y.\, \lambda p.\, (p = \mathsf{refl}_{\mathsf{refl}_x})$$

and a function

$$c : \prod_{x:A} (p = \mathsf{refl}_{\mathsf{refl}_x})$$

But there is not always such a function c, since it is not always the case that there is only one path in $x = x$. Because of the possiblity of nontrivial homotopies, one might fail to have $(p = p) = (p = \mathsf{refl}_x)$.

Attempting this proof in Coq would go as

```
Definition f : ∀ (A : Type) (x : A) (p : x = x), p = 1.
intros. path_induction.
exact 1.
```

which returns the error message

The term "1" *has type* "p = p" *while it is expected to have type* "p = 1".

Exercise 1.15 (p. 57) Show that indiscernability of identicals follows from path induction.

Solution Consider some family $C : A \to \mathcal{U}$, and define

$$D : \prod_{x,y:A} (x =_A y) \to \mathcal{U}, \qquad D(x, y, p) :\equiv C(x) \to C(y)$$

Note that we have the function

$$\lambda x.\, \mathsf{id}_{C(x)} : \prod_{x:A} C(x) \to C(x) \equiv \prod_{x:A} D(x, x, \mathsf{refl}_x)$$

So by path induction there is a function

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} D(x, y, p) \equiv \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x) \to C(y)$$

such that

$$f(x, x, \mathsf{refl}_x) :\equiv \mathsf{id}_{C(x)}$$

But this is just the statement of the indiscernability of identicals: for every such family C, there is such an f.

2 Homotopy type theory

Exercise 2.1 (p. 103) Show that the three obvious proofs of Lemma 2.1.2 are pairwise equal.

Solution Lemma 2.1.2 states that for every type A and every $x, y, z : A$, there is a function

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

written $p \mapsto q \mapsto p \cdot q$ such that $\mathsf{refl}_x \cdot \mathsf{refl}_x = \mathsf{refl}_x$ for all $x : A$. Each proof is an object $\cdot_i$ of type

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

So we need to show that $\cdot_1 = \cdot_2 = \cdot_3$.

The first proof is induction over p. Consider the family

$$C_1(x, y, p) :\equiv \prod_{z:A} (y = x)(x = z)$$

we have

$$\lambda z.\, \lambda q.\, q : \left(\prod_{z:A} (x = z) \to (x = z) \right) \equiv C_1(x, x, \mathsf{refl}_x)$$

So by path induction, there is a function

$$p \bullet_1 q : (x = z)$$

such that $\mathsf{refl}_x \bullet_1 q \equiv q$.

For the shorter version, we say that by induction it suffices to consider the case where y is x and p is refl_x. Then given some $q : x = z$, we want to construct an element of $x = z$; but this is just q, so induction gives us a function $p \bullet_1 q : x = z$ such that $\mathsf{refl}_x \bullet_1 q :\equiv q$.

```
Definition cat' {A : Type} {x y z : A} (p : x = y) (q : y = z) : x = z.
  induction p. apply q.
Defined.
```

For the second, consider the family

$$C_2(y, z, q) :\equiv \prod_{z:A} (x = y) \to (x = z)$$

and element

$$\lambda z.\, \lambda p.\, p : \left(\prod_{z:A} (x = z) \to (x = z) \right) \equiv C_2(z, z, \mathsf{refl}_z)$$

Induction gives us a function

$$p \bullet_2 q : (x = z)$$

such that

$$p \bullet_2 \mathsf{refl}_z = \mathsf{refl}_z$$

```
Definition cat'' {A:Type} {x y z : A} (p : x = y) (q : y = z) : x = z.
  induction q. apply p.
Defined.
```

Finally, for $\bullet_3$, we have the construction from the text. Take the type families

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

and

$$E(x, z, q) :\equiv (x = z)$$

Since $E(x, x, \mathsf{refl}_x) \equiv (x = x)$, we have $e(x) :\equiv \mathsf{refl}_x : E(x, x, \mathsf{refl}_x)$, and induction gives us a function

$$d : \left(\prod_{(x,z:A)} \prod_{(q:x=z)} (x = z) \right) \equiv \prod_{x:A} D(x, x, \mathsf{refl}_x)$$

So path induction again gives us a function

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

Which we can write $p \bullet_3 q : (x = z)$. By the definitional equality of f, we have that $\mathsf{refl}_x \bullet q \equiv d(x)$, and by the definitional equality of d, we have $\mathsf{refl}_x \bullet \mathsf{refl}_x \equiv \mathsf{refl}_x$.

```
Definition cat''' {A:Type} {x y z : A} (p : x = y) (q : y = z) : x = z.
  induction p, q. reflexivity.
```

```
Defined.
```

Now, to show that $p \cdot_1 q = p \cdot_2 q = p \cdot_3 q$, which we will do by induction on p and q. For the first pair, we want to construct for every $x, y, z : A$, $p : x = y$, and $q : y = z$, an element of $p \cdot_1 q = p \cdot_2 q$. By induction on p, it suffices to assume that y is x and p is refl_x; similarly, by induction on q it suffices to assume that z is also x and q is refl_x. Then by the computation rule for $\cdot_1$, $\mathsf{refl}_x \cdot_1 \mathsf{refl}_x \equiv \mathsf{refl}_x$, and by the computation rule for $\cdot_2$, $\mathsf{refl}_x \cdot_2 \mathsf{refl}_x \equiv \mathsf{refl}_x$. Thus we have

$$\mathsf{refl}_{\mathsf{refl}_x} : (\mathsf{refl}_x \cdot_1 \mathsf{refl}_x = \mathsf{refl}_x \cdot_2 \mathsf{refl}_x)$$

which provides the necessary data for induction.

Writing this out a bit more fully for practice, we have the family

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

defined by

$$C(x, y, p) :\equiv \prod_{(z:A)} \prod_{(q:y=z)} (p \cdot_1 q = p \cdot_2 q)$$

and in order to apply induction, we need an element of

$$\prod_{x:A} C(x, x, \mathsf{refl}_x) \equiv \prod_{(x,z:A)} \prod_{(q:x=z)} (\mathsf{refl}_x \cdot_1 q = \mathsf{refl}_x \cdot_2 q) \equiv \prod_{(x,z:A)} \prod_{(q:x=z)} (q = \mathsf{refl}_x \cdot_2 q)$$

Define $D(x, z, q) :\equiv (q = \mathsf{refl}_x \cdot_2 q)$. Then

$$\mathsf{refl}_{\mathsf{refl}_x} : D(x, x, \mathsf{refl}_x) \equiv (\mathsf{refl}_x = \mathsf{refl}_x \cdot_2 \mathsf{refl}_x) \equiv (\mathsf{refl}_x = \mathsf{refl}_x)$$

So by induction we have a function $f : \prod_{(x,z:A)} \prod_{(p:x=z)} (q = \mathsf{refl}_x \cdot_2 q)$ with $f(x, x, \mathsf{refl}_x) :\equiv \mathsf{refl}_{\mathsf{refl}_x}$. Thus we have the element required for induction on p, and there is a function

$$f' : \prod_{(x,y,z:A)} \prod_{(p:x=y)} \prod_{q:y=z} (p \cdot_1 q = p \cdot_2 q)$$

which we wanted to show.

```
Theorem cat'_eq_cat'' : ∀ {A:Type} {x y z : A} (p : x = y) (q : y = z),
  (cat' p q) = (cat'' p q).
Proof.
  induction p, q. reflexivity.
Defined.
```

For the next pair, we again use induction. For all $x, y, z : A$, $p : x = y$, and $q : y = z$, we need to construct an element of $p \cdot_2 q = p \cdot_3 q$. By induction on p and q, it suffices to consider the case where y and z are x and p and q are refl_x. Then $(\mathsf{refl}_x \cdot_2 \mathsf{refl}_x = \mathsf{refl}_x \cdot_3 \mathsf{refl}_x) \equiv (\mathsf{refl}_x = \mathsf{refl}_x)$, and $\mathsf{refl}_{\mathsf{refl}_x}$ inhabits this type.

```
Theorem cat''_eq_cat''' : ∀ {A:Type} {x y z : A} (p : x = y) (q : y = z),
  (cat'' p q) = (cat''' p q).
Proof.
  induction p, q. reflexivity.
Defined.
```

The third proof goes exactly the same.

```
Theorem cat'_eq_cat''' : ∀ {A:Type} {x y z : A} (p : x = y) (q : y = z),
  (cat' p q) = (cat''' p q).
Proof.
  induction p, q. reflexivity.
Defined.
```

Note that all three of these proofs must end with `Defined` instead of `Qed` if we want to make use of the computational identity (e.g., $p \cdot_1 \mathsf{refl}_x \equiv p$) that they produce, as we will in the next exercise.

Exercise 2.2 (p. 103) Show that 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 $(p \cdot_1 q)$, $(p \cdot_2 q)$, and $(p \cdot_3 q)$, then the concatenated equality

$$(p \cdot_1 q) = (p \cdot_2 q) = (p \cdot_3 q)$$

is equal to the equality $(p \cdot_1 q) = (p \cdot_3 q)$.

Solution Let $x, y, z : A$, $p : x = y$, $q : y = z$, and let $r_{12} : (p \cdot_1 q = p \cdot_2 q)$, $r_{23} : (p \cdot_2 q = p \cdot_3 q)$, and $r_{13} : (p \cdot_1 q = p \cdot_3 q)$ be the proofs from the last exercise. We want to show that $r_{12} \cdot r_{23} = r_{13}$, where $\cdot = \cdot_3$ is the concatenation operation from the book. By induction on p and q, it suffices to consider the case where y and z are x and p and q are refl_x. Then we have $r_{12} \equiv \mathsf{refl}_{\mathsf{refl}_x}$, $r_{23} \equiv \mathsf{refl}_{\mathsf{refl}_x}$, and $r_{13} \equiv \mathsf{refl}_{\mathsf{refl}_x}$ by the definitions. But then the type we're trying to witness is

$$(\mathsf{refl}_{\mathsf{refl}_x} \cdot \mathsf{refl}_{\mathsf{refl}_x} = \mathsf{refl}_{\mathsf{refl}_x}) \equiv (\mathsf{refl}_{\mathsf{refl}_x} = \mathsf{refl}_{\mathsf{refl}_x})$$

from the definition of $\cdot$, so $\mathsf{refl}_{\mathsf{refl}_{\mathsf{refl}_x}}$ is our witness.

```
Theorem comm_triangle : ∀ (A:Type) (x y z : A) (p : x = y) (q : y = z),
  (cat'_eq_cat'' p q) @ (cat''_eq_cat''' p q) = (cat'_eq_cat''' p q).
Proof.
  induction p, q. reflexivity.
Qed.
```

Exercise 2.3 (p. 103) Give a fourth, different proof of Lemma 2.1.2, and prove that it is equal to the others.

Solution Let $x, y : A$ and $p : x = y$. Rather than fixing some q and constructing an element of $x = z$ out of that, we can directly construct an element of

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

by induction on p. It suffices to consider the case where y is x and p is a refl_x, which then makes it easy to produce such an element; namely,

$$\lambda z.\, \mathsf{id}_{x=z} : \prod_{z:A} (x = z) \to (x = z)$$

Induction then gives us a function $p \cdot_4 q : (x = z)$ such that $\lambda q.\, (\mathsf{refl}_x \cdot_4 q) :\equiv \mathsf{id}_{x=z}$.

```
Definition cat'''' {A:Type} {x y z : A} (p : x = y) (q : y = z) : x = z.
  generalize q. generalize z.
  induction p. trivial.
Defined.
```

To prove that it's equal to the others, we can just show that it's equal to $\cdot$ and then use concatenation. Again by induction on p and q, it suffices to consider the case where y and z are x and p and q are refl_x. Then we have

$$((\mathsf{refl}_x \cdot_3 \mathsf{refl}_x) = (\mathsf{refl}_x \cdot_4 \mathsf{refl}_x)) \equiv (\mathsf{refl}_x = \mathsf{id}_{x=x}(\mathsf{refl}_x)) \equiv (\mathsf{refl}_x = \mathsf{refl}_x)$$

So $\mathsf{refl}_{\mathsf{refl}_x}$ is again our witness.

```
Theorem cat'''_eq_cat'''' : ∀ {A:Type} {x y z : A} (p : x = y) (q : y = z),
  (cat''' p q) = (cat'''' p q).
Proof.
  induction p, q. reflexivity.
Qed.
```

Exercise 2.4 (p. 103) Define, by induction on n, a general notion of n-dimensional path in a type A, simultaneously with the type of boundaries for such paths.

Solution A 0-path in A is an element $x : A$, so the type of 0-paths is just A. If p and q are n-paths, then so is $p = q$. In the other direction, the boundary of a 0-path is empty, and the boundary of an $n+1$ path is an n-path.

```
Fixpoint npath (A:Type) (n:nat) : Type :=
  match n with
    | O ⇒ A
    | S n' ⇒ {p : (boundary A n') & {q : (boundary A n') & p = q}}
  end

with boundary (A:Type) (n:nat) : Type :=
  match n with
    | O ⇒ Empty
    | S n' ⇒ (npath A n')
  end.
```

Exercise 2.5 (p. 103) Prove that the functions

$$(f(x) = f(y)) \to (p_*(f(x)) = f(y)) \qquad \text{and}$$
$$(p_*(f(x)) = f(y)) \to (f(x) = f(y))$$

are inverse equivalences.

Solution I take it that "inverse equivalences" means that each of the maps is the quasi-inverse of the other. Suppose that $x, y : A$, $p : x = y$, and $f : A \to B$. Then we have the objects

$$\mathsf{ap}_f(p) : (f(x) = f(y)) \qquad \mathsf{transportconst}_p^B(f(x)) : (p_*(f(x)) = f(x))$$

thus

$$\left(\mathsf{transportconst}_p^B(f(x)) \cdot -\right) : (f(x) = f(y)) \to (p_*(f(x)) = f(y))$$
$$\left((\mathsf{transportconst}_p^B(f(x)))^{-1} \cdot -\right) : (p_*(f(x)) = f(y)) \to (f(x) = f(y))$$

Which are our maps. Composing the first with the second, we obtain an element

$$\left(\mathsf{transportconst}_p^B(f(x)) \cdot \left((\mathsf{transportconst}_p^B(f(x)))^{-1} \cdot -\right)\right)$$

of $f(x) = f(y)$. Using Lemma 2.1.4, we can show that this is homotopic to the identity:

$$\begin{aligned}
&\prod_{q:f(x)=f(y)} \left(\mathsf{transportconst}_p^B(f(x)) \cdot \left((\mathsf{transportconst}_p^B(f(x)))^{-1} \cdot q\right) = q\right) \\
&= \prod_{q:f(x)=f(y)} \left(\left(\mathsf{transportconst}_p^B(f(x)) \cdot (\mathsf{transportconst}_p^B(f(x)))^{-1}\right) \cdot q = q\right) \\
&= \prod_{q:f(x)=f(y)} (q = q)
\end{aligned}$$

which is inhabited by refl_q. The same argument goes the other way, so this concatenation is an equivalence.

```
Definition eq2_3_6 {A B : Type} {x y : A} (f : A → B) (p : x = y) (q : f x = f y) :
    (@transport _ (fun _ ⇒ B) _ _ p (f x) = f y) :=
```

```
  (transport_const p (f x)) @ q.
Definition eq2_3_7 {A B : Type} {x y : A} (f : A → B) (p : x = y)
  (q : @transport _ (fun _ ⇒ B) _ _ p (f x) = f y) :
  (f x = f y) :=
  (transport_const p (f x))^ @ q.
Lemma isequiv_transportconst (A B:Type) (x y z : A) (f : A → B) (p : x = y) :
  IsEquiv (eq2_3_6 f p).
Proof.
  refine (isequiv_adjointify _ (eq2_3_7 f p) _ _); intro q;
    unfold eq2_3_6, eq2_3_7; path_induction; reflexivity.
Qed.
```

Exercise 2.6 (p. 103) Prove that if $p : x = y$, then the function $(p \cdot -) : (y = z) \to (x = z)$ is an equivalence.

Solution Suppose that $p : x = y$. To show that $(p \cdot -)$ is an equivalence, we need to exhibit a quasi-inverse to it. This is a triple (g, α, β) of a function $g : (x = z) \to (y = z)$ and homotopies $\alpha : (p \cdot -) \circ g \sim \mathsf{id}_{x=z}$ and $\beta : g \circ (p \cdot -) \sim \mathsf{id}_{y=z}$. For g, we can take $(p^{-1} \cdot -)$. For the homotopies, we can use the results of Lemma 2.1.4. So we have

$$((p \cdot -) \circ g) \sim \mathsf{id}_{x=z} \equiv \prod_{q:x=z} (p \cdot (p^{-1} \cdot q) = q) = \prod_{q:x=z} ((p \cdot p^{-1}) \cdot q = q) = \prod_{q:x=z} (\mathsf{refl}_x \cdot q = q) = \prod_{q:x=z} (q = q)$$

which is inhabited by refl_q and

$$(g \circ (p \cdot -)) \sim \mathsf{id}_{y=z} \equiv \prod_{q:y=z} (p^{-1} \cdot (p \cdot q) = q) = \prod_{q:y=z} ((p^{-1} \cdot p) \cdot q = q) = \prod_{q:y=z} (\mathsf{refl}_y \cdot q = q) = \prod_{q:y=z} (q = q)$$

which is inhabited by refl_q. So $(p \cdot -)$ has a quasi-inverse, hence it is an equivalence.

```
Lemma isequiv_eqcat (A:Type) (x y z : A) (p : x = y) : IsEquiv (fun q:(y=z) ⇒ p @ q).
Proof.
  refine (isequiv_adjointify _ (fun q:(x=z) ⇒ p^ @ q) _ _); intro q;
    by path_induction.
Qed.
```

Exercise 2.7 (p. 104) State and prove a generalization of Theorem 2.6.5 from cartesian products to Σ-types.

Solution Suppose that we have types A and A' and type families $B : A \to \mathcal{U}$ and $B' : A' \to \mathcal{U}$, along with a function $g : A \to A'$ and a dependent function $h : \prod_{(x:A)} B(x) \to B'(f(x))$. We can then define a function $f : (\sum_{(x:A)} B(x)) \to (\sum_{(x:A')} B'(x))$ by $f(x) :\equiv (g(\mathsf{pr}_1 x), h(\mathsf{pr}_1 x, \mathsf{pr}_2 x))$. Let $x, y : \sum_{(a:A)} B(a)$, and suppose that $p : \mathsf{pr}_1 x = \mathsf{pr}_1 y$ and that $q : p_*(\mathsf{pr}_2 x) = \mathsf{pr}_2 y$. The left-side of Theorem 2.6.5 generalizes directly to $f(\mathsf{pair}^=(p, q))$, where now $\mathsf{pair}^=$ is given by the backward direction of Theorem 2.7.2.

The right hand side is trickier. It ought to represent the application of g and h, followed by the application of $\mathsf{pair}^=$, as Theorem 2.6.5 does. Applying g produces the first argument to $\mathsf{pair}^=$, $\mathsf{ap}_g(p) \equiv g(p)$. For h, we'll need to construct the right object. We need one of type

$$(g(p))_*(h(\mathsf{pr}_1 x, \mathsf{pr}_2 x)) = h(\mathsf{pr}_1 y, \mathsf{pr}_2 y)$$

Which we'll construct by induction. It suffices to consider the case where $x \equiv (a, b)$, $y \equiv (a', b')$, $p \equiv \mathsf{refl}_a$, and $q \equiv \mathsf{refl}_b$. Then we need an object of type

$$\big((g(\mathsf{refl}_a))_*(h(a, b)) = h(a', b')\big) \equiv \big(h(a, b) = h(a', b')\big)$$

which we can easily construct by applying h to p and q. So by induction, we have an object

$$T(h, p, q) : (g(p))_*(h(\mathsf{pr}_1 x, \mathsf{pr}_2 x)) = h(\mathsf{pr}_1 y, \mathsf{pr}_2 y)$$

such that $T(h, \mathsf{refl}_a, \mathsf{refl}_b) \equiv \mathsf{refl}_{h(a,b)}$.

Now we can state the generalization. We show that

$$f(\mathsf{pair}^=(p, q)) = \mathsf{pair}^=(g(p), T(h, p, q))$$

by induction. So let $x \equiv (a, b)$, $y \equiv (a', b')$, $p \equiv \mathsf{refl}_a$, and $q \equiv \mathsf{refl}_b$. Then we need to show that

$$\mathsf{refl}_{f((a,b))} = \mathsf{refl}_{(g(a), h(a,b))}$$

But from the definition of f, this is a judgemental equality. So we're done.

```
Module Exercise2_7.

Definition T {A A' : Type} {B : A → Type} {B' : A' → Type}
           {g : A → A'} (h : ∀ a, B a → B' (g a))
           {x y : {a:A & B a}}
           (p : x.1 = y.1) (q : p # x.2 = y.2)
  : (ap g p) # (h x.1 x.2) = h y.1 y.2.
Proof.
  destruct x as [a b], y as [a' b'].
  simpl in p. induction p.
  simpl in *. f_ap.
Defined.

Definition functor_sigma {A A' : Type} {B : A → Type} {B' : A' → Type}
           (g : A → A') (h : ∀ a, B a → B' (g a))
  : {x : A & B x} → {x : A' & B' x}
  := fun z ⇒ (g z.1; h z.1 z.2).

Theorem ex2_7 : ∀ {A A' : Type} {B : A → Type} {B' : A' → Type}
                  (g : A → A') (h : ∀ a, B a → B' (g a))
                  (x y : {a:A & B a})
                  (p : x.1 = y.1) (q : p # x.2 = y.2),
  ap (functor_sigma g h) (path_sigma B x y p q)
  = path_sigma B' (functor_sigma g h x) (functor_sigma g h y)
               (ap g p) (T h p q).
intros.
destruct x as [a b], y as [a' b'].
simpl in p. induction p.
simpl in q. induction q.
reflexivity.
Defined.

End Exercise2_7.
```

Exercise 2.8 (p. 104) State and prove an analogue of Theorem 2.6.5 for coproducts.

Solution Let $A, A', B, B' : \mathcal{U}$, and let $g : A \to A'$ and $h : B \to B'$. These allow us to construct a function $f : A + B \to A' + B'$ given by

$$f(\mathsf{inl}(a)) :\equiv \mathsf{inl}'(g(a)) \qquad f(\mathsf{inr}(b)) :\equiv \mathsf{inr}'(h(b))$$

Now, we want to show that ap_f is functorial, which requires something corresponding to $\mathsf{pair}^=$. The type of this function will vary depending on which two $x, y : A + B$ we consider. Suppose that $p : x = y$; there are four cases:

- $x = \mathsf{inl}(a_1)$ and $y = \mathsf{inl}(a_2)$. Then $\mathsf{pair}^=$ is given by $\mathsf{ap}_{\mathsf{inl}}$, and we must show that

  $$f(\mathsf{inl}(p)) = \mathsf{inl}'(g(p))$$

  which is easy with path induction; it suffices to consider $p \equiv \mathsf{refl}_a$, which reduces our equality to

  $$\mathsf{refl}_{f(\mathsf{inl}(a))} = \mathsf{refl}_{\mathsf{inl}'(g(a))}$$

  and this is a judgemental equality, given the definition of f.
- $x = \mathsf{inl}(a)$ and $y = \mathsf{inr}(b)$. Then by 2.12.3 $(x = y) \simeq \mathbf{0}$, and p allows us to construct anything we like.
- $x = \mathsf{inr}(b)$ and $y = \mathsf{inl}(a)$ proceeds just as in the previous case.
- $x = \mathsf{inr}(b)$ and $y = \mathsf{inr}(b)$ proceeds just as in the first case.

Since these are all the cases, we've proven an analogue to Theorem 2.6.5 for coproducts. A closer analogue would not involve $p : x = y$, but rather equalities $p : a =_A a'$ and $q : b =_B b'$, and then would show that for all the different cases, ap_f is functorial. However, this follows from the version proven, which is more simply stated.

```
Theorem ap_functor_sum :
  ∀ (A A' B B' : Type) (g : A → A') (h : B → B') (x y : A+B) (p : x = y),
  ap (functor_sum g h) p
  = path_sum (functor_sum g h x) (functor_sum g h y)
             (path_sum_inv (ap (functor_sum g h) p)).
Proof.
  intros.
  destruct x as [a | b], y as [a' | b'].
  induction p. reflexivity.
  contradiction ((equiv_path_sum _ _)^-1 p).
  contradiction ((equiv_path_sum _ _)^-1 p).
  induction p. reflexivity.
Defined.
```

Exercise 2.9 (p. 104) Prove that coproducts have the expected universal property,

$$(A + B \to X) \simeq (A \to X) \times (B \to X).$$

Can you generalize this to an equivalence involving dependent functions?

Solution To define the ex2_9_f map, let $h : A + B \to X$ and define $f : (A + B \to X) \to (A \to X) \times (B \to X)$ by

$$f(h) :\equiv (\lambda a.\, h(\mathsf{inl}(a)), \lambda b.\, h(\mathsf{inr}(b)))$$

To show that f is an equivalence, we'll need a quasi-inverse, given by

$$g(h) :\equiv \mathsf{rec}_{A+B}(X, \mathsf{pr}_1 h, \mathsf{pr}_2 h)$$

As well as the homotopies $\alpha : f \circ g \sim \mathsf{id}_{(A\to X)\times(B\to X)}$ and $\beta : g \circ f \sim \mathsf{id}_{A+B\to X}$. For α we need a witness to

$$\begin{aligned}
&\prod_{h:(A\to X)\times(B\to X)} (f(g(h)) = \mathsf{id}_{(A\to X)\times(B\to X)}(h)) \\
&\equiv \prod_{h:(A\to X)\times(B\to X)} ((\lambda a.\, \mathsf{rec}_{A+B}(X, \mathsf{pr}_1 h, \mathsf{pr}_2 h, \mathsf{inl}(a)), \lambda b.\, \mathsf{rec}_{A+B}(X, \mathsf{pr}_1 h, \mathsf{pr}_2 h, \mathsf{inr}(b))) = h) \\
&\equiv \prod_{h:(A\to X)\times(B\to X)} ((\mathsf{pr}_1 h, \mathsf{pr}_2 h) = h)
\end{aligned}$$

and this is inhabited by uppt. For β, we need an inhabitant of

$$\prod_{h:A+B\to X} (g(f(h)) = \mathsf{id}_{A+B\to X}(h))$$
$$\equiv \prod_{h:A+B\to X} (\mathsf{rec}_{A+B}(X, \lambda a.\, h(\mathsf{inl}(a)), \lambda b.\, h(\mathsf{inr}(b))) = h)$$

which, assuming function extensionality, is inhabited. So (g, α, β) is a quasi-inverse to f, giving the universal property.

```
Definition ex2_9_f {A B X : Type} (h : (A + B → X)) : (A→X) × (B→X) :=
  (h o inl , h o inr).

Definition ex2_9_g {A B X : Type} (h : (A→X) × (B→X)) : A + B → X :=
  fun x ⇒ match x with
            | inl a ⇒ (fst h) a
            | inr b ⇒ (snd h) b
          end.

Theorem ex2_9 : ∀ A B X, (A + B→X) ≃ (A→X) × (B→X).
Proof.
  intros. refine (equiv_adjointify ex2_9_f ex2_9_g _ _); intro h.
  destruct h as (x, y). reflexivity.
  apply path_forall. intro z. destruct z; reflexivity.
Qed.
```

All of this generalizes directly to the case of dependent functions.

```
Definition ex2_9_f' {A B : Type} {C: A + B → Type} (h : ∀ (p:A + B), C p)
  : (∀ a:A, C(inl a)) × (∀ b:B, C(inr b)) :=
(fun _ ⇒ h (inl _), fun _ ⇒ h (inr _)).

Definition ex2_9_g' {A B : Type} {C: A + B → Type}
           (h : (∀ a:A, C(inl a)) × (∀ b:B, C(inr b))) :
  ∀ (p:A + B), C p :=
 fun _ ⇒ match _ as s return (C s) with
           | inl a ⇒ fst h a
           | inr b ⇒ snd h b
         end.

Theorem ex2_9' : ∀ A B C,
  (∀ (p:A + B), C(p)) ≃ (∀ a:A, C(inl a)) × (∀ b:B, C(inr b)).
Proof.
  intros.
  refine (equiv_adjointify ex2_9_f' ex2_9_g' _ _); unfold ex2_9_f', ex2_9_g'.
  intro. destruct x. apply path_prod; simpl;
  apply path_forall; unfold pointwise_paths; reflexivity.
  intro. apply path_forall; intro p. destruct p; reflexivity.
Qed.
```

Exercise 2.10 (p. 104) Prove that Σ-types are "associative", in that for any $A : \mathcal{U}$ and families $B : A \to \mathcal{U}$ and $C : (\sum_{(x:A)} B(x)) \to \mathcal{U}$, we have

$$\left(\sum_{(x:A)} \sum_{(y:B(x))} C((x,y))\right) \simeq \left(\sum_{p:\sum_{(x:A)} B(x)} C(p)\right)$$

Solution The map

$$f(a, b, c) :\equiv ((a, b), c)$$

where $a : A$, $b : B(a)$, and $c : C((a, b))$ is an equivalence. For a quasi-inverse, we have

$$g(p, c) :\equiv (\mathsf{pr}_1 p, \mathsf{pr}_2 p, c)$$

As proof, by induction we can consider the case where $p \equiv (a, b)$. Then we have

$$f(g((a, b), c)) = f(a, b, c) = ((a, b), c)$$

and

$$g(f(a, b, c)) = g((a, b), c) = (a, b, c)$$

So f is an equivalence.

```
Definition ex2_10_f {A : Type} {B : A → Type} {C : {x:A & B x} → Type} :
  {x:A & {y : B x & C (x; y)}} → {p : {x:A & B x} & C p}.
  intro abc. destruct abc as [a [b c]]. apply ((a; b); c).
Defined.

Definition ex2_10_g {A : Type} {B : A → Type} {C : {x:A & B x} → Type} :
  {p : {x:A & B x} & C p} → {x:A & {y : B x & C (x; y)}}.
  intro abc. destruct abc as [[a b] c].
  ∃ a; ∃ b; apply c.
Defined.

Theorem ex2_10 : ∀ A B C, IsEquiv(@ex2_10_f A B C).
Proof.
  intros.
  refine (isequiv_adjointify ex2_10_f ex2_10_g _ _);
  unfold ex2_10_f, ex2_10_g; intro abc;
  [ destruct abc as [[a b] c] | destruct abc as [a [b c]]];
  reflexivity.
Qed.
```

Exercise 2.11 (p. 104) A (homotopy) commutative square

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

consists of functions f, g, h, and k as shown, together with a path $f \circ h = g \circ k$. Note that this is exactly an element of the pullback $(P \to A) \times_{P \to C} (P \to B)$ as defined in 2.15.11. A commutative square is called a (homotopy) pullback square if for any X, the induced map

$$(X \to P) \to (X \to A) \times_{X \to C} (X \to B)$$

is an equivalence. Prove that the pullback $P :\equiv A \times_C B$ defined in 2.15.11 is the corner of a pullback square.

Solution I'll start using the usual notation pr_n^j for the jth projection from an n-tuple. So, for example, $\mathsf{pr}_3^2 :\equiv \mathsf{pr}_1 \circ \mathsf{pr}_2$. To show that P is the corner of a pullback square, we need to produce the three other corners and show that it is a pullback. Given $f : A \to C$ and $G : B \to C$, we define

$$P :\equiv \sum_{(a:A)} \sum_{(b:B)} (f(a) = g(b))$$

I claim this is the corner of the following pullback square:

$$\begin{tikzcd} P \arrow[r, "\mathsf{pr}_3^2"] \arrow[d, "\mathsf{pr}_3^1"'] & B \arrow[d, "g"] \\ A \arrow[r, "f"'] & C \end{tikzcd}$$

To show that it's commutative, it suffices to consider an element (a, b, p) of P. We then have

$$g(\mathsf{pr}_3^2(a, b, p)) = g(b) = f(a) = f(\mathsf{pr}_3^1(a, b, p))$$

making the square commutative.

To show that it has the required universal property, we need to construct the equivalence. Suppose that $h : X \to P$. Then we can compose it in either direction around the square to give

$$f \circ \mathsf{pr}_3^1 \circ h : X \to C \qquad\qquad g \circ \mathsf{pr}_3^2 \circ h : X \to C$$

If we can show that these two maps are equal, then we can produce an element of $(X \to A) \times_{X \to C} (X \to B)$. And we can, using function extensionality. Suppose that $x : X$. Then $h(x) : P$, which by induction we can assume is of the form $h(x) \equiv (a, b, p)$, with $p : f(a) = g(b)$. This means that

$$f(\mathsf{pr}_3^1(h(x))) \equiv f(\mathsf{pr}_3^1(a, b, p)) \equiv f(a)$$

and

$$g(\mathsf{pr}_3^2(a, b, c)) \equiv g(b)$$

and p proves that these are equal. So by function extensionality,

$$f \circ \mathsf{pr}_3^1 \circ h = g \circ \mathsf{pr}_3^2 \circ h$$

meaning that we can define

$$h \mapsto (\mathsf{pr}_3^1 \circ h, \mathsf{pr}_3^2 \circ h, \mathsf{funext}(\mathsf{pr}_3^3 \circ h))$$

giving the forward map $(X \to P) \to (X \to A) \times_{X \to C} (X \to B)$.

We now need to exhibit a quasi-inverse. Suppose that $h' : (X \to A) \times_{X \to C} (X \to B)$. By induction, we may assume that $h' = (h_A, h_B, q)$, where $q : f \circ h_A = g \circ h_B$. We want to construct a function $X \to P$, so suppose that $x : X$. Then we can construct an element of P like so:

$$h(x) :\equiv (h_A(x), h_B(x), \mathsf{happly}(q)(x))$$

Note that this expression is well-typed, since $h_A(x) : A$, $h_B(x) : B$, and $\mathsf{happly}(q)(x) : f(h_A(x)) = g(h_B(x))$.

In order to show that this is a quasi-inverse, we need to show that the two possible compositions are homotopic to the identity. Suppose that $h : X \to P$; then applying the forward and backward constructions gives

$$(x \mapsto (\mathsf{pr}_3^1(h(x)), \mathsf{pr}_3^2(h(x)), \mathsf{happly}(\mathsf{funext}(\mathsf{pr}_3^3 \circ h))(x))) \equiv (x \mapsto (\mathsf{pr}_3^1(h(x)), \mathsf{pr}_3^2(h(x)), \mathsf{pr}_3^3(h(x))))$$

which by function extensionality is clearly equal to h.

For the other direction, suppose that $h' : (X \to A) \times_{X \to C} (X \to B)$, which by induction we may suppose is of the form (h_A, h_B, p). Going back and forth gives

$$\begin{aligned}\Big(&\mathsf{pr}_3^1 \circ (x \mapsto (h_A(x), h_B(x), \mathsf{happly}(p)(x))),\\ &\quad \mathsf{pr}_3^2 \circ (x \mapsto (h_A(x), h_B(x), \mathsf{happly}(p)(x))),\\ &\quad \mathsf{funext}(\mathsf{pr}_3^3 \circ (x \mapsto (h_A(x), h_B(x), \mathsf{happly}(p)(x))))\Big)\end{aligned}$$

applying function extensionality again results in

$$(h_A, h_B, \mathsf{funext}(\mathsf{happly}(p))) \equiv (h_A, h_B, p)$$

So we have an equivalence.

```
Section Exercise2_11.

Variables (A B C X : Type) (f: A → C) (g: B → C).

Definition P := {a:A & {b:B & f a = g b}}.
Definition funpull := {h:X→A & {k:X→B & f o h = g o k}}.

Definition pi1 (p : P) : A := p.1.
Definition pi2 (p : P) : B := p.2.1.

Definition ex2_11_f `{Funext} : (X → P) → funpull.
  intro h.
  refine (pi1 o h; (pi2 o h; _)).
  apply path_forall; intro.
  exact (h x).2.2.
Defined.

Definition ex2_11_g : funpull → (X → P).
  intros h x.
  refine (h.1 x; (h.2.1 x; _)).
  exact (apD10 h.2.2 x).
Defined.

Theorem ex2_11 `{Funext} : (X → P) ≃ funpull.
  refine (equiv_adjointify ex2_11_f ex2_11_g _ _).

  (*  alpha *)
  unfold Sect, ex2_11_g, ex2_11_f, path_forall; simpl.
  destruct 0 as [f' [g' p]]; simpl. f_ap. f_ap.
  apply (ap apD10)^-1.
  apply eisretr.

  (*  beta *)
  unfold Sect, ex2_11_g, ex2_11_f, path_forall; intro h; simpl.
  apply path_forall; intro x.
  repeat (apply path_sigma_uncurried; ∃ idpath; simpl).
  change (h x).2.2 with ((fun x' ⇒ (h x').2.2) x); f_ap.
  apply eisretr.
Qed.

End Exercise2_11.
```

***Exercise 2.12 (p. 104)** Suppose given two commutative squares

$$\begin{tikzcd} A \arrow[r] \arrow[d] & C \arrow[r] \arrow[d] & E \arrow[d] \\ B \arrow[r] & D \arrow[r] & F \end{tikzcd}$$

and suppose that the right-hand square is a pullback square. Prove that the left-hand square is a pullback square if and only if the outer rectangle is a pullback square.

Solution Label the arrows

$$\begin{tikzcd} A \arrow[r] \arrow[d] & C \arrow[r] \arrow[d] & E \arrow[d, "f"] \\ B \arrow[r, "h"'] & D \arrow[r, "g"'] & F \end{tikzcd}$$

and suppose that the right square is a pullback. That is, for all X there is an equivalence

$$e : (X \to C) \simeq (X \to D) \times_{X \to F} (X \to E)$$

There are two obvious maps

$$(X \to A) \to (X \to B) \times_{X \to D} (X \to C)$$
$$(X \to A) \to (X \to B) \times_{X \to F} (X \to E)$$

and we want to show that the first is an equivalence iff the second is. To do this, we show that

$$(X \to B) \times_{X \to D} (X \to C) \simeq (X \to B) \times_{X \to F} (X \to E)$$

and then use the fact that $\simeq$ is an equivalence relation.

So suppose that $k : (X \to B) \times_{X \to D} (X \to C)$. We can construct two maps of the correct signature by composition:

$$\left(\mathsf{pr}_3^1(k), \mathsf{pr}_3^2(e(\mathsf{pr}_3^2(k))), -\right) : (X \to B) \times_{X \to F} (X \to E)$$

So we just need a proof that these commute correctly. We have

$$\mathsf{pr}_3^3(e(\mathsf{pr}_3^2(k))) : f \circ g^* f \circ \mathsf{pr}_3^2(k) = g \circ f^* g \circ \mathsf{pr}_3^2(k)$$
$$\mathsf{pr}_3^3(k) : h \circ \mathsf{pr}_3^1(k) = f^* g \circ \mathsf{pr}_3^2(k)$$

which we can straightforwardly combine to obtain the desired equality:

$$\mathsf{pr}_3^3(e(\mathsf{pr}_3^2(k))) \cdot \mathsf{ap}_{g \circ -}(\mathsf{pr}_3^3(k))^{-1} : f \circ g^* f \circ \mathsf{pr}_3^2(k) = g \circ h \circ \mathsf{pr}_3^1(k)$$

giving us a forward map

$$\phi : k \mapsto \left(\mathsf{pr}_3^1(k), \mathsf{pr}_3^2(e(\mathsf{pr}_3^2(k))), \mathsf{pr}_3^3(e(\mathsf{pr}_3^2(k))) \cdot \mathsf{ap}_{g \circ -}(\mathsf{pr}_3^3(k))^{-1}\right)$$

To go the other way, suppose that $k : (X \to B) \times_{X \to F} (X \to E)$. As before, the first entry of our output will just be $\mathsf{pr}_3^1(k)$ again. The second we'll have to build using e^{-1}:

$$e^{-1}\left(h \circ \mathsf{pr}_3^1(k), \mathsf{pr}_3^2(k), \mathsf{pr}_3^3(k)\right) : X \to C$$

And now we must prove that we have the needed commutation relation:

$$h \circ \mathsf{pr}_3^1(k) = f^* g \circ e^{-1}\left(h \circ \mathsf{pr}_3^1(k), \mathsf{pr}_3^2(k), \mathsf{pr}_3^3(k)\right)$$

note that by definition of e, this is judgementally equivalent to

$$h \circ \mathsf{pr}_3^1(k) = \mathsf{pr}_3^1\left(e\left(e^{-1}\left(h \circ \mathsf{pr}_3^1(k), \mathsf{pr}_3^2(k), \mathsf{pr}_3^3(k)\right)\right)\right)$$

which, since $\alpha : e \circ e^{-1} \sim \mathsf{id}$, is equal to

$$h \circ \mathsf{pr}_3^1(k) = \mathsf{pr}_3^1\left(h \circ \mathsf{pr}_3^1(k), \mathsf{pr}_3^2(k), \mathsf{pr}_3^3(k)\right)$$

a definitional equality. So our backward map is

$$\phi^{-1} : k \mapsto \left(\mathsf{pr}_3^1(k), e^{-1}\left(h \circ \mathsf{pr}_3^1(k), \mathsf{pr}_3^2(k), \mathsf{pr}_3^3(k)\right), \mathsf{ap}_{\mathsf{pr}_3^1}\alpha^{-1}\right)$$

To show that these are quasi-inverses, suppose that $k : (X \to B) \times_{X \to F} (X \to E)$, which we may assume is of the form (k_1, k_2, p). Then

$$\phi(\phi^{-1}(k)) \equiv \left(k_1, \mathsf{pr}_3^2\left(e\left(e^{-1}\left(h \circ k_1, k_2, p\right)\right)\right), \mathsf{pr}_3^3\left(e\left(e^{-1}\left(h \circ k_1, k_2, p\right)\right)\right) \cdot \mathsf{ap}_{g \circ -}\left(\mathsf{ap}_{\mathsf{pr}_3^1}\alpha^{-1}\right)^{-1}\right)$$

Which, since e is an equivalence and by the way ap interacts with inverses, is equal (up to homotopy) to

$$\phi(\phi^{-1}(k)) = \left(k_1, k_2, p \cdot \mathsf{ap}_{g \circ -}\mathsf{ap}_{\mathsf{pr}_3^1}\alpha\right)$$

But $\mathsf{ap}_{g \circ -}\mathsf{ap}_{\mathsf{pr}_3^1}\alpha$ is equal to reflexivity, up to the inverse of the former homotopy. So we have

$$\phi(\phi^{-1}(k)) = k$$

by the universal property of Σ types.

For the other direction, suppose that $k : (X \to B) \times_{X \to D} (X \to C)$, which we may assume is of the form (k_1, k_2, p). Then

$$\phi^{-1}(\phi(k)) \equiv \left(k_1, e^{-1}\left(h \circ k_1, \mathsf{pr}_3^2(e(k_2)), \mathsf{pr}_3^3(e(k_2)) \cdot \mathsf{ap}_{g \circ -}p^{-1}\right), \mathsf{ap}_{\mathsf{pr}_3^1}\alpha^{-1}\right)$$

which, by definition of e, reduces to

$$\phi^{-1}(\phi(k)) \equiv \left(k_1, e^{-1}\left(h \circ k_1, f^* g \circ k_2, \mathsf{pr}_3^3(e(k_2)) \cdot \mathsf{ap}_{g \circ -}p^{-1}\right), \mathsf{ap}_{\mathsf{pr}_3^1}\alpha^{-1}\right)$$

Now, by p, we have $h \circ k_1 = g^* f \circ k_2$, so the second term may be written

$$\phi^{-1}(\phi(k)) = \left(k_1, e^{-1}(e(k_2)), \mathsf{ap}_{\mathsf{pr}_3^1}\alpha^{-1}\right)$$

and since e is an equivalence we have up to homotopy

$$\phi^{-1}(\phi(k)) \equiv \left(k_1, k_2, \mathsf{ap}_{\mathsf{pr}_3^1}\alpha^{-1}\right)$$

which by a similar argument as the previous case gives $\phi^{-1}(\phi(k)) = k$. In neither case is the argument particularly convincing. That's what Coq is for, I guess.

```
Section Exercise2_12.

Context {A B C D E F X : Type} (f : E -> F) (g : D -> F) (h : B -> D)
        (gsf : C -> D) (fsg : C -> E) (r : g o gsf = f o fsg).
```

```
Definition e : (X → C) → {a : X → D & {b : X → E & g ∘ a = f ∘ b}}.
Proof.
  intro k.
  ∃ (gsf ∘ k).
  ∃ (fsg ∘ k).
  apply ((ap (fun z ⇒ z ∘ k) r)).
Defined.

Hypothesis isequiv_e : IsEquiv e.

Definition left_to_outer :
  {a : X → B & {b : X → C & h ∘ a = gsf ∘ b}}
  → {a : X → B & {b : X → E & g ∘ h ∘ a = f ∘ b}}
  := fun k ⇒
       (k.1; ((e k.2.1).2.1; (ap (compose g) k.2.2) @ (e k.2.1).2.2)).

Definition outer_to_left :
  {a : X → B & {b : X → E & g ∘ h ∘ a = f ∘ b}}
  → {a : X → B & {b : X → C & h ∘ a = gsf ∘ b}}
  := fun k ⇒
       (k.1; (e^-1 (h ∘ k.1; k.2); ap pr1 (eisretr e (h ∘ k.1; k.2))^)).

Theorem left_equiv_outer :
  {a : X → B & {b : X → C & h ∘ a = gsf ∘ b}}
  ≃ {a : X → B & {b : X → E & g ∘ h ∘ a = f ∘ b}}.
Proof.
  refine (equiv_adjointify left_to_outer outer_to_left _ _).
  intro k.
  apply path_sigma_uncurried. ∃ 1.
  apply path_sigma_uncurried. simpl.
  transparent assert (H : (fsg ∘ e^-1 (h ∘ k.1; k.2) = k.2.1)).
  change (fsg ∘ e^-1 (h ∘ k.1; k.2))
         with (e (e^-1 (h ∘ k.1; k.2))).2.1.
  change k.2.1 with (pr1 (@pr2 (X → D)
                              (fun a ⇒ {b : X → E & g ∘ a = f ∘ b})
                              (h ∘ k.1; k.2))).
  admit.
  admit.

  (*  outer_to_left o left_to_outer == id *)
  intro k. destruct k as [k1 [k2 p]].
  apply path_sigma_uncurried. ∃ 1. simpl.
  apply path_sigma_uncurried. simpl.
  transparent assert (H : (
    e^-1 (h ∘ k1; (fsg ∘ k2;
                   ap (compose g) p @ ap (fun z : C → F ⇒ z ∘ k2) r))
    = k2
  )).
  apply (ap e)^-1.
  refine ((eisretr e _) @ _). unfold e.
  apply path_sigma_uncurried. simpl.
  ∃ p. simpl.
  apply path_sigma_uncurried. simpl.
  admit. admit.
Admitted.
```

```
End Exercise2_12.
```

Exercise 2.13 (p. 104) Show that $(\mathbf{2} \simeq \mathbf{2}) \simeq \mathbf{2}$.

Solution The result essentially says that $\mathbf{2}$ is equivalent to itself in two ways: the identity provides one equivalence, and negation gives the other. So we first define these. $\mathsf{id}_\mathbf{2}$ is its own quasi-inverse; we have $\mathsf{id}_\mathbf{2} \circ \mathsf{id}_\mathbf{2} \equiv \mathsf{id}_\mathbf{2}$, so $\mathsf{id}_\mathbf{2} \circ \mathsf{id}_\mathbf{2} \sim \mathsf{id}_\mathbf{2}$ easily. $\neg$ is also its own quasi-inverse, since for any x, $\neg\neg x = x$.

To show the result, we need to map $\mathsf{id}_\mathbf{2}$ and $\neg$ onto $\mathbf{2}$ is a quasi-invertible way. But we need to define this map on all of $\mathbf{2} \simeq \mathbf{2}$. So for any $h : \mathbf{2} \simeq \mathbf{2}$, let $f(h) = h(0_\mathbf{2})$, and define $g : \mathbf{2} \to (\mathbf{2} \simeq \mathbf{2})$ by

$$g(0_\mathbf{2}) = \mathsf{id}_\mathbf{2} \qquad g(1_\mathbf{2}) = \neg$$

To show that these are quasi-inverses, note first that whatever else is the case, an equivalence $\mathbf{2} \simeq \mathbf{2}$ can't be a constant function, which we can prove by a case analysis. Each of $f(0_\mathbf{2})$ and $f(1_\mathbf{2})$ is in $\mathbf{2}$, so it is either $0_\mathbf{2}$ or $1_\mathbf{2}$. So we have the cases:

- $f(0_\mathbf{2}) = f(1_\mathbf{2})$, in which case we can apply f^{-1} to either side to get a contradiction, or
- $f(0_\mathbf{2}) = \neg f(1_\mathbf{2})$. In which case we have the result

Showing that $f \circ g \sim \mathsf{id}_\mathbf{2}$ is easy, since we can do it by cases. We have

$$f(g(0_\mathbf{2})) = f(\mathsf{id}_\mathbf{2}) = \mathsf{id}_\mathbf{2}(0_\mathbf{2}) = 0_\mathbf{2}$$
$$f(g(1_\mathbf{2})) = f(\neg) = \neg 0_\mathbf{2} = 1_\mathbf{2}$$

For the other direction, suppose that $h : \mathbf{2} \simeq \mathbf{2}$ and that function extensionality holds. $h(0_\mathbf{2})$ is either $0_\mathbf{2}$ or $1_\mathbf{2}$. If the first, then because h isn't constant we have $h(1_\mathbf{2}) = \neg h(0_\mathbf{2}) = 1_\mathbf{2}$, hence $h = \mathsf{id}_\mathbf{2}$. Furthermore,

$$g(f(h)) = g(h(0_\mathbf{2})) = g(0_\mathbf{2}) = \mathsf{id}_\mathbf{2} = h$$

The same argument works for the other case. So f is an equivalence, and $(\mathbf{2} \simeq \mathbf{2}) \simeq \mathbf{2}$.

```
Lemma id_isequiv : Bool ≃ Bool.
Proof.
  refine (equiv_adjointify idmap idmap (fun _ ⇒ 1) (fun _ ⇒ 1)).
Defined.

Lemma negb_isequiv : Bool ≃ Bool.
Proof.
  refine (equiv_adjointify negb negb _ _);
  intro; destruct x; reflexivity.
Defined.

Definition ex2_13_f (x : Bool ≃ Bool) : Bool := x false.

Definition ex2_13_g (b : Bool) : (Bool ≃ Bool) :=
  if b
  then {| equiv_fun := negb |}
  else {| equiv_fun := idmap |}.

Lemma equiv_not_const (f : Bool → Bool) `{IsEquiv Bool Bool f} :
  f false = negb (f true).
Proof.
  pose proof (eissect f true) as H1.
  pose proof (eissect f false) as H2.
  destruct (f true), (f false);
  try (etransitivity; try (eassumption || (symmetry; eassumption)));
```

```
  try (simpl; reflexivity).
Defined.

Theorem negb_involutive : ∀ b, negb (negb b) = b.
Proof. destruct b; reflexivity. Qed.

Theorem ex2_13 `{Funext} : (Bool ≃ Bool) ≃ Bool.
Proof.
  refine (equiv_adjointify ex2_13_f ex2_13_g _ _);
  unfold Sect, ex2_13_f, ex2_13_g.

  (* alpha *)
  destruct x; reflexivity.

  (* beta *)
  destruct x. pose proof (equiv_not_const equiv_fun) as H1.
  apply path_equiv; apply path_forall; intro x; destruct x; simpl.
  destruct (equiv_fun false); simpl;
    repeat (transitivity (negb (negb (equiv_fun true)));
      [rewrite ← H1; reflexivity | apply negb_involutive]).
  destruct (equiv_fun false); reflexivity.
Qed.
```

Exercise 2.14 (p. 104) Suppose we add to type theory the equality reflection rule which says that if there is an element $p : x = y$, then in fact $x \equiv y$. Prove that for any $p : x = x$ we have $p \equiv \mathsf{refl}_x$.

Solution Suppose that $p : x = x$; we show that $p = \mathsf{refl}_x$, by path induction. It suffices to consider the case where $p \equiv \mathsf{refl}_x$, in which case we have $\mathsf{refl}_{\mathsf{refl}_x} : \mathsf{refl}_x = \mathsf{refl}_x$. Thus $p = \mathsf{refl}_x$ is inhabited, so by the equality reflection rule, $p \equiv \mathsf{refl}_x$.

Exercise 2.15 (p. 105) Show that Lemma 2.10.5 can be strengthened to

$$\mathsf{transport}^B(p, -) =_{B(x)\to B(y)} \mathsf{idtoeqv}(\mathsf{ap}_B(p))$$

without using function extensionality.

Solution By induction on p, we have

$$\begin{aligned}
\mathsf{transport}^B(\mathsf{refl}_{B(x)}, -) &\equiv \mathsf{id}_{B(x)} \\
&\equiv \mathsf{transport}^{X\mapsto X}(\mathsf{refl}_{B(x)}, -) \\
&\equiv \mathsf{transport}^{X\mapsto X}(\mathsf{ap}_B(\mathsf{refl}_x), -) \\
&\equiv \mathsf{idtoeqv}(\mathsf{ap}_B(\mathsf{refl}_x))
\end{aligned}$$

```
Definition idtoeqv {A B : Type} : (A = B) → (A ≃ B).
  intro X.
  refine (equiv_adjointify (transport idmap X) (transport idmap X^) _ _);
  intro b; [ apply (transport_pV idmap X) | apply (transport_Vp idmap X)].
Defined.

Lemma ex2_15 : ∀ (A : Type) (B : A → Type) (x y : A) (p : x = y),
                 transport B p = idtoeqv (ap B p).
Proof.
  intros. unfold idtoeqv. induction p. reflexivity.
Defined.
```

Exercise 2.16 (p. 105) Suppose that rather than function extensionality, we suppose only the existence of an element

$$\mathsf{funext} : \prod_{(A:\mathcal{U})} \prod_{(B:A\to\mathcal{U})} \prod_{f,g:\prod_{(x:A)} B(x)} (f \sim g) \to (f = g)$$

(with no relationship to happly assumed). Prove that in fact, this is sufficient to imply the whole function extensionality axiom (that happly is an equivalence).

Solution Suppose that we have such an element, and let $A : \mathcal{U}$, $B : A \to \mathcal{U}$, and $f, g : \prod_{(x:A)} B(x)$. I will suppress the A and B throughout. If this implies the whole function extensionality axiom, then it must be the case that we can construct the funext from the book, which has a particular computation rule. This is not too difficult; define

$$\mathsf{funext}'(f, g, h) :\equiv \mathsf{funext}(f, g, h) \cdot (\mathsf{funext}(g, g, h))^{-1}$$

Then we have

$$\begin{aligned}
\mathsf{funext}'(f, f, \lambda x.\,\mathsf{refl}_{f(x)}) &\equiv \mathsf{funext}(f, f, \lambda x.\,\mathsf{refl}_{f(x)}) \cdot (\mathsf{funext}(f, f, \lambda x.\,\mathsf{refl}_{f(x)}))^{-1} \\
&\equiv \mathsf{refl}_f
\end{aligned}$$

by Lemma 2.1.4. So now we need to show that funext$'$ is a quasi-inverse to happly. One direction is easy; since happly computes on refl, by induction we have

$$\mathsf{funext}'(f, f, \mathsf{happly}(\mathsf{refl}_f)) \equiv \mathsf{funext}'(f, f, \lambda x.\,\mathsf{refl}_{f(x)}) \equiv \mathsf{refl}_f$$

and thus $\mathsf{funext}'(f, g) \circ h \sim \mathsf{id}_{f=g}$. The other direction is more difficult. We need to show that for all $h : f \sim g$, $\mathsf{happly}(\mathsf{funext}(f, g, h)) = h$. However, since h isn't an inductive type, we can't really do induction on it. In the special case that $g \equiv f$ and $h \equiv \lambda x.\,\mathsf{refl}_{f(x)}$, we have

$$\mathsf{happly}(\mathsf{funext}(f, f, \lambda x.\,\mathsf{refl}_{f(x)})) \equiv \mathsf{happly}(\mathsf{refl}_f) \equiv \lambda x.\,\mathsf{refl}_{f(x)}$$

So if we could find a way to reduce to this case, then we'd have the result. One way to do this is to show that $(g, h) = (f, \lambda x.\,\mathsf{refl}_{f(x)})$; since we'd need to show this for all g and h, this would be the same as showing that the type

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

is contractible, in the sense discussed in Exercise 1.7. From Theorem 2.15.7, this is equivalent to

$$\prod_{(x:A)} \sum_{(y:B(x))} (f(x) = y)$$

So if we can show that this type is contractible, then we can get the reduction to the special case.

Now, we know from the previously-discussed Lemma 3.11.8 that for any x, $\sum_{(y:B(x))} (f(x) = y)$ is contractible. Now we want to apply Lemma 3.11.6, but the proof requires function extensionality, so we'll have to try to recap it. Suppose that $j, k : \prod_{(x:A)} \sum_{(y:B(x))} (f(x) = y)$. For any $x : A$, we have $j(x) = k(x)$ because $\sum_{(y:B(x))} (f(x) = y)$ is contractible. Hence there's some $p : j \sim k$, so $\mathsf{funext}'(j, k, p) : (j = k)$. This means that

$$\prod_{(x:A)} \sum_{(y:B(x))} (f(x) = y)$$

is contractible. So, transporting across the equivalence,

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

is contractible. Since any two contractible types are equivalent, this means

$$\left(\Sigma_{(g:\prod_{(x:A)} B(x))}(f = g)\right) \simeq \left(\Sigma_{(g:\prod_{(x:A)} B(x))}(f \sim g)\right)$$

Since the first is contractible by Lemma 3.11.8. Thus, we've shown that $\mathsf{total}(\mathsf{happly}(f))$, as defined in Definition 4.7.5, is an equivalence. By Theorem 4.7.7, this makes $\mathsf{happly}(f,g)$ an equivalence for all g, proving the result. Fingers crossed that none of the HoTT library lemmas I use depend on Funext or Univalence.

```
Section Exercise2_16.

Variable funext : ∀ (A : Type) (B : A → Type) (f g : ∀ (x:A), B x),
                    (f ∼ g) → (f = g).

Definition funext' {A : Type} {B : A → Type} (f g : ∀ (x:A), B x) :
  (f∼g) → (f=g) :=
  (fun h : (f∼g) ⇒ (funext A B f g h) @ (funext A B g g (fun _ ⇒ 1))^).

Lemma funext'_computes {A : Type} {B : A → Type} (f : ∀ (x:A), B x) :
  funext' f f (fun _ ⇒ 1) = 1.
Proof.
  unfold funext'. rewrite concat_pV. reflexivity.
Defined.

Lemma Lemma3118 {C} : ∀ (c:C), Contr {x : C & c = x}.
Proof.
  intro c. ∃ (c; 1).
  intro x. destruct x as [x p]. path_induction. reflexivity.
Defined.

Lemma Lemma3116 A B f : Contr (∀ x:A, {y : B x & f x = y}).
Proof.
  ∃ (fun x:A ⇒ (f x; 1)).
  intro k. apply (funext' (fun x ⇒ (f x; 1)) k); intro x.
  assert (Contr {y : B x & f x = y}). apply Lemma3118.
  destruct X. destruct center. rewrite ← (contr (k x)).
  apply path_sigma_uncurried. ∃ p.
  induction p. reflexivity.
Defined.

Definition choice {A B f} :
  (∀ (x:A), {y : B x & f x = y}) → {g : ∀ (x:A), B x & f ∼ g}.
  intro k. ∃ (fun x:A ⇒ (k x).1).
  intro x. apply (k x).2.
Defined.

Definition choice_inv {A B f} :
  ({g : ∀ (x:A), B x & f ∼ g}) → (∀ (x:A), {y : B x & f x = y}).
  intros k x. apply (k.1 x; k.2 x).
Defined.

Lemma Theorem2157 {A B f} : IsEquiv(@choice A B f).
Proof.
  refine (isequiv_adjointify choice choice_inv _ _); intro k;
  unfold choice, choice_inv; simpl; [| apply funext'; intro x];
  apply path_sigma_uncurried; ∃ 1; reflexivity.
Defined.

Lemma contr_equiv_commute {A B} : A ≃ B → Contr A → Contr B.
Proof.
```

```
  intros f k.
  ∃ (f (center A)). intro x. transitivity (f (f⁻¹ x)).
  apply (ap f). apply (contr (f⁻¹ x)).
  apply eisretr.
Defined.

Lemma reduce_to_refl {A B f} : Contr {g : ∀ x:A, B x & f ∼ g}.
Proof.
  apply (@contr_equiv_commute (∀ (x:A), {y : B x & f x = y})).
  refine (BuildEquiv _ _ choice Theorem2157).
  apply Lemma3116.
Defined.

Definition total_happly {A B f} :
  {g : ∀ x:A, B x & f = g} → {g : ∀ x:A, B x & f ∼ g}.
  intros. destruct X. ∃ x. apply apD10. apply p.
Defined.

Definition total_happly_inv {A B f} :
  {g : ∀ x:A, B x & f ∼ g} → {g : ∀ x:A, B x & f = g}.
  intros. destruct X. ∃ x. apply funext'. apply p.
Defined.

Lemma total_equivalence {A B f} : IsEquiv(@total_happly A B f).
Proof.
  refine (isequiv_adjointify total_happly total_happly_inv _ _); intro k.
  - assert (Contr {g : ∀ x:A, B x & f ∼ g}). apply reduce_to_refl.
    destruct X. rewrite (contr k)^.
    apply (contr (total_happly (total_happly_inv center)))^.
  - assert (Contr {g : ∀ x:A, B x & f = g}). apply Lemma3118.
    destruct X. rewrite (contr k)^.
    apply (contr (total_happly_inv (total_happly center)))^.
Defined.

Definition total {A P Q} (f : ∀ (x:A), P x → Q x) := fun w ⇒ (w.1; f w.1 w.2).

Lemma total_happly_is {A B f} : (@total_happly A B f) = total (@apD10 A B f).
Proof.
  unfold total_happly.
  apply funext'; intro.
  destruct x.
  reflexivity.
Qed.

Definition fx_inv {A P Q} {f : ∀ x:A, P x → Q x} {k : IsEquiv (total f)}
           (x : A) (y : Q x) : P x.
  destruct k.
  change x with (x; y).1.
  apply (transport _ (base_path (eisretr (x; y)))).
  apply (equiv_inv (x; y)).2.
Defined.

Lemma Theorem477 (A : Type) (P Q : A → Type) (f : ∀ x:A, P x → Q x) :
  IsEquiv (total f) → ∀ x:A, IsEquiv (f x).
Proof.
  intros.
  refine (isequiv_adjointify (f x) (fx_inv x) _ _); unfold fx_inv; intro y.
```

```
  - destruct X.
    rewrite ap_transport. simpl. unfold base_path.
    apply (fiber_path (eisretr (x; y))).
  - destruct X. unfold base_path.
    change (x; f x y) with ((total f) (x; y)).
    rewrite eisadj. rewrite ← ap_compose.
    assert ((ap (pr1 o total f) (eissect (x; y))) = (base_path (eissect (x; y)))).
    unfold compose. unfold base_path. reflexivity.
    rewrite X. unfold base_path. simpl.
    transitivity (x; y).2.
    apply (fiber_path (eissect (x; y))).
    reflexivity.
Defined.
```

```
Theorem ex2_16 {A B} (f g : ∀ (x:A), B x) : IsEquiv(@apD10 A B f g).
Proof.
  apply Theorem477.
  rewrite ← total_happly_is.
  apply total_equivalence.
Qed.
```

```
End Exercise2_16.
```

Exercise 2.17 (p. 105)

(i) Show that if $A \simeq A'$ and $B \simeq B'$, then $(A \times B) \simeq (A' \times B')$.

(ii) Give two proofs of this fact, one using univalence and one not using it, and show that the two proofs are equal.

(iii) Formulate and prove analogous results for the other type formers: Σ, $\to$, Π, and $+$.

Solution (i) Suppose that $g : A \simeq A'$ and $h : B \simeq B'$. By the univalence axiom, this means that $A = A'$ and $B = B'$. But then $A \times B = A' \times B'$, so again by univalence $(A \times B) \simeq (A' \times B')$.

```
Theorem ex2_17_i '{Univalence}: ∀ (A A' B B' : Type),
  A ≃ A' → B ≃ B' → (A × B) ≃ (A' × B').
Proof.
  intros A A' B B' f g.
  apply equiv_path_universe in f.
  apply equiv_path_universe in g.
  apply equiv_path_universe.
  apply (transport (fun x:Type ⇒ A × B = A' × x) g).
  apply (transport (fun x:Type ⇒ A × B = x × B) f).
  reflexivity.
Defined.
```

(ii) To prove this without univalence, we construct an explicit equivalence. Suppose that $f : A \to A'$ and $g : B \to B'$ are both equivalences, and define $h : A \times B \to A' \times B'$ by

$$h(a, b) :\equiv (f(a), g(b))$$

with the appropriate inverse

$$h^{-1}(a', b') :\equiv (f^{-1}(a'), g^{-1}(b'))$$

Clearly these are quasi-inverses, since

$$h(h^{-1}(a', b')) \equiv h(f^{-1}(a'), g^{-1}(b')) \equiv (f(f^{-1}(a')), g(g^{-1}(b'))) \equiv (a', b')$$

and vice versa.

```
Theorem ex2_17_i' : ∀ (A A' B B' : Type),
  A ≃ A' → B ≃ B' → (A × B) ≃ (A' × B').
Proof.
  intros A A' B B' f g.
  refine (equiv_adjointify (fun z ⇒ (f (fst z), g (snd z)))
                           (fun z ⇒ (f⁻¹ (fst z), g⁻¹ (snd z)))
                           _ _);
    intro z; destruct z; apply path_prod; simpl;
    try (apply eisretr); try (apply eissect).
Defined.
```

To prove that the proofs are equivalent, it suffices to show that the underlying functions are equal, by Lemma 3.5.1.

```
Theorem equal_proofs '{Univalence} : ex2_17_i = ex2_17_i'.
Proof.
  unfold ex2_17_i, ex2_17_i'. simpl. unfold compose.
  apply path_forall; intro A. apply path_forall; intro A'.
  apply path_forall; intro B. apply path_forall; intro B'.
  apply path_forall; intro f. apply path_forall; intro g.
  (* equiv_fun *)
  assert (transport idmap
                (transport (fun x : Type ⇒ A × B = A' × x)
                   (path_universe_uncurried g)
                   (transport (fun x : Type ⇒ A × B = x × B)
                      (path_universe_uncurried f) 1))
          =
          fun z : A × B ⇒ (f (fst z), g (snd z))) as H1.
  apply path_forall; intro z.
  repeat (rewrite trans_paths; hott_simpl).
  rewrite transport_pp.
  rewrite ← transport_idmap_ap.
  rewrite ← (transport_idmap_ap Type (fun a:Type ⇒ a × B) A A' (path_universe_uncurried f) z).
  rewrite (@transport_prod Type idmap (fun x:Type ⇒ B)).
  rewrite transport_prod. simpl.
  destruct z; apply path_prod; simpl.
    rewrite transport_const.
    assert ((path_universe_uncurried f) = (path_universe (equiv_fun f))).
    unfold path_universe. destruct f. reflexivity.
    rewrite X. apply transport_path_universe.
    rewrite transport_const.
    assert ((path_universe_uncurried g) = (path_universe (equiv_fun g))).
    unfold path_universe. destruct g. reflexivity.
    rewrite X. apply transport_path_universe.
 unfold equiv_path, equiv_adjointify.
 (* Lemma 3.5.1 *)
 apply path_equiv. apply H1.
Qed.
```

(iii) The proofs of the rest of these are pretty much routine. With univalence, we can just convert everything to equality, rewrite, and then convert back to equivalences. However, since Coq's rewriting approach can be fiddly, we sometimes have to write things out explicitly. Most of the conceptual work in this problem is just stating the generalizations, though, which are as follows:

Σ If $f : A \simeq A'$ and for all $x : A$ we have $B(x) \simeq B'(f(x))$, then $(\sum_{(x:A)} B(x)) \simeq (\sum_{(x':A')} B'(x'))$. Another way to state the second assumption is that there is a fiberwise equivalence $g : \prod_{(x:A)} B(x) \simeq B'(f(x))$.

$\to$ If $A \simeq A'$ and $B \simeq B'$, then $(A \to B) \simeq (A' \to B')$.

Π If $f : A \simeq A'$ and there is a fiberwise equivalence $g : \prod_{(x:A)} B(x) \simeq B'(f(x))$, then

$$\left(\prod_{x:A} B(x)\right) \simeq \left(\prod_{x':A'} B'(f(x'))\right)$$

$+$ If $A \simeq A'$ and $B \simeq B'$, then $A + B \simeq A' + B'$.

```
Definition sigma_f `{Univalence} {A A' : Type} {B : A → Type} {B' : A' → Type}
        (f : A ≃ A') (g : ∀ x : A, B x ≃ B' (f x)) :
  {x : A & B x} → {x' : A' & B' x'}.
Proof.
  intros. ∃ (f X.1). apply (g X.1 X.2).
Defined.

Definition sigma_f_inv `{Univalence} {A A' : Type} {B : A → Type} {B' : A' → Type}
      (f : A ≃ A') (g : ∀ x : A, B x ≃ B' (f x)) (X : {x' : A' & B' x'})
      :=
      (f^-1 X.1; (g (f^-1 X.1))^-1 ((eisretr f X.1)^ # X.2)).

Theorem ex2_17_sigma `{Univalence} (A A' : Type) (B : A → Type) (B' : A' → Type)
        (f : A ≃ A') (g : ∀ x : A, B x ≃ B' (f x)) :
  {x : A & B x} ≃ {x' : A' & B' x'}.
Proof.
  intros.
  refine (equiv_adjointify (sigma_f f g) (sigma_f_inv f g) _ _); intro h;
  unfold sigma_f, sigma_f_inv; simpl; apply path_sigma_uncurried; simpl.
  ∃ (eisretr f h.1). simpl.
  rewrite (eisretr (g (f^-1 h.1))). rewrite transport_pV. reflexivity.
  ∃ (eissect f h.1).
  refine ((ap_transport (eissect f h.1) (fun x' ⇒ (g x') ^-1)
                          (transport B' (eisretr f (f h.1)) ^ (g h.1 h.2)))^ @ _).
  rewrite transport_compose, eisadj, transport_pV. apply eissect.
Defined.

Theorem ex2_17_maps `{Univalence} : ∀ (A A' B B' : Type),
  A ≃ A' → B ≃ B' → (A → B) ≃ (A' → B').
Proof.
  intros A A' B B' HA HB.
  apply equiv_path_universe in HA.
  apply equiv_path_universe in HB.
  apply equiv_path_universe.
  rewrite HA, HB. reflexivity.
Defined.

Definition pi_f {A A' : Type} {B : A → Type} {B' : A' → Type}
        (f : A ≃ A') (g : ∀ x : A, B x ≃ B' (f x)) :
  (∀ x:A, B x) → (∀ x':A', B' x').
```

```
  intros.
  apply ((eisretr f x') # ((g (f^-1 x')) (X (f^-1 x')))).
Defined.

Definition pi_f_inv {A A' : Type} {B : A → Type} {B' : A' → Type}
           (f : A ≃ A') (g : ∀ x : A, B x ≃ B' (f x)) :
  (∀ x':A', B' x') → (∀ x:A, B x).
  intros.
  apply (g x)^-1. apply (X (f x)).
Defined.

Theorem ex2_17_pi {A A' : Type} {B : A → Type} {B' : A' → Type}
           (f : A ≃ A') (g : ∀ x : A, B x ≃ B' (f x)) :
  (∀ x:A, B x) ≃ (∀ x':A', B' x').
Proof.
  refine (equiv_adjointify (pi_f f g) (pi_f_inv f g) _ _); intro h;
  unfold pi_f, pi_f_inv.
  apply path_forall; intro x'.
  rewrite (eisretr (g (f^-1 x'))). induction (eisretr f x'). reflexivity.
  apply path_forall; intro x.
  apply (ap (g x))^-1. rewrite (eisretr (g x)).
  rewrite eisadj. rewrite ← transport_compose.
  induction (eissect f x). reflexivity.
Qed.

Theorem ex2_17_sum `{Univalence} : ∀ (A A' B B' : Type),
  A ≃ A' → B ≃ B' → (A + B) ≃ (A' + B').
Proof.
  intros A A' B B' HA HB.
  apply equiv_path_universe in HA.
  apply equiv_path_universe in HB.
  apply equiv_path_universe.
  rewrite HA, HB. reflexivity.
Qed.
```

3 Sets and logic

```
Notation Brck Q := (minus1Trunc Q).
```

Exercise 3.1 (p. 127) Prove that if $A \simeq B$ and A is a set, then so is B.

Solution Suppose that $A \simeq B$ and that A is a set. Since A is a set, $x =_A y$ is a mere proposition. And since $A \simeq B$, this means that $x =_B y$ is a mere proposition, hence that B is a set.

Alternatively, we can unravel some definitions. By assumption we have $f : A \simeq B$ and

$$g : \mathsf{isSet}(A) \equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q)$$

Now suppose that $x, y : B$ and $p, q : x = y$. Then $f^{-1}(x), f^{-1}(y) : A$ and $f^{-1}(p), f^{-1}(q) : f^{-1}(x) = f^{-1}(y)$, so

$$f\Big(g(f^{-1}(x), f^{-1}(y), f^{-1}(p), f^{-1}(q))\Big) : f(f^{-1}(p)) = f(f^{-1}(q))$$

Since f^{-1} is a quasi-inverse of f, we have the homotopy $\alpha : \prod_{(a:A)}(f(f^{-1}(a)) = a)$, thus

$$\alpha_x^{-1} \cdot f\Big(g(f^{-1}(x), f^{-1}(y), f^{-1}(p), f^{-1}(q))\Big) \cdot \alpha_y : p = q$$

So we've constructed an element of

$$\mathsf{isSet}(B) : \prod_{(x,y:B)} \prod_{(p,q:x=y)} (p = q)$$

```
Theorem ex3_1 (A B : Type) `{Univalence} : A ≃ B → IsHSet A → IsHSet B.
Proof.
  intros f g.
  apply equiv_path_universe in f.
  rewrite ← f.
  apply g.
Defined.
```

```
Theorem ex3_1' (A B : Type) : A ≃ B → IsHSet A → IsHSet B.
Proof.
  intros f g x y.
  apply hprop_allpath. intros p q.
  assert (ap f^-1 p = ap f^-1 q). apply g.
  apply ((ap (ap f^-1))^-1 X).
Defined.
```

Exercise 3.2 (p. 127) Prove that if A and B are sets, then so is $A + B$.

Solution Suppose that A and B are sets. Then for all $a, a' : A$ and $b, b' : B$, $a = a'$ and $b = b'$ are contractible. Given the characterization of the path space of $A + B$ in \S2.12, it must also be contractible. Hence $A + B$ is a set.

More explicitly, suppose that $z, z' : A + B$ and $p, q : z = z'$. By induction, there are four cases.

- $z \equiv \mathsf{inl}(a)$ and $z' \equiv \mathsf{inl}(a')$. Then $(z = z') \simeq (a = a')$, and since A is a set, $a = a'$ is contractible, so $(z = z')$ is as well.
- $z \equiv \mathsf{inl}(a)$ and $z' \equiv \mathsf{inr}(b)$. Then $(z = z') \simeq \mathbf{0}$, so p is a contradiction.
- $z \equiv \mathsf{inr}(b)$ and $z' \equiv \mathsf{inl}(a)$. Then $(z = z') \simeq \mathbf{0}$, so p is a contradiction.
- $z \equiv \mathsf{inr}(b)$ and $z' \equiv \mathsf{inr}(b')$. Then $(z = z') \simeq (b = b')$, and since B is a set, this type is contractible.

So $z = z'$ is contractible, making $A + B$ a set.

```
Theorem ex3_2 (A B : Type) : IsHSet A → IsHSet B → IsHSet (A + B).
Proof.
  intros f g.
  intros z z'. apply hprop_allpath. intros p q.
  assert ((path_sum z z')^-1 p = (path_sum z z')^-1 q).
  pose proof ((path_sum z z')^-1 p).
  destruct z as [a | b], z' as [a' | b'].
  apply f. contradiction. contradiction. apply g.
  apply ((ap (path_sum z z')^-1)^-1 X).
Defined.
```

Exercise 3.3 (p. 127) Prove that if A is a set and $B : A \to \mathcal{U}$ is a type family such that $B(x)$ is a set for all $x : A$, then $\sum_{(x:A)} B(x)$ is a set.

Solution At this point the pattern in these proofs is relatively obvious: show that the path space of the combined types is determined by the path spaces of the base types, and then apply the fact that the base types are sets. So here we suppose that $ww' : \sum_{(x:A)} B(x)$, and that $pq : (w = w')$. Now

$$(w = w') \simeq \sum_{p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w')} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')$$

by Theorem 2.7.2. Since A is a set, $\mathsf{pr}_1(w) = \mathsf{pr}_1(w')$ is contractible, so $(w = w') \simeq ((\mathsf{refl}_{\mathsf{pr}_1(w)})_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w')) \equiv (\mathsf{pr}_2(w) = \mathsf{pr}_2(w'))$ by Lemma 3.11.9. And since B is a set, this too is contractible, making $w = w'$ contractible and $\sum_{(x:A)} B(x)$ a set.

```
Theorem ex3_3 (A : Type) (B : A → Type) :
  IsHSet A → (∀ x:A, IsHSet (B x)) → IsHSet {x : A & B x}.
Proof.
  intros f g.
  intros w w'. apply hprop_allpath. intros p q.
  assert ((path_sigma_uncurried B w w')^-1 p = (path_sigma_uncurried B w w')^-1 q).
  apply path_sigma_uncurried. simpl.
  assert (p..1 = q..1). apply f. ∃ X. apply (g w'.1).
  apply ((ap (path_sigma_uncurried B w w')^-1)^-1 X).
Defined.
```

Exercise 3.4 (p. 127) Show that A is a mere proposition if and only if $A \to A$ is contractible.

Solution For the forward direction, suppose that A is a mere proposition. Then by Example 3.6.2, $A \to A$ is a mere proposition. We also have $\mathsf{id}_A : A \to A$ when A is inhabited and $! : A \to A$ when it's not, so $A \to A$ is contractible.

For the other direction, suppose that $A \to A$ is contractible and that $xy : A$. We have the functions $z \mapsto x$ and $z \mapsto y$, and since $A \to A$ is contractible these functions are equal. happly then gives $x = y$, so A is a mere proposition.

```
Theorem ex3_4 (A : Type) : IsHProp A ↔ Contr (A → A).
Proof.
  split; intro H.
  (* forward *)
  ∃ idmap; intro f.
  apply path_forall; intro x. apply H.
  (* backward *)
  apply hprop_allpath; intros x y.
  assert ((fun z:A ⇒ x) = (fun z:A ⇒ y)).
  destruct H. transitivity center.
  apply (contr (fun _ ⇒ x))^. apply (contr (fun _ : A ⇒ y)).
  apply (apD10 X x).
Defined.
```

Exercise 3.5 (p. 127) Show that $\mathsf{isProp}(A) \simeq (A \to \mathsf{isContr}(A))$.

Solution Lemma 3.3.3 gives us maps $\mathsf{isProp}(A) \to (A \to \mathsf{isContr}(A))$ and $(A \to \mathsf{isContr}(A)) \to \mathsf{isProp}(A)$. Note that $\mathsf{isContr}(A)$ is a mere proposition, so $A \to \mathsf{isContr}(A)$ is as well. $\mathsf{isProp}(A)$ is always a mere proposition, so by Lemma 3.3.3 we have the equivalence.

```
Theorem ex3_5 (A : Type) : IsHProp A ≃ (A → Contr A).
Proof.
```

```
  (*  Lemma 3.3.3 *)
  apply equiv_iff_hprop.
  (*  Lemma 3.11.3 *)
  apply contr_inhabited_hprop.
  apply hprop_inhabited_contr.
Qed.
```

Exercise 3.6 (p. 127) Show that if A is a mere proposition, then so is $A + (\neg A)$.

Solution Suppose that A is a mere proposition, and that $x, y : A + (\neg A)$. By a case analysis, we have

- $x = \mathsf{inl}(a)$ and $y = \mathsf{inl}(a')$. Then $(x = y) \simeq (a = a')$, and A is a mere proposition, so this holds.
- $x = \mathsf{inl}(a)$ and $y = \mathsf{inr}(f)$. Then $f(a) : \mathbf{0}$, a contradiction.
- $x = \mathsf{inr}(f)$ and $y = \mathsf{inl}(a)$. Then $f(a) : \mathbf{0}$, a contradiction.
- $x = \mathsf{inr}(f)$ and $y = \mathsf{inr}(f')$. Then $(x = y) \simeq (f = f')$, and $\neg A$ is a mere proposition, so this holds.

```
Theorem ex3_6 {A} : IsHProp A → IsHProp (A + ¬A).
Proof.
  intro H.
  assert (IsHProp (¬A)) as H'.
  apply hprop_allpath. intros f f'. apply path_forall; intro x. contradiction.
  apply hprop_allpath. intros x y.
  destruct x as [a | f], y as [a' | f'].
  apply (ap inl). apply H.
  contradiction.
  contradiction.
  apply (ap inr). apply H'.
Defined.
```

Exercise 3.7 (p. 127) More generally, show that if A and B are mere propositions and $\neg(A \times B)$, then $A + B$ is also a mere proposition.

Solution Suppose that A and B are mere propositions with $f : \neg(A \times B)$, and let $x, y : A + B$. Then we have cases:

- $x = \mathsf{inl}(a)$ and $y = \mathsf{inl}(a')$. Then $(x = y) \simeq (a = a')$, and A is a mere proposition, so this holds.
- $x = \mathsf{inl}(a)$ and $y = \mathsf{inr}(b)$. Then $f(a, b) : \mathbf{0}$, a contradiction.
- $x = \mathsf{inr}(b)$ and $y = \mathsf{inl}(a)$. Then $f(a, b) : \mathbf{0}$, a contradiction.
- $x = \mathsf{inr}(b)$ and $y = \mathsf{inr}(b')$. Then $(x = y) \simeq (b = b')$, and B is a mere proposition, so this holds.

```
Theorem ex3_7 {A B} : IsHProp A → IsHProp B → ~(A × B) → IsHProp (A+B).
Proof.
  intros HA HB f.
  apply hprop_allpath; intros x y.
  destruct x as [a | b], y as [a' | b'].
  apply (ap inl). apply HA.
  assert Empty. apply (f (a, b')). contradiction.
  assert Empty. apply (f (a', b)). contradiction.
  apply (ap inr). apply HB.
Defined.
```

Exercise 3.8 (p. 127) Assuming that some type $\mathsf{isequiv}(f)$ satisfies

(i) For each $f : A \to B$, there is a function $\mathsf{qinv}(f) \to \mathsf{isequiv}(f)$;
(ii) For each f we have $\mathsf{isequiv}(f) \to \mathsf{qinv}(f)$;
(iii) For any two $e_1, e_2 : \mathsf{isequiv}(f)$ we have $e_1 = e_2$,

show that the type $\|\mathsf{qinv}(f)\|$ satisfies the same conditions and is equivalent to $\mathsf{isequiv}(f)$.

Solution Suppose that $f : A \to B$. There is a function $\mathsf{qinv}(f) \to \|\mathsf{qinv}(f)\|$ by definition. Since $\mathsf{isequiv}(f)$ is a mere proposition (by iii), the recursion principle for $\|\mathsf{qinv}(f)\|$ gives a map $\|\mathsf{qinv}(f)\| \to \mathsf{isequiv}(f)$, which we compose with the map from (ii) to give a map $\|\mathsf{qinv}(f)\| \to \mathsf{qinv}(f)$. Finally, $\|\mathsf{qinv}(f)\|$ is a mere proposition by construction. Since $\|\mathsf{qinv}(f)\|$ and $\mathsf{isequiv}(f)$ are both mere propositions and logically equivalent, $\|\mathsf{qinv}(f)\| \simeq \mathsf{isequiv}(f)$ by Lemma 3.3.3.

```
Section Exercise3_8.

Variables (E Q : Type).
Hypothesis H1 : Q → E.
Hypothesis H2 : E → Q.
Hypothesis H3 : ∀ e e' : E, e = e'.

Definition ex3_8_i : Q → (Brck Q) := min1.

Definition ex3_8_ii : (Brck Q) → Q.
  intro q. apply H2. apply (@minus1Trunc_rect_nondep Q E).
  apply H1. apply H3. apply q.
Defined.

Theorem ex3_8_iii : ∀ q q' : Brck Q, q = q'.
  apply allpath_hprop.
Defined.

Theorem ex3_8_iv : (Brck Q) ≃ E.
  apply @equiv_iff_hprop.
  apply hprop_allpath. apply ex3_8_iii.
  apply hprop_allpath. apply H3.
  apply (H1 o ex3_8_ii).
  apply (ex3_8_i o H2).
Defined.

End Exercise3_8.
```

Exercise 3.9 (p. 127) Show that if LEM holds, then the type $\mathsf{Prop} :\equiv \sum_{(A:\mathcal{U})} \mathsf{isProp}(A)$ is equivalent to $\mathbf{2}$.

Solution Suppose that

$$f : \prod_{A:\mathcal{U}} (\mathsf{isProp}(A) \to (A + \neg A))$$

To construct a map $\mathsf{Prop} \to \mathbf{2}$, it suffices to consider an element of the form (A, g), where $g : \mathsf{isProp}(A)$. Then $f(g) : A + \neg A$, so we have two cases:

- $f(g) \equiv \mathsf{inl}(a)$, in which case we send it to $1_{\mathbf{2}}$, or
- $f(g) \equiv \mathsf{inr}(a)$, in which case we send it to $0_{\mathbf{2}}$.

To go the other way, note that LEM splits Prop into two equivalence classes (basically, the true and false propositions), and $\mathbf{1}$ and $\mathbf{0}$ are in different classes. Univalence quotients out these classes, leaving us with two elements. We'll use $\mathbf{1}$ and $\mathbf{0}$ as representatives, so we send $0_{\mathbf{2}}$ to $\mathbf{0}$ and $1_{\mathbf{2}}$ to $\mathbf{1}$.

Coq has some trouble with the universes here, so we have to specify that we want (Unit : Type) and (Empty : Type); otherwise we get the Type0 versions.

```
Section Exercise3_9.

Hypothesis LEM : ∀ (A : Type), IsHProp A → (A + ¬A).

Definition ex3_9_f (P : {A:Type & IsHProp A}) : Bool :=
  match (LEM P.1 P.2) with
    | inl a ⇒ true
    | inr a' ⇒ false
  end.

Lemma hprop_Unit : IsHProp (Unit : Type).
  apply hprop_inhabited_contr. intro u. apply contr_unit.
Defined.

Definition ex3_9_inv (b : Bool) : {A : Type & IsHProp A} :=
  match b with
    | true ⇒ @existT Type IsHProp (Unit : Type) hprop_Unit
    | false ⇒ @existT Type IsHProp (Empty : Type) hprop_Empty
  end.

Theorem ex3_9 `{Univalence} : {A : Type & IsHProp A} ≃ Bool.
Proof.
  refine (equiv_adjointify ex3_9_f ex3_9_inv _ _).
  intro b. unfold ex3_9_f, ex3_9_inv.
  destruct b.
    simpl. destruct (LEM (Unit:Type) hprop_Unit).
      reflexivity.
      contradiction n. exact tt.
    simpl. destruct (LEM (Empty:Type) hprop_Empty).
      contradiction. reflexivity.
  intro w. destruct w as [A p]. unfold ex3_9_f, ex3_9_inv.
    simpl. destruct (LEM A p) as [x | x].
    apply path_sigma_uncurried. simpl.
    assert ((Unit:Type) = A).
      assert (Contr A). apply contr_inhabited_hprop. apply p. apply x.
      apply equiv_path_universe. apply equiv_inverse. apply equiv_contr_unit.
    ∃ X. induction X. simpl.
    assert (IsHProp (IsHProp (Unit:Type))). apply HProp_HProp. apply X.
    apply path_sigma_uncurried. simpl.
    assert ((Empty:Type) = A).
      apply equiv_path_universe. apply equiv_iff_hprop.
        intro z. contradiction.
        intro a. contradiction.
    ∃ X. induction X. simpl.
    assert (IsHProp (IsHProp (Empty:Type))). apply HProp_HProp. apply X.
Qed.

End Exercise3_9.
```

Exercise 3.10 (p. 127) Show that if $\mathcal{U}_{i+1}$ satisfies LEM, then the canonical inclusion $\mathsf{Prop}_{\mathcal{U}_i} \to \mathsf{Prop}_{\mathcal{U}_{i+1}}$ is an equivalence.

Solution If LEM_{i+1} holds, then LEM_i holds as well. For suppose that $A : \mathcal{U}_i$ and $p : \mathsf{isProp}(A)$. Then we also have $A : \mathcal{U}_{i+1}$, so $\mathsf{LEM}_{i+1}(A, p) : A + \neg A$, establishing LEM_i. By the previous exercise, then, $\mathsf{Prop}_{\mathcal{U}_i} \simeq \mathbf{2} \simeq \mathsf{Prop}_{\mathcal{U}_{i+1}}$.

Since Coq doesn't let the user access the `Type`$_i$ hierarchy, there's not much to do here. This is really more of a "proof by contemplation" anyway.

Exercise 3.11 (p. 127) Show that it is not the case that for all $A : \mathcal{U}$ we have $\|A\| \to A$.

Solution We can essentially just copy Theorem 3.2.2. Suppose given a function $f : \prod_{(A:\mathcal{U})} \|A\| \to A$, and recall the equivalence $e : \mathbf{2} \simeq \mathbf{2}$ from Exercise 2.13 given by $e(1_\mathbf{2}) :\equiv 0_\mathbf{2}$ and $e(0_\mathbf{2}) = 1_\mathbf{2}$. Then $\mathsf{ua}(e) : \mathbf{2} = \mathbf{2}$, $f(\mathbf{2}) : \|\mathbf{2}\| \to \mathbf{2}$, and

$$\mathsf{apd}_f(\mathsf{ua}(e)) : \mathsf{transport}^{A \mapsto (\|A\| \to A)}(\mathsf{ua}(e), f(\mathbf{2})) = f(\mathbf{2})$$

So for $u : \|\mathbf{2}\|$,

$$\mathsf{happly}(\mathsf{apd}_f(\mathsf{ua}(e)), u) : \mathsf{transport}^{A \mapsto (\|A\| \to A)}(\mathsf{ua}(e), f(\mathbf{2}))(u) = f(\mathbf{2})(u)$$

and by 2.9.4, we have

$$\mathsf{transport}^{A \mapsto (\|A\| \to A)}(\mathsf{ua}(e), f(\mathbf{2}))(u) = \mathsf{transport}^{A \mapsto A}(\mathsf{ua}(e), f(\mathbf{2})(\mathsf{transport}^{|-|}(\mathsf{ua}(e)^{-1}, u)))$$

But, any two $u, v : \|A\|$ are equal, since $\|A\|$ is contractible. So $\mathsf{transport}^{|-|}(\mathsf{ua}(e)^{-1}, u) = u$, and so

$$\mathsf{happly}(\mathsf{apd}_f(\mathsf{ua}(e)), u) : \mathsf{transport}^{A \mapsto A}(\mathsf{ua}(e), f(\mathbf{2})(u)) = f(\mathbf{2})(u)$$

and the propositional computation rule for ua gives

$$\mathsf{happly}(\mathsf{apd}_f(\mathsf{ua}(e)), u) : e(f(\mathbf{2})(u)) = f(\mathbf{2})(u)$$

But e has no fixed points, so we have a contradiction.

```
Lemma negb_no_fixpoint : ∀ b, ¬ (negb b = b).
Proof.
  intros b H. destruct b; simpl in H.
    apply (false_ne_true H).
    apply (true_ne_false H).
Defined.

Theorem ex3_11 `{Univalence} : ¬ (∀ A, Brck A → A).
Proof.
  intro f.
  assert (∀ b, negb (f Bool b) = f Bool b). intro b.
  assert (transport (fun A ⇒ Brck A → A) (path_universe negb) (f Bool) b
          =
          f Bool b).
  apply (apD10 (apD f (path_universe negb)) b).
  assert (transport (fun A ⇒ Brck A → A) (path_universe negb) (f Bool) b
          =
          transport idmap (path_universe negb)
                    (f Bool (transport (fun A ⇒ Brck A)
                                       (path_universe negb)^
                                       b))).
  apply (@transport_arrow Type0 (fun A ⇒ Brck A) idmap).
  rewrite X in X0.
```

```
  assert (b = (transport (fun A : Type ⇒ Brck A) (path_universe negb) ^ b)).
  apply allpath_hprop. rewrite ← X1 in X0. symmetry in X0.
  assert (transport idmap (path_universe negb) (f Bool b) = negb (f Bool b)).
  apply transport_path_universe. rewrite X2 in X0. apply X0.
  apply (@negb_no_fixpoint (f Bool (min1 true))).
  apply (X (min1 true)).
Qed.
```

Exercise 3.12 (p. 127) Show that if LEM holds, then for all $A : \mathcal{U}$ we have $\big\| \|A\| \to A \big\|$.

Solution Suppose that LEM holds, and that $A : \mathcal{U}$. By LEM, either $\|A\|$ or $\neg\|A\|$. If the former, then we can use the recursion principle for $\|A\|$ to construct a map to $\big\| \|A\| \to A \big\|$, then apply it to the element of $\|A\|$. So we need a map $A \to \big\| \|A\| \to A \big\|$, which is not hard to get:

$$\lambda(a : A).\ \big|\lambda(a' : \|A\|).\ a\big| : A \to \big\| \|A\| \to A \big\|$$

If the latter, then we have the canonical map out of the empty type $\|A\| \to A$, hence we have $\big\| \|A\| \to A \big\|$.

```
Section Exercise3_12.

Hypothesis LEM : ∀ A, IsHProp A → (A + ¬A).

Theorem ex3_12 : ∀ A, Brck (Brck A → A).
Proof.
  intro A.
  destruct (LEM (Brck A) minus1Trunc_is_prop).
  apply (minus1Trunc_rect_nondep (fun a ⇒ min1 (fun _ : Brck A ⇒ a))).
  apply minus1Trunc_is_prop. apply m.
  apply min1. intro a. contradiction n.
Defined.

End Exercise3_12.
```

Exercise 3.13 (p. 127) Show that the axiom

$$\mathsf{LEM}' : \prod_{A:\mathcal{U}} (A + \neg A)$$

implies that for $X : \mathcal{U}$, $A : X \to \mathcal{U}$, and $P : \prod_{(x:X)} A(x) \to \mathcal{U}$, if X is a set, $A(x)$ is a set for all $x : X$, and $P(x,a)$ is a mere proposition for all $x : X$ and $a : A(x)$,

$$\left(\prod_{x:X} \left\| \sum_{a:A(x)} P(x,a) \right\|\right) \to \left\| \sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x, g(x)) \right\|.$$

Solution By Lemma 3.8.2, it suffices to show that for any set X and any $Y : X \to \mathcal{U}$ such that $Y(x)$ is a set, we have

$$\left(\prod_{x:X} \|Y(x)\|\right) \to \left\| \prod_{x:X} Y(x) \right\|$$

Suppose that $f : \prod_{(x:X)} \|Y(x)\|$. By LEM', either $Y(x)$ is inhabited or it's not. If it is, then $\mathsf{LEM}'(Y(x)) \equiv y : Y(x)$, and we have

$$|\lambda(x : X).\ y| : \left\| \prod_{x:X} Y(x) \right\|$$

Suppose instead that $\neg Y(x)$ and that $x : X$. Then $f(x) : \|Y(x)\|$. Since we're trying to derive a mere proposition, we can ignore this truncation and suppose that $f(x) : Y(x)$, in which case we have a contradiction, and we're done.

The reason we can ignore the truncation (and apply *strip_truncations* in Coq) in hypotheses is given by the reasoning in the previous Exercise. If the conclusion is a mere proposition, then the recursion principle for $\|Y(x)\|$ allows us to construct an arrow out of $\|Y(x)\|$ if we have one from $Y(x)$.

```
Definition AC := ∀ X A P,
  IsHSet X → (∀ x, IsHSet (A x)) → (∀ x a, IsHProp (P x a))
  → ((∀ x:X, Brck {a:A x & P x a})
       → Brck {g : ∀ x, A x & ∀ x, P x (g x)}).

Definition AC_simpl := ∀ (X : hSet) (Y : X → Type),
  (∀ x, IsHSet (Y x)) →
  ((∀ x, Brck (Y x)) → Brck (∀ x, Y x)).

Lemma hprop_is_hset (A : Type) : IsHProp A → IsHSet A.
Proof.
  typeclasses eauto.
Defined.

Lemma Lemma382 : AC ≃ AC_simpl.
Proof.
  apply equiv_iff_hprop; unfold AC, AC_simpl.

  intros AC. intros X Y HY D.
  assert (Brck ({g : ∀ x, Y x & ∀ x, (fun x a ⇒ Unit) x (g x)})
                  ≃ Brck (∀ x, Y x)).
  apply equiv_iff_hprop.
  intro w. strip_truncations. apply min1. apply w.1.
  intro g. strip_truncations. apply min1. ∃ g. intro x. apply tt.
  apply X0. apply (AC X Y (fun x a ⇒ Unit)). apply X. apply HY.
  intros. apply hprop_Unit. intros.
  assert (Brck (Y x)) as y by apply D. strip_truncations.
  apply min1. ∃ y. apply tt.

  intros AC_simpl X A P HX HA HP D.
  assert (Brck (∀ x, {a : A x & P x a})
               ≃ Brck {g : ∀ x, A x & ∀ x, P x (g x)}).
  apply equiv_iff_hprop.
  intros. strip_truncations. apply min1. ∃ (fun x ⇒ (X0 x).1).
  intro x. apply (X0 x).2.
  intros. strip_truncations. apply min1. intro x. apply (X0.1 x; X0.2 x).
  apply X0. apply (AC_simpl (default_HSet X HX) (fun x ⇒ {a : A x & P x a})).
  intros. apply ex3_3. apply (HA x). intro a.
  apply hprop_is_hset. apply (HP x a).
  intro x. apply (D x).
Defined.

Section Exercise3_13.

Hypothesis LEM' : ∀ A, A + ¬A.

Theorem ex3_13: AC.
Proof.
  apply Lemma382. unfold AC_simpl. intros X Y HX HY.
  apply min1. intros.
  destruct (LEM' (Y x)). apply y.
```

```
  assert (Brck (Y x)) as y'. apply HY.
  assert (¬ Brck (Y x)) as nn. intro p. strip_truncations. contradiction.
  contradiction nn.
Defined.

End Exercise3_13.
```

Exercise 3.14 (p. 127) Show that assuming LEM, the double negation $\neg\neg A$ has the same universal property as the propositional truncation $\|A\|$, and is therefore equivalent to it.

Solution Suppose that $a : \neg\neg A$ and that we have some function $g : A \to B$, where B is a mere proposition, so $p : \mathsf{isProp}(B)$. We can construct a function $\neg\neg A \to \neg\neg B$ by using contraposition twice, producing $g'' : \neg\neg A \to \neg\neg B$

$$g''(h) :\equiv \lambda(f : \neg B).\, h(\lambda(a : A).\, f(g(a)))$$

LEM then allows us to use double negation elimination to produce a map $\neg\neg B \to B$. Suppose that $f : \neg\neg B$. Then we have $\mathsf{LEM}(B, p) : B + \neg B$, and in the left case we can produce the witness, and in the right case we use f to derive a contradiction. Explicitly, we have $\ell : \neg\neg B \to B$ given by

$$\ell(f) :\equiv \mathsf{rec}_{B+\neg\neg B}(B, \mathsf{id}_B, f, \mathsf{LEM}(B, p))$$

The computation rule does not hold judgementally for $g'' \circ \ell$. I don't see that it can, given the use of LEM. Clearly it does hold propositionally, if one takes $|a|' :\equiv \lambda f.\, f(a)$ to be the analogue of the constructor for $\|A\|$; for any $a : A$, we have $g(a) : B$, and the fact that B is a mere proposition ensures that $(g'' \circ \ell)(|a|') = g(a)$.

```
Section Exercise3_14.

Hypothesis LEM : ∀ A, IsHProp A → (A + ¬A).

Definition Brck' (A : Type) := ¬ ¬ A.
Definition min1' {A : Type} (a : A) : Brck' A := fun f ⇒ f a.

Definition contrapositive {A B : Type} : (A → B) → (¬ B → ¬ A).
  intros. intro a. apply X0. apply X. apply a.
Defined.

Definition DNE {B : Type} `{IsHProp B} : ¬ ¬ B → B.
  intros. destruct (LEM B IsHProp0). apply b. contradiction X.
Defined.

Definition trunc_rect' {A B : Type} (g : A → B) : IsHProp B → Brck' A → B.
  intros HB a. apply DNE. apply (contrapositive (contrapositive g)). apply a.
Defined.

End Exercise3_14.
```

Exercise 3.15 (p. 128) Show that if we assume propositional resizing, then the type

$$\prod_{P:\mathsf{Prop}} ((A \to P) \to P)$$

has the same universal property as $\|A\|$.

Solution Let $A : \mathcal{U}_i$, so that for $\|A\|'' :\equiv \prod_{(P:\mathsf{Prop}_{\mathcal{U}_i})}((A \to P) \to P)$ we have $\|A\|'' : \mathcal{U}_{i+1}$. By propositional resizing, however, we have a corresponding $\|A\|'' : \mathcal{U}_i$. To construct an arrow $\|A\|'' \to B$, suppose that $f : \|A\|''$ and $g : A \to B$. Then $f(B,g) : B$. So $\lambda f.\, \tilde{f}(B,g) : \|A\|'' \to B$, where $\tilde{f}$ is the image of f under the inverse of the canonical inclusion $\mathsf{Prop}_{\mathcal{U}_i} \to \mathsf{Prop}_{\mathcal{U}_{i+1}}$.

To show that the computation rule holds, let

$$|a|'' :\equiv \lambda P.\, \lambda f.\, f(a) : \prod_{P:\mathsf{Prop}} ((A \to P) \to P)$$

We need to show that $(\lambda f.\, \tilde{f}(B,g))(|a|'') \equiv g(a)$. Assuming that propositional resizing gives a judgemental equality, we have

$$\begin{aligned}(\lambda f.\, \tilde{f}(B,g))(|a|'') &\equiv (\lambda f.\, \tilde{f}(B,g))(\lambda P.\, \lambda f.\, f(a)) \\ &\equiv (\lambda P.\, \lambda f.\, f(a))(B,g) \\ &\equiv g(a)\end{aligned}$$

```
Definition Brck'' (A : Type) := ∀ (P : hProp), ((A → P) → P).
Definition min1'' {A : Type} (a : A) := fun (P : hProp) (f : A → P) ⇒ f a.
Definition trunc_rect'' {A B : Type} (g : A → B) : IsHProp B → Brck'' A → B.
  intros p f.
  apply (f (hp B p)). apply g.
Defined.
```

Exercise 3.16 (p. 128) Assuming LEM, show that double negation commutes with universal quantification of mere propositions over sets. That is, show that if X is a set and each $Y(x)$ is a mere proposition, then LEM implies

$$\left(\prod_{x:X} \neg\neg Y(x)\right) \simeq \left(\neg\neg \prod_{x:X} Y(x)\right).$$

Solution Each side is a mere proposition, since one side is a dependent function into a mere proposition and the other is a negation. So we just need to show that each implies the other. From left to right we use the fact that LEM is equivalent to double negation to obtain $\prod_{(x:X)} Y(x)$, and double negation introduction is always allowed, giving the right side. For the other direction we do the same.

```
Section Exercise3_16.
Hypothesis LEM : ∀ A, IsHProp A → (A + ¬ A).
Theorem ex3_16 (X : hSet) (Y : X → Type) :
  (∀ x, IsHProp (Y x)) →
  (∀ x, ¬ ¬ Y x) ≃ ¬ ¬ (∀ x, Y x).
Proof.
  intro HY. apply equiv_iff_hprop; intro H.
  intro f. apply f. intro x.
  destruct (LEM (Y x)).
    apply HY. apply y.
    contradiction (H x).
  intro x.
  destruct (LEM (Y x)).
    apply HY. intro f. contradiction.
    assert (¬ (∀ x, Y x)). intro f. contradiction (f x).
    contradiction H.
Qed.
End Exercise3_16.
```

Exercise 3.17 (p. 128) Show that the rules for the propositional truncation given in §3.7 are sufficient to imply the following induction principle: for any type family $B : \|A\| \to \mathcal{U}$ such that each $B(x)$ is a mere proposition, if for every $a : A$ we have $B(|a|)$, then for every $x : \|A\|$ we have $B(x)$.

Solution Suppose that $B : \|A\| \to \mathcal{U}$, $B(x)$ is a mere proposition for all $x : \|A\|$ and that $f : \prod_{(a:A)} B(|a|)$. Suppose that $x : \|A\|$; we need to construct an element of $B(x)$. By the induction principle for $\|A\|$, it suffices to exhibit a map $A \to B(x)$. So suppose that $a : A$, and we'll construct an element of $B(x)$. Since $\|A\|$ is contractible, we have $p : |a| = x$, and $p_*(f(a)) : B(x)$.

```
Theorem ex3_17 (A : Type) (B : Brck A → Type) :
  (∀ x, IsHProp (B x)) → (∀ a, B (min1 a)) → (∀ x, B x).
Proof.
  intros H B f. intro x.
  apply (@minus1Trunc_rect_nondep A (B x)).
  intro a. assert (min1 a = x) as p. apply allpath_hprop.
  apply (transport _ p). apply (f a).
  apply allpath_hprop. apply x.
Defined.
```

Exercise 3.18 (p. 128) Show that the law of excluded middle

$$\mathsf{LEM} : \prod_{A:\mathcal{U}} (\mathsf{isProp}(A) \to (A + \neg A))$$

and the law of double negation

$$\mathsf{DN} : \prod_{A:\mathcal{U}} (\mathsf{isProp}(A) \to (\neg\neg A \to A))$$

are logically equivalent.

Solution For the forward direction, suppose that LEM holds, that $A : \mathcal{U}$, that $H : \mathsf{isProp}(A)$, and that $f : \neg\neg A$. We then need to produce an element of A. We have $z :\equiv \mathsf{LEM}(A, H) : A + \neg A$, so we can consider cases:

- $z \equiv \mathsf{inl}(a)$, in which case we can produce a.
- $z \equiv \mathsf{inr}(x)$, in which case we have $f(x) : \mathbf{0}$, a contradiction.

giving the forward direction.

Suppose instead that DN holds, and we have $A : \mathcal{U}$ and $H : \mathsf{isProp}(A)$. We need to provide an element of $A + \neg A$. By Exercise 3.6, $A + \neg A$ is a mere proposition, so by DN, if we can give an element of $\neg\neg(A + \neg A)$, then we'll get one of $A + \neg A$. In Exercise 1.13 we constructed such an element, so producing that gives one of $A + \neg A$, and we're done.

```
Theorem ex3_18:
  (∀ A, IsHProp A → (A + ¬A)) ↔ (∀ A, IsHProp A → (¬ ¬A → A)).
Proof.
  split.
  intros LEM A H f. destruct (LEM A H). apply a. contradiction.
  intros DN A H. apply (DN (A + ¬A) (ex3_6 H)).
  exact (fun g : ¬ (A + ¬ A) ⇒ g (inr (fun a:A ⇒ g (inl a)))).
Qed.
```

*Exercise 3.19 (p. 128)** Suppose $P : \mathbb{N} \to \mathcal{U}$ is a decidable family of mere propositions. Prove that

$$\left\| \sum_{n:\mathbb{N}} P(n) \right\| \to \sum_{n:\mathbb{N}} P(n).$$

Solution Since $P : \mathbb{N} \to \mathcal{U}$ is decidable, we have $f : \prod_{(n:\mathbb{N})}(P(n) + \neg P(n))$. So if $\left\|\sum_{(n:\mathbb{N})} P(n)\right\|$ is inhabited, then there is some smallest n such that $P(n)$. It would be nice if we could define a function to return the smallest n such that $P(n)$. But unbounded minimization isn't a total function, so that won't obviously work. Following the discussion of Corollary 3.9.2, what we can do instead is to define some

$$Q : \left(\sum_{n:\mathbb{N}} P(n)\right) \to \mathcal{U}$$

such that $\sum_{(w:\sum_{(n:\mathbb{N})} P(n))} Q(w)$ is a mere proposition. Then we can project out an element of $\sum_{(n:\mathbb{N})} P(n)$.

$Q(w)$ will be the proposition that w is the smallest member of $\sum_{(n\mathbb{N})} P(n)$. Explicitly,

$$Q(w) :\equiv \prod_{w':\sum_{(n:\mathbb{N})} P(n)} \mathsf{pr}_1(w) \leq \mathsf{pr}_1(w')$$

Then we have

$$\sum_{w:\sum_{(n:\mathbb{N})} P(n)} Q(w) \equiv \sum_{(w:\sum_{(n:\mathbb{N})} P(n))} \prod_{(w':\sum_{(n:\mathbb{N})} P(n))} \mathsf{pr}_1(w) \leq \mathsf{pr}_1(w')$$

which we must show to be a mere proposition. Suppose that w and w' are two elements of this type. By $\mathsf{pr}_2(w)$ and $\mathsf{pr}_2(w')$, we have $\mathsf{pr}_1(w) \leq \mathsf{pr}_1(w')$ and $\mathsf{pr}_1(w') \leq \mathsf{pr}_1(w)$, so $\mathsf{pr}_1(w) = \mathsf{pr}_1(w')$. Since $\mathbb{N}$ has decidable equality, $\mathsf{pr}_1(w) \leq \mathsf{pr}_2(w')$ is a mere proposition for all w and w', meaning that $Q(w)$ is a mere proposition. So $w = w'$, meaning that our type is contractible.

Now we can use the universal property of $\left\|\sum_{(n:\mathbb{N})} P(n)\right\|$ to construct an arrow into $\sum_{(w:\sum_{(n:\mathbb{N})} P(n))} Q(w)$ by way of a function $\left(\sum_{(n:\mathbb{N})} P(n)\right) \to \sum_{(w:\sum_{(n:\mathbb{N})} P(n))} Q(w)$. So suppose that we have some element $w : \sum_{(n:\mathbb{N})} P(n)$. Using bounded minimization, we can obtain the smallest element of $\sum_{(n:\mathbb{N})} P(n)$ that's less than or equal to w, and this will in fact be the smallest element *tout court*. This means that it's a member of our constructed type, so we've constructed a map

$$\left\| \sum_{n:\mathbb{N}} P(n) \right\| \to \sum_{w:\sum_{(n:\mathbb{N})} P(n)} Q(w)$$

and projecting out gives the function in the statement.

I'm having just the damnedest time trying to work everything out in Coq. At some point I'll sort out my loadpath to cut out the nat lemmas. I'm sure I'm overcomplicating the correctness proofs for *bounded_min*, though. No way can they be this long.

```
Local Open Scope nat_scope.

Fixpoint nat_code (n m : nat) :=
  match n, m with
    | O, O ⇒ Unit
    | S n', O ⇒ Empty
    | O, S m' ⇒ Empty
    | S n', S m' ⇒ nat_code n' m'
  end.

Fixpoint nat_r (n : nat) : nat_code n n :=
  match n with
```

```
    | O ⇒ tt
    | S n′ ⇒ nat_r n′
  end.

Definition nat_encode (n m : nat) (p : n = m) : (nat_code n m)
  := transport (nat_code n) p (nat_r n).

Definition nat_decode : ∀ (n m : nat), (nat_code n m) → (n = m).
Proof.
  induction n, m; intro.
  reflexivity. contradiction. contradiction.
  apply (ap S). apply IHn. apply X.
Defined.

Theorem equiv_path_nat : ∀ n m, (nat_code n m) ≃ (n = m).
Proof.
  intros.
  refine (equiv_adjointify (nat_decode n m) (nat_encode n m) _ _).

  intro p. induction p. simpl.
  induction n. reflexivity. simpl.
  apply (ap (ap S) IHn).

  generalize dependent m.
  induction n. induction m.
  intro c. apply eta_unit.
  intro c. contradiction.
  induction m.
  intro c. contradiction.
  intro c. simpl. unfold nat_encode.
  refine ((transport_compose _ S _ _)^ @ _).
  simpl. apply IHn.
Defined.

Lemma Sn_neq_O : ∀ n, S n ≠ O.
Proof.
  intros n H. apply nat_encode in H. contradiction.
Defined.

Lemma plus_cancelL : ∀ n m k, n + m = n + k → m = k.
Proof.
  intro n. induction n. trivial.
  intros m k H.
  simpl in H. apply S_inj in H. apply IHn. apply H.
Defined.

Lemma plus_assoc : ∀ n m k, n + (m + k) = (n + m) + k.
Proof.
  induction n. trivial.
  intros m k. simpl. apply (ap S).
  apply IHn.
Defined.

Lemma plus_O_r : ∀ n, n = n + O.
Proof.
  induction n. reflexivity.
  simpl. apply (ap S). apply IHn.
Defined.
```

```
Lemma le_antisymmetric (n m : nat) : (n ≤ m) → (m ≤ n) → (n = m).
Proof.
  intros p q. destruct p as [k p], q as [k' q].
  transparent assert (Hm : (m + k' + k = m)).
  apply ((ap (fun l ⇒ l + k) q) @ p).
  transparent assert (Hk : (k' + k = 0)).
  apply (plus_cancelL m).
  apply ((plus_assoc _ _ _) @ Hm @ (plus_O_r m)).
  refine (q^ @ _). refine ((ap (fun l ⇒ l + k') p)^ @ _).
Admitted.

Definition decidable {A} (P : A → Type) := ∀ a, (P a + ¬ P a).

Module Exercise3_19.

Variable (P : nat → Type).
Hypothesis (H : decidable P).
Hypothesis (HP : ∀ n, IsHProp (P n)).

Definition Q (w : {n : nat & P n}) :=
  ∀ w' : {n : nat & P n}, w.1 ≤ w'.1.

Lemma hprop_sigmaQ : IsHProp ({w : {n : nat & P n} & Q w}).
Proof.
  apply hprop_allpath. intros w w'.
  transparent assert (H : (w.1 = w'.1)).
Admitted.

Theorem ex3_19 : Brck {n : nat & P n} → {n : nat & P n}.
Proof.
  intros. apply (@pr1 {n : nat & P n} Q).
Admitted.

End Exercise3_19.
```

Exercise 3.20 (p. 128) Prove Lemma 3.11.9(ii): if A is contractible with center a, then $\sum_{(x:A)} P(x)$ is equivalent to $P(a)$.

Solution Suppose that A is contractible with center a. For the forward direction, suppose that $w : \sum_{(x:A)} P(x)$. Then $\mathsf{pr}_1(w) = a$, since A is contractible, so from $\mathsf{pr}_2(w) : P(\mathsf{pr}_1(w))$ and the indiscernibility of identicals, we have $P(a)$. For the backward direction, suppose that $p : P(a)$. Then we have $(a, p) : \sum_{(x:A)} P(x)$.

To show that these are quasi-inverses, suppose that $p : P(a)$. Going backward gives $(a, p) : \sum_{(x:A)} P(x)$, and going forward we have $(\mathsf{contr}_a^{-1})_* p$. Since A is contractible, $\mathsf{contr}_a = \mathsf{refl}_a$, so this reduces to p, as needed. For the other direction, suppose that $w : \sum_{(x:X)} P(x)$. Going forward gives $(\mathsf{contr}^{-1}_{\mathsf{pr}_1(w)})_* \mathsf{pr}_2(w) : P(a)$, and going back gives

$$(a, (\mathsf{contr}^{-1}_{\mathsf{pr}_1(w)})_* \mathsf{pr}_2(w)) : \sum_{x:A} P(x)$$

By Theoremm 2.7.2, it suffices to show that $a = \mathsf{pr}_1(w)$ and that

$$(\mathsf{contr}_{\mathsf{pr}_1(w)})_* (\mathsf{contr}^{-1}_{\mathsf{pr}_1(w)})_* \mathsf{pr}_2(w) = \mathsf{pr}_2(w)$$

The first of these is given by the fact that A is contractible. The second results from the functorality of transport.

```
Definition ex3_20_f (A : Type) (P : A → Type) (HA : Contr A) :
  {x : A & P x} → P (center A).
  intros. apply (transport _ (contr X.1)^). apply X.2.
Defined.

Definition ex3_20_g (A : Type) (P : A → Type) (HA : Contr A) :
  P (center A) → {x : A & P x}.
  intros. apply (center A; X).
Defined.

Theorem ex3_20 (A : Type) (P : A → Type) (HA : Contr A) :
  {x : A & P x} ≃ P (center A).
Proof.
  refine (equiv_adjointify (ex3_20_f A P HA) (ex3_20_g A P HA) _ _);
  unfold ex3_20_f, ex3_20_g.

  intro p. simpl.
  assert (Contr (center A = center A)). apply contr_paths_contr.
  assert (contr (center A) = idpath). apply allpath_hprop.
  rewrite X0. reflexivity.

  intro w. apply path_sigma_uncurried.
  simpl. ∃ (contr w.1).
  apply transport_pV.
Defined.
```

Exercise 3.21 (p. 128) Prove that $\mathsf{isProp}(P) \simeq (P \simeq \|P\|)$.

Solution $\mathsf{isProp}(P)$ is a mere proposition by Lemma 3.3.5. $P \simeq \|P\|$ is also a mere proposition. An equivalence is determined by its underlying function, and for all $f, g : P \to \|P\|$, $f = g$ by function extensionality and the fact that $\|P\|$ is a mere proposition. Since each of the two sides is a mere proposition, we just need to show that they imply each other, by Lemma 3.3.3. Lemma 3.9.1 gives the forward direction. For the backward direction, suppose that $e : P \simeq \|P\|$, and let $x, y : P$. Then $e(x) = e(y)$, since $\|P\|$ is a proposition, and applying e^{-1} to each side gives $x = y$. Thus P is a mere proposition.

```
Theorem ex3_31 (P : Type) : IsHProp P ≃ (P ≃ Brck P).
Proof.
  assert (IsHProp (P ≃ Brck P)). apply hprop_allpath; intros e1 e2.
  apply path_equiv. apply path_forall; intro p.
  apply hprop_allpath. apply allpath_hprop.
  apply equiv_iff_hprop.

  intro HP. apply equiv_iff_hprop. apply min1.
  apply (minus1Trunc_rect_nondep idmap). apply HP.

  intro e. apply hprop_allpath; intros x y.
  assert (e x = e y) as p. apply hprop_allpath. apply allpath_hprop.
  rewrite (eissect e x)^. rewrite (eissect e y)^.
  apply (ap e^-1 p).
Defined.
```

***Exercise 3.22 (p. 128)** As in classical set theory, the finite version of the axiom of choice is a theorem. Prove that the axiom of choice holds when X is a finite type $\mathsf{Fin}(n)$.

Solution We want to show that for all n, $A : \mathsf{Fin}(n) \to \mathcal{U}$, and $P : \prod_{(m_n:\mathsf{Fin}(n))} A(m_n) \to \mathcal{U}$, if A is a family of sets and P a family of propositions, then

$$\left(\prod_{m_n:\mathsf{Fin}(n)} \left\| \sum_{a:A(m_n)} P(m_n, a) \right\| \right) \to \left\| \sum_{(g:\prod_{(m_n:\mathsf{Fin}(n))} A(m_n))} \prod_{(m_n:\mathsf{Fin}(n))} P(m_n, g(m_n)) \right\|.$$

We proceed by induction. For the base case, suppose that $n \equiv 0$, so we are interested in $\mathsf{Fin}(0) :\equiv \sum_{(n:\mathbb{N})}(m < 0)$, which is equivalent to $\mathbf{0}$. Then we have $\mathsf{ind}_{\mathbf{0}}(A) : \prod_{(m_0:\mathsf{Fin}(0))} A(m_0)$, so

$$(\mathsf{ind}_{\mathbf{0}}(A), \mathsf{ind}_{\mathbf{0}}(\lambda m_0.\, P(m_0, \mathsf{ind}_{\mathbf{0}}(A, m_0))))$$

is an element of the codomain.

For the induction step, suppose that we have an element

$$f : \prod_{m_{n+1}:\mathsf{Fin}(n+1)} \left\| \sum_{a:A(m_{n+1})} P(m_{n+1}, a) \right\|$$

which can be modified in the obvious way to give a function

$$\tilde{f} : \prod_{m_n:\mathsf{Fin}(n)} \left\| \sum_{a:A(m_n)} P(m_n, a) \right\|$$

So by the induction step, and since the element we're trying to construct is a mere proposition, we have an element

$$w : \sum_{(g:\prod_{(m_n:\mathsf{Fin}(n))} A(m_n))} \prod_{(m_n:\mathsf{Fin}(n))} P(m_n, g(m_n))$$

Now, we need to construct an element of

$$\left\| \sum_{(g:\prod_{(m_{n+1}:\mathsf{Fin}(n+1))} A(m_{n+1}))} \prod_{(m_{n+1}:\mathsf{Fin}(n+1))} P(m_{n+1}, g(m_{n+1})) \right\|$$

To construct the first slot, suppose that $k : \mathsf{Fin}(n+1)$. Then because we have $e : \mathsf{Fin}(n+1) \simeq \mathsf{Fin}(n) + \mathbf{1}$, there are two cases: either $e(k) : \mathsf{Fin}(n)$ or $e(k) = *$. In the first case, we set $g(e(k)) :\equiv (\mathsf{pr}_1 w)(e(k))$. In the second, we

Suppose the first. Then we can modify f in the obvious way to obtain

$$\tilde{f} : \prod_{m_n:\mathsf{Fin}(n)} \left\| \sum_{a:A(m_n)} P(m_n, a) \right\|$$

So by the induction step, and since the element we're trying to construct is a mere proposition, we have an element

$$w : \sum_{(g:\prod_{(m_n:\mathsf{Fin}(n))} A(m_n))} \prod_{(m_n:\mathsf{Fin}(n))} P(m_n, g(m_n))$$

```
Infix "≠" := (fun n m ⇒ ¬ (n = m)) : nat_scope.

Definition pred (n : nat) :=
  match n with
   | O ⇒ O
```

```
    | S n' ⇒ n'
  end.
Lemma S_pred_inv : ∀ n, (n ≠ O) → S (pred n) = n.
Proof.
  induction n. intros. contradiction X. reflexivity.
  intros. reflexivity.
Defined.
Lemma nat_eq_decidable : ∀ (n m : nat), (n = m) + ¬ (n = m).
Proof.
  induction n, m.
  left. reflexivity.
  right. intro H. apply nat_encode in H. contradiction.
  right. intro H. apply nat_encode in H. contradiction.
  destruct (IHn m).
    left. apply (ap S p).
    right. intro H. apply S_inj in H. contradiction.
Defined.
Definition cardF_f {n} : Fin (S n) → (Fin n) + Unit.
  intro x. destruct x as [m [k p]].
  destruct (nat_eq_decidable m n).
  right. apply tt.
  left. ∃ m. ∃ (pred k).
  rewrite S_pred_inv.
  simpl in p. rewrite ← plus_n_Sm in p. apply S_inj in p. apply p.
  intro. rewrite H in p. rewrite ← plus_1_r in p. apply S_inj in p.
  contradiction.
Defined.
Definition Fin_incl {n : nat} : Fin n → Fin (S n).
  intros m. destruct m as [m [k p]].
  ∃ m. ∃ (S k). apply ((plus_n_Sm m (S k))^ @ (ap S p)).
Defined.
Definition cardF_g {n} : (Fin n) + Unit → Fin (S n).
  intro x. destruct x as [m | t]. apply (Fin_incl m).
  ∃ n. ∃ 0. apply (plus_1_r n)^.
Defined.
Lemma sum_O_r (n m : nat) : n + m = n → m = 0.
Proof.
  induction n. simpl. apply idmap.
  intros. simpl in H. apply S_inj in H. apply IHn. apply H.
Defined.
Lemma cardFO : Fin O ≃ Empty.
Proof.
  refine (equiv_adjointify _ _ _ _).
  intro n. destruct n as [n [k p]].
  assert (S (n + k) = 0). transitivity (n + S k). apply plus_n_Sm. apply p.
  apply equiv_path_nat in X. contradiction.
  intro e. contradiction.
  intro e. contradiction.
  intro n. destruct n as [n [k p]].
  assert (S (n + k) = 0). transitivity (n + S k). apply plus_n_Sm. apply p.
```

```
  apply equiv_path_nat in X. contradiction.
Defined.

Lemma cardF {n : nat} : Fin (S n) ≃ (Fin n) + Unit.
Admitted.

Theorem ex3_22 `{Univalence}: ∀ (n : nat) (A : Fin n → Type)
                                 (P : ∀ (m : Fin n), A m → Type),
  (∀ n, IsHSet (A n)) → (∀ m a, IsHProp (P m a)) →
  (∀ m, Brck {a : A m & P m a}) →
  Brck {g : ∀ m, A m & ∀ k, P k (g k)}.
Proof.
  induction n.
  (*  case n = 0 *)
  intros A P HA HP f. apply min1.
  assert (∀ m, A m). intros. contradiction (cardFO m).
  ∃ X. intro. contradiction (cardFO k).
  (*  case n = S n *)
  intros A P HA HP f.
  assert (∀ k : Fin n, Brck {a : A (Fin_incl k) & P (Fin_incl k) a}) as w.
  intros. apply (f (Fin_incl k)).
  apply IHn in w.
  strip_truncations.
  assert (∀ m : Fin (S n), A m) as g.
  intro m.
  rewrite ← (eissect cardF m).
  destruct (cardF m) as [em | m_m]; simpl.
Admitted.

Local Close Scope nat_scope.
```

4 Equivalences

***Exercise 4.1 (p. 147)** Consider the type of "two-sided adjoint equivalence data" for $f : A \to B$,

$$\sum_{(g:B\to A)} \; \sum_{(\eta:g\circ f\sim \mathsf{id}_A)} \; \sum_{\epsilon:f\circ g\sim \mathsf{id}_B} \left(\prod_{x:A} f(\eta x) = \epsilon(fx)\right) \times \left(\prod_{y:B} g(\epsilon y) = \eta(gy)\right)$$

By Lemma 4.2.2, we know that if f is an equivalence, then this type is inhabited. Give a characterization of this type analogous to Lemma 4.1.1. Give an example showing that this type is not generally a mere proposition.

***Exercise 4.2 (p. 147)** Show that for any $A, B : \mathcal{U}$, the following type is equivalent to $A \simeq B$.

$$\sum_{R:A\to B\to \mathcal{U}} \left(\prod_{a:A} \mathsf{isContr}\left(\sum_{b:B} R(a,b)\right)\right) \times \left(\prod_{b:B} \mathsf{isContr}\left(\sum_{a:A} R(a,b)\right)\right).$$

Extract from this a definition of a type satisfying the three desiderata of $\mathsf{isequiv}(f)$.

***Exercise 4.3 (p. 147)** Reformulate the proof of Lemma 4.1.1 without using univalence.

```
Definition qinv {A B : Type} (f : A → B) :=
  {g : B → A & (f o g ∼ idmap) × (g o f ∼ idmap)}.
```

```
Axiom qinv_isequiv : ∀ A B (f : A → B), qinv f → IsEquiv f.
Axiom isequiv_qinv : ∀ A B (f : A → B), IsEquiv f → qinv f.

Definition ex4_3_f {A B : Type} {f : A → B} : qinv f → ∀ (x:A), x = x.
  intros.
  destruct X as [g [alpha beta]].
  etransitivity (g (f x)).
  apply (beta x)^. apply (beta x).
Defined.

Theorem Theorem411 {A B : Type} (f : A → B) : (qinv f) →
  (qinv f) ≃ (∀ x:A, x = x).
Proof.
Admitted.
```

*__Exercise 4.4 (p. 147)__ Suppose $f : A \to B$ and $g : B \to C$ and $b : B$.

(i) Show that there is a natural map $\mathsf{fib}_{g \circ f}(g(b)) \to \mathsf{fib}_g(g(b))$ whose fiber over $(b, \mathsf{refl}_{g(b)})$ is equivalent to $\mathsf{fib}_f(b)$.

(ii) Show that $\mathsf{fib}_{g \circ f}(g(b)) \simeq \sum_{(w:\mathsf{fib}_g(g(b)))} \mathsf{fib}_f(\mathsf{pr}_1 w)$.

Solution (i) Unfolding the fib notation, we are looking for a map

$$\left(\sum_{a:A} (g(f(a)) = g(b))\right) \to \left(\sum_{b':B} (g(b') = g(b))\right)$$

The obvious choice is $f^* :\equiv (a, p) \mapsto (f(a), p)$. We then must show that $\mathsf{fib}_{f^*}(b, \mathsf{refl}_{g(b)}) \simeq \mathsf{fib}_f(b)$. Unfolding the notation again, we're looking for an equivalence

$$\left(\sum_{w:\mathsf{fib}_{g \circ f}(g(b))} (f^*(w) = (b, \mathsf{refl}_{g(b)}))\right) \simeq \left(\sum_{a:A} (f(a) = b)\right)$$

For the arrow, suppose that (w, p) is an element of the domain, so that $w : \mathsf{fib}_{g \circ f}(g(b))$ and $q : f^*(w) = (b, \mathsf{refl}_{g(b)})$. By the induction principle for $\mathsf{fib}_{g \circ f}(g(b))$, it suffices to consider the case where $w \equiv (a, p)$, for $a : A$ and $p : g(f(a)) = g(b)$. Then

$$q : (f^*(a, p) = (b, \mathsf{refl}_{g(b)})) \equiv ((f(a), p) = (b, \mathsf{refl}_{g(b)}))$$

thus $(a, \mathsf{pr}_1 q) : \mathsf{fib}_f(b)$. Explicitly, our map is

$$z \mapsto (\mathsf{pr}_1(\mathsf{pr}_1 z), \mathsf{pr}_1(\mathsf{pr}_2 z))$$

For a quasi-inverse, suppose that $(a, p) : \mathsf{fib}_f(b)$. Then $(a, g(p)) : \mathsf{fib}_{g \circ f}(g(b))$. We need a proof that

$$(f^*(a, g(p)) = (b, \mathsf{refl}_{g(b)})) \equiv ((f(a), g(p)) = (b, \mathsf{refl}_{g(b)}))$$

p provides the proof of equality for the first slots. For the second, by induction we can consider the case where $f(a) \equiv b$ and $p \equiv \mathsf{refl}_b$. Then $g(p) \equiv \mathsf{refl}_{g(b)}$, and the proof we seek is just reflexivity.

```
Section Exercise4_4.

Variables (A B C D : Type) (f : A → B) (g : B → C) (b : B).

Definition f_star (z : ((hfiber (g o f) (g b)))) : (hfiber g (g b)) :=
  (f z.1; z.2).

Definition ex4_4_f (z : (hfiber f_star (b; 1))) : (hfiber f b) :=
```

```
  (z.1.1; (base_path z.2)).
Definition ex4_4_g (w : (hfiber f b)) : (hfiber f_star (b; 1)).
  refine ((w.1; ap g w.2); _).
  unfold f_star. simpl.
  apply path_sigma_uncurried. ∃ w.2. simpl.
  induction w.2. reflexivity.
Defined.
Lemma ex4_4_alpha : Sect ex4_4_g ex4_4_f.
Proof.
  unfold ex4_4_f, ex4_4_g.
  intro w. destruct w as [a p]. simpl.
  apply path_sigma_uncurried; simpl.
  ∃ 1. simpl. unfold f_star in *.
  induction p. reflexivity.
Defined.
Lemma ex4_4_beta : Sect ex4_4_f ex4_4_g.
Proof.
  unfold ex4_4_f, ex4_4_g, f_star. intro w.
  apply path_sigma_uncurried. simpl.
  assert ((w.1.1; ap g (base_path w.2)) = w.1).
  unfold hfiber in w.
  apply (@path_sigma A (fun x:A ⇒ (g ∘ f) x = g b)
                    (w.1.1; ap g (base_path w.2))
                    w.1
                    1).
  simpl.
  (*
  apply (@hfiber_triangle B C g (g b) (b; 1) (f w.1.1; w.1.2) w.2^).

  apply path_sigma_uncurried. exists 1. simpl.
  destruct w as [a p] q. simpl in *.
  transitivity ((ap g (base_path q))^)^. symmetry. apply inv_V.
  transitivity (ap g (base_path q)^)^. hott_simpl.
  transitivity (ap g (base_path q^))^. unfold base_path. hott_simpl.
  apply moveR_V1.
  apply symmetry.
  apply (@hfiber_triangle B C g (g b) (b; 1) (f a; p) q^).
  exists X.
  destruct w as [a p] q. simpl in *.
  *)
Admitted.
Theorem ex4_4 : (hfiber (f_star) (b; 1)) ≃ (hfiber f b).
Proof.
  apply (equiv_adjointify ex4_4_f ex4_4_g ex4_4_alpha ex4_4_beta).
Defined.
End Exercise4_4.
```

Exercise 4.5 (p. 147) Prove that equivalences satisfy the *2-out-of-6 property*: given $f : A \to B$ and $g : B \to C$ and $h : C \to D$, if $g \circ f$ and $h \circ g$ are equivalences, so are f, g, h, and $h \circ g \circ f$. Use this to give a higher-level proof of Theorem 2.11.1.

Solution Suppose that $g \circ f$ and $h \circ g$ are equivalences.

- f is an equivalence with quasi-inverse $(g \circ f)^{-1} \circ g$. It's a retract because

$$\begin{aligned} f \circ (g \circ f)^{-1} \circ g &\sim (h \circ g)^{-1} \circ (h \circ g) \circ f \circ (g \circ f)^{-1} \circ g \\ &\sim (h \circ g)^{-1} \circ h \circ g \\ &\sim \mathsf{id}_B \end{aligned}$$

  and a section because $(g \circ f)^{-1} \circ g \circ f \sim \mathsf{id}_A$.
- g is an equivalence with quasi-inverse $(h \circ g)^{-1} \circ h$. First we have

$$\begin{aligned} g \circ (h \circ g)^{-1} \circ h &\sim g \circ (h \circ g)^{-1} \circ h \circ g \circ f \circ (g \circ f)^{-1} \\ &\sim g \circ f \circ (g \circ f)^{-1} \\ &\sim \mathsf{id}_C \end{aligned}$$

  and second $(h \circ g)^{-1} \circ h \circ g \sim \mathsf{id}_B$.
- h is an equivalence with quasi-inverse $g \circ (h \circ g)^{-1}$. First, $h \circ g \circ (h \circ g)^{-1} \sim \mathsf{id}_D$, and we have $g \circ (h \circ g)^{-1} \circ h \sim \mathsf{id}_C$ by the previous part.
- $h \circ g \circ f$ is an equivalence with quasi-inverse $f^{-1} \circ (h \circ g)^{-1}$. Both directions are immediate:

$$h \circ g \circ f \circ f^{-1} \circ (h \circ g)^{-1} \sim \mathsf{id}_D$$
$$f^{-1} \circ (h \circ g)^{-1} \circ h \circ g \circ f \sim \mathsf{id}_A$$

Now we must give a higher-level proof that if $f : A \to B$ is an equivalence, then for all $a, a' : A$ so is ap_f. This uses the following somewhat obvious fact, which I don't recall seeing in the text or proving yet: if $f : A \to B$ is an equivalence and $f \sim g$, then g is an equivalence. For any $a : A$ we have $f^{-1}(g(a)) = f^{-1}(f(a)) = a$ and for any $b : B$, $g(f^{-1}(b)) = f(f^{-1}(b)) = b$, giving $\mathsf{isequiv}(g)$.

Consider the sequence

$$(a = a') \xrightarrow{\mathsf{ap}_f} (f(a) = f(a')) \xrightarrow{\mathsf{ap}_{f^{-1}}} \left(f^{-1}(f(a)) = f^{-1}(f(a'))\right) \xrightarrow{\mathsf{ap}_f} \left(f(f^{-1}(f(a))) = f(f^{-1}(f(a')))\right)$$

Since f is an equivalence, we have

$$\alpha : \prod_{b:B} f(f^{-1}(b)) = b \qquad \beta : \prod_{a:A} f^{-1}(f(a)) = a$$

For all $p : a = a'$, $\mathsf{ap}_{f^{-1}}(\mathsf{ap}_f(p)) = \beta_a \cdot p \cdot \beta_{a'}^{-1}$, which follows from the functorality of ap and the naturality of homotopies (Lemmas 2.2.2 and 2.4.3). In other words, the composition of the first two arrows is homotopic to concatenating with β on either side, which is obviously an equivalence. Similarly, the composition of the second two arrows is homotopic to concatenating with the appropriate α on either side, again an obvious equivalence. So by the 2-out-of-6 property, the first arrow is an equivalence, which was to be proved.

```
Theorem two_out_of_six {A B C D : Type} (f : A → B) (g : B → C) (h : C → D) :
  IsEquiv (g o f) → IsEquiv (h o g) →
  (IsEquiv f ∧ IsEquiv g ∧ IsEquiv h ∧ IsEquiv (h o g o f)).
Proof.
  intros Hgf Hhg. split.

  (*  case f *)
  refine (isequiv_adjointify f ((g o f) ^-1 o g) _ _).
  intro b.
  change (f (((g o f) ^-1 o g) b)) with ((f o (g o f) ^-1 o g) b).
  assert ((f o (g o f) ^-1 o g) b
```

```
          =
          ((h ∘ g) ^-1 ∘ (h ∘ g) ∘ f ∘ (g ∘ f) ^-1 ∘ g) b).
change (((h ∘ g) ^-1 ∘ (h ∘ g) ∘ f ∘ (g ∘ f) ^-1 ∘ g) b)
       with ((((h ∘ g) ^-1 ((h ∘ g) ((f ∘ (g ∘ f) ^-1 ∘ g) b))))).
rewrite (eissect (h ∘ g)). reflexivity.
rewrite X.
change (((h ∘ g) ^-1 ∘ (h ∘ g) ∘ f ∘ (g ∘ f) ^-1 ∘ g) b)
       with ((((h ∘ g) ^-1 ∘ h) ((((g ∘ f) ((g ∘ f) ^-1 (g b))))))).
rewrite (eisretr (g ∘ f)).
change (((h ∘ g) ^-1 ∘ h) (g b)) with (((h ∘ g) ^-1 ∘ (h ∘ g)) b).
apply (eissect (h ∘ g)).
intro a. apply (eissect (g ∘ f)).
split.
(*  case g *)
refine (isequiv_adjointify g ((h ∘ g) ^-1 ∘ h) _ _).
intro c.
change (g (((h ∘ g) ^-1 ∘ h) c)) with ((g ∘ (h ∘ g) ^-1 ∘ h) c).
assert ((g ∘ (h ∘ g) ^-1 ∘ h) c
          =
          (g ∘ (h ∘ g) ^-1 ∘ h ∘ g ∘ f ∘ (g ∘ f) ^-1) c).
change ((g ∘ (h ∘ g) ^-1 ∘ h ∘ g ∘ f ∘ (g ∘ f) ^-1) c)
       with (((g ∘ (h ∘ g) ^-1 ∘ h) ((g ∘ f) ((g ∘ f) ^-1 c)))).
rewrite (eisretr (g ∘ f)). reflexivity.
rewrite X.
change ((g ∘ (h ∘ g) ^-1 ∘ h ∘ g ∘ f ∘ (g ∘ f) ^-1) c)
       with (g (((h ∘ g) ^-1 ((h ∘ g) ((f ∘ (g ∘ f) ^-1) c))))).
rewrite (eissect (h ∘ g)).
change (g ((f ∘ (g ∘ f) ^-1) c)) with (((g ∘ f) ∘ (g ∘ f) ^-1) c).
apply (eisretr (g ∘ f)).
intro b. apply (eissect (h ∘ g)).
split.
(*  case h *)
refine (isequiv_adjointify h (g ∘ (h ∘ g) ^-1) _ _).
intro d. apply (eisretr (h ∘ g)).
intro c.
change ((g ∘ (h ∘ g) ^-1) (h c)) with ((g ∘ (h ∘ g) ^-1 ∘ h) c).
assert ((g ∘ (h ∘ g) ^-1 ∘ h) c
          =
          (g ∘ (h ∘ g) ^-1 ∘ h ∘ g ∘ f ∘ (g ∘ f) ^-1) c).
change ((g ∘ (h ∘ g) ^-1 ∘ h ∘ g ∘ f ∘ (g ∘ f) ^-1) c)
       with (((g ∘ (h ∘ g) ^-1 ∘ h) ((g ∘ f) ((g ∘ f) ^-1 c)))).
rewrite (eisretr (g ∘ f)). reflexivity.
rewrite X.
change ((g ∘ (h ∘ g) ^-1 ∘ h ∘ g ∘ f ∘ (g ∘ f) ^-1) c)
       with (g ((h ∘ g) ^-1 ((h ∘ g) ((f ∘ (g ∘ f) ^-1) c)))).
rewrite (eissect (h ∘ g)).
change (g ((f ∘ (g ∘ f) ^-1) c)) with (((g ∘ f) ∘ (g ∘ f) ^-1) c).
apply (eisretr (g ∘ f)).
(*  case h o g o f *)
refine (isequiv_adjointify (h ∘ g ∘ f) ((g ∘ f) ^-1 ∘ g ∘ (h ∘ g) ^-1) _ _).
```

```
  intro d.
  change ((h o g o f) (((g o f) ^-1 o g o (h o g) ^-1) d))
         with (h ((g o f) ((g o f) ^-1 ((g o (h o g) ^-1) d)))).
  rewrite (eisretr (g o f)).
  apply (eisretr (h o g)).

  intro a.
  change (((g o f) ^-1 o g o (h o g) ^-1) ((h o g o f) a))
         with ((((g o f) ^-1 o g) ((h o g) ^-1 ((h o g) (f a))))).
  rewrite (eissect (h o g)). apply (eissect (g o f)).
Qed.

Theorem isequiv_homotopic' : ∀ (A B : Type) (f g : A → B),
  IsEquiv f → f ~ g → IsEquiv g.
Proof.
  intros A B f g p h.
  refine (isequiv_adjointify g f^-1 _ _).
  intros b. apply ((h (f^-1 b))^ @ (eisretr f b)).
  intros a. apply ((ap f^-1 (h a))^ @ (eissect f a)).
Defined.

Theorem Theorem2111' (A B : Type) (a a' : A) (f : A → B) (H : IsEquiv f) :
  IsEquiv (fun p : a = a' ⇒ ap f p).
Proof.
  apply (two_out_of_six (fun p : a = a' ⇒ ap f p)
                        (fun p : (f a) = (f a') ⇒ ap f^-1 p)
                        (fun p : (f^-1 (f a)) = (f^-1 (f a')) ⇒ ap f p)).
  apply (isequiv_homotopic (fun p ⇒ (eissect f a) @ p @ (eissect f a')^)).
  refine (isequiv_adjointify _
                             (fun p ⇒ (eissect f a)^ @ p @ (eissect f a'))
                             _ _);
  intro; hott_simpl.
  intro p. induction p. hott_simpl.

  apply (isequiv_homotopic (fun p ⇒ (eisretr f (f a)) @ p @ (eisretr f (f a'))^)).
  refine (isequiv_adjointify _
                        (fun p ⇒ (eisretr f (f a))^ @ p @ (eisretr f (f a')))
                        _ _);
  intro; hott_simpl.
  intro p. induction p. hott_simpl.
Defined.
```

*__Exercise 4.6 (p. 147)__ For $A, B : \mathcal{U}$, define

$$\mathsf{idtoqinv}(A, B) : (A = B) \to \sum_{f:A\to B} \mathsf{qinv}(f)$$

by path induction in the obvious way. Let qinv-univalence denote the modified form of the univalence axiom which asserts that for all $A, B : \mathcal{U}$ the function $\mathsf{idtoqinv}(A, B)$ has a quasi-inverse.

(i) Show that qinv-univalence can be used instead of univalence in the proof of function extensionality in §4.9.

(ii) Show that qinv-univalence can be used instead of univalence in the proof of Theorem 4.1.3.

(iii) Show that qinv-univalence is inconsistent. Thus, the use of a "good" version of isequiv is essential in the statement of univalence.

Solution (i) The proof of function extensionality uses univalence in the proof of Lemma 4.9.2. Assume that $\mathcal{U}$ is qinv-univalent, and that $A, B, X : \mathcal{U}$ with $e : A \simeq B$. From e we obtain $f : A \to B$ and $p : \mathsf{ishae}(f)$, and from the latter we obtain an element $q : \mathsf{qinv}(f)$. qinv-univalence says that we may write $(f, q) = \mathsf{idtoqinv}_{A,B}(r)$ for some $r : A = B$. Then by path induction, we may assume that $r \equiv \mathsf{refl}_A$, making $e = \mathsf{id}_A$, and the function $g \mapsto g \circ \mathsf{id}_A$ is clearly an equivalence $(X \to A) \simeq (X \to B)$, establishing Lemma 4.9.2. Since the rest of the section is either an application of Lemma 4.9.2 or doesn't use the univalence axiom, the proof of function extensionality goes through.

```
Section Exercise4_6.

Definition idtoqinv {A B} : (A = B) → {f : A → B & (qinv f)}.
  path_induction. ∃ idmap. ∃ idmap.
  split; intro a; reflexivity.
Defined.

Hypothesis qinv_univalence : ∀ A B, qinv (@idtoqinv A B).

Theorem ex4_6i (A B X : Type) (e : A ≃ B) : (X → A) ≃ (X → B).
Proof.
  destruct e as [f p].
  assert (qinv f) as q. ∃ f^-1. split.
    apply (eisretr f). apply (eissect f).
  assert (A = B) as r. apply (qinv_univalence A B). apply (f ; q).
  path_induction. apply equiv_idmap.
Defined.
```

(ii) Theorem 4.1.3 provides an example of types A and B and a function $f : A \to B$ such that $\mathsf{qinv}(f)$ is not a mere proposition, relying on the result of Lemma 4.1.1. Since Lemma 4.1.1 does not actually rely on univalence (cf. Exercise 4.3), we only need to worry about the use of univalence in the proof of Theorem 4.1.3. Define $X :\equiv \sum_{(A:\mathcal{U})} \| \mathbf{2} = A \|$ and $a :\equiv (\mathbf{2}, |\mathsf{refl}_{\mathbf{2}}|) : X$. Let $e : \mathbf{2} \simeq \mathbf{2}$ be the non-identity equivalence from Exercise 2.13, which gives us $\neg : \mathbf{2} \to \mathbf{2}$ and $r : \mathsf{qinv}(\neg)$. Define $q :\equiv \mathsf{idtoqinv}_{\mathbf{2},\mathbf{2}}^{-1}(\neg, r)$. Now we can run the proof as before, applying Lemma 4.1.2.

Here univalence is used only in establishing that $a = a$ is a set, by showing that it's equivalent to $(\mathbf{2} \simeq \mathbf{2})$.

```
Lemma Lemma412 (A : Type) (a : A) (q : a = a) :
  IsHSet (a = a) → (∀ x, Brck (a = x))
  → (∀ p : a = a, p @ q = q @ p)
  → {f : ∀ (x:A), x = x & f a = q}.
Proof.
  intros i g iii.
  assert (∀ (x y : A), IsHSet (x = y)).
  intros x y.
  assert (Brck (a = x)) as gx. apply (g x).
  assert (Brck (a = y)) as gy. apply (g y).
  strip_truncations.
  apply (ex3_1' (a = a)).
  refine (equiv_adjointify (fun p ⇒ gx^ @ p @ gy) (fun p ⇒ gx @ p @ gy^) _ _);
  intros p; hott_simpl.
  apply i.
  assert (∀ x, IsHProp ({r : x = x & ∀ s : a = x, r = s^ @ q @ s})).
  intro x. assert (Brck (a = x)) as p. apply (g x). strip_truncations.
  apply hprop_allpath; intros h h'; destruct h as [r h], h' as [r' h'].
  apply path_sigma_uncurried. ∃ ((h p) @ (h' p)^).
  simpl. apply path_forall; intro s.
  apply (X x x).
```

```
  assert (∀ x, {r : x = x & ∀ s : a = x, r = (s ^ @ q) @ s}).
  intro x. assert (Brck (a = x)) as p. apply (g x). strip_truncations.
  ∃ (p^ @ q @ p). intro s.
  apply (cancelR _ _ s^). hott_simpl.
  apply (cancelL p). hott_simpl.
  transitivity (q @ (p @ s^)). hott_simpl.
  symmetry. apply (iii (p @ s^)).
  ∃ (fun x ⇒ (X1 x).1).
  transitivity (1^ @ q @ 1).
  apply ((X1 a).2 1). hott_simpl.
Defined.

Definition Bool_Bool_to_a_a :
  ((Bool:Type) ≃ (Bool:Type)) →
  (((Bool:Type); min1 1):{A : Type & Brck ((Bool:Type) = A)})
  =
  (((Bool:Type); min1 1):{A : Type & Brck ((Bool:Type) = A)}).
  intros.
  apply path_sigma_hprop. simpl.
  apply (qinv_univalence Bool Bool).1.
  destruct X. ∃ equiv_fun.
  destruct equiv_isequiv. ∃ equiv_inv.
  split. apply eisretr. apply eissect.
Defined.

Definition a_a_to_Bool_Bool :
  (((Bool:Type); min1 1):{A : Type & Brck ((Bool:Type) = A)})
  =
  (((Bool:Type); min1 1):{A : Type & Brck ((Bool:Type) = A)})
  → ((Bool:Type) ≃ (Bool:Type)).
  intros. simpl. apply base_path in X. simpl in X.
  apply idtoqinv in X.
  apply (BuildEquiv Bool Bool X.1).
  apply (isequiv_adjointify X.1 X.2.1 (fst X.2.2) (snd X.2.2)).
Defined.

Theorem ex4_6ii : {A : Type & {B : Type & {f : A → B & ¬ IsHProp (qinv f)}}}.
Proof.
  set (X := {A : Type & Brck ((Bool:Type) = A)}).
  refine (X; (X; _)).
  set (a := ((Bool:Type); min1 1) : X).
  set (e := negb_isequiv). destruct e as [lnot H].
  set (r := (lnot^-1; (eisretr lnot, eissect lnot)) : qinv lnot).
  (*  Coq update broke this
  set (q := (path_sigma_hprop a a ((qinv_univalence Bool Bool).1 (lnot; r)))).
  assert {f : forall x, x = x & (f a) = q}.
  apply Lemma412.
  apply (ex3_1' ((Bool:Type) <~> (Bool:Type))).
  refine (equiv_adjointify Bool_Bool_to_a_a a_a_to_Bool_Bool _ _);
  unfold Bool_Bool_to_a_a, a_a_to_Bool_Bool.
  intro p. simpl.
   *)
Admitted.
```

```
End Exercise4_6.
```

5 Induction

Exercise 5.1 (p. 175) Derive the induction principle for the type $\mathsf{List}(A)$ of lists from its definition as an inductive type in §5.1.

Solution The induction principle constructs an element $f : \prod_{(\ell:\mathsf{List}(A))} P(\ell)$ for some family $P : \mathsf{List}(A) \to \mathcal{U}$. The constructors for $\mathsf{List}(A)$ are $\mathsf{nil} : \mathsf{List}(A)$ and $\mathsf{cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A)$, so the hypothesis for the induction principle is given by

$$d : P(\mathsf{nil}) \to \left(\prod_{(h:A)} \prod_{(t:\mathsf{List}(A))} P(t) \to P(\mathsf{cons}(h,t))\right) \to \prod_{\ell:\mathsf{List}(A)} P(\ell)$$

So, given a $p_n : P(\mathsf{nil})$ and a function $p_c : \prod_{(h:A)} \prod_{(t:\mathsf{List}(A))} P(t) \to P(\mathsf{cons}(h,t))$, we obtain a function $f : \prod_{(\ell:\mathsf{List}(A))} P(\ell)$ with the following computation rules:

$$\begin{aligned} f(\mathsf{nil}) &:\equiv p_n \\ f(\mathsf{cons}(h,t)) &:\equiv p_c(h,t,f(t)) \end{aligned}$$

In Coq we can just use the pattern-matching syntax.

```
Module Ex1.
  Section Ex1.
    Variable A : Type.
    Variable P : list A → Type.
    Hypothesis d : P nil
                      → (∀ h t, P t → P (cons h t))
                      → ∀ l, P l.
    Variable p_n : P nil.
    Variable p_c : ∀ h t, P t → P (cons h t).
    Fixpoint f (l : list A) : P l :=
      match l with
        | nil ⇒ p_n
        | cons h t ⇒ p_c h t (f t)
      end.
    End Ex1.
End Ex1.
```

Exercise 5.2 (p. 175) Construct two functions on natural numbers which satisfy the same recurrence (e_z, e_s) but are not definitionally equal.

Solution Let C be any type, with $c : C$ some element. The constant function $f' :\equiv \lambda n.\, c$ is not definitionally equal to the function defined recursively by

$$\begin{aligned} f(0) &:\equiv c \\ f(\mathsf{succ}(n)) &:\equiv f(n) \end{aligned}$$

However, they both satisfy the same recurrence; namely, $e_z :\equiv c$ and $e_s :\equiv \lambda n.\, \mathsf{id}_C$.

```
Module Ex2.
```

```
Section Ex2.
  Variables (C : Type) (c : C).
  Definition f (n : nat) := c.
  Fixpoint f' (n : nat) :=
    match n with
      | O ⇒ c
      | S n' ⇒ f' n'
    end.
  Theorem ex5_2_O : f O = f' O.
  Proof.
    reflexivity.
  Qed.
  Theorem ex5_2_S : ∀ n, f (S n) = f' (S n).
  Proof.
    intros. unfold f, f'.
    induction n. reflexivity. apply IHn.
  Qed.
End Ex2.
```


```
End Ex2.
```


Exercise 5.3 (p. 175) Construct two different recurrences (e_z, e_s) on the same type E which are both satisfied by the same function $f : \mathbb{N} \to E$.

Solution From the previous exercise we have the recurrences

$$e_z :\equiv c \qquad\qquad e_s :\equiv \lambda n.\, \mathsf{id}_C$$

which give rise to the same function as the recurrences

$$e'_z :\equiv c \qquad\qquad e'_s :\equiv \lambda n.\, \lambda x.\, c$$

Clearly $f :\equiv \lambda n.\, c$ satisfies both of these recurrences. However, suppose that $c, c' : C$ are such that $c \neq c'$. Then $\lambda n.\, \lambda x.\, \neq \lambda n.\, \mathsf{id}_C$, so $e_s \neq e'_s$, so the recurrences are not equal.

```
Module Ex3.
Section Ex3.
  Variables (C : Type) (c c' : C) (p : ¬ (c = c')).
  Definition ez := c.
  Definition es (n : nat) (x : C) := x.
  Definition ez' := c.
  Definition es' (n : nat) := fun (x : C) ⇒ c.
  Theorem f_O : Ex2.f C c O = ez.
  Proof.
    reflexivity.
  Defined.
  Theorem f_S : ∀ n, Ex2.f C c (S n) = es n (Ex2.f C c n).
  Proof.
    reflexivity.
  Defined.
  Theorem f_O' : Ex2.f C c O = ez'.
  Proof.
```

```
    reflexivity.
  Defined.
  Theorem f_S' : ∀ n, Ex2.f C c (S n) = es' n (Ex2.f C c n).
  Proof.
    reflexivity.
  Defined.
  Theorem ex5_3 : ¬ ((ez, es) = (ez', es')).
  Proof.
    intro q. apply (ap snd) in q. simpl in q. unfold es, es' in q.
    assert (idmap = fun x:C ⇒ c) as r.
    apply (apD10 q O).
    assert (c' = c) as s.
    apply (apD10 r).
    symmetry in s.
    contradiction p.
  Defined.
End Ex3.
End Ex3.
```

Exercise 5.4 (p. 175) Show that for any type family $E : \mathbf{2} \to \mathcal{U}$, the induction operator

$$\mathsf{ind}_{\mathbf{2}}(E) : (E(0_{\mathbf{2}}) \times E(1_{\mathbf{2}})) \to \prod_{b:\mathbf{2}} E(b)$$

is an equivalence.

Solution For a quasi-inverse, suppose that $f : \prod_{(b:\mathbf{2})} E(b)$. To provide an element of $E(0_{\mathbf{2}}) \times E(1_{\mathbf{2}})$, we take the pair $(f(0_{\mathbf{2}}), f(1_{\mathbf{2}}))$. For one direction around the loop, consider an element (e_0, e_1) of the domain. We then have

$$(\mathsf{ind}_{\mathbf{2}}(E, e_0, e_1, 0_{\mathbf{2}}), \mathsf{ind}_{\mathbf{2}}(E, e_0, e_1, 1_{\mathbf{2}})) \equiv (e_0, e_1)$$

by the computation rule for $\mathsf{ind}_{\mathbf{2}}$. For the other direction, suppose that $f : \prod_{(b:\mathbf{2})} E(b)$, so that once around the loop gives $\mathsf{ind}_{\mathbf{2}}(E, f(0_{\mathbf{2}}), f(1_{\mathbf{2}}))$. Suppose that $b : \mathbf{2}$. Then there are two cases:

- $b \equiv 0_{\mathbf{2}}$ gives $\mathsf{ind}_{\mathbf{2}}(E, f(0_{\mathbf{2}}), f(1_{\mathbf{2}}), 0_{\mathbf{2}}) \equiv f(0_{\mathbf{2}})$
- $b \equiv 1_{\mathbf{2}}$ gives $\mathsf{ind}_{\mathbf{2}}(E, f(0_{\mathbf{2}}), f(1_{\mathbf{2}}), 1_{\mathbf{2}}) \equiv f(1_{\mathbf{2}})$

by the computational rule for $\mathsf{ind}_{\mathbf{2}}$. By function extensionality, then, the result is equal to f.

```
Definition Bool_rect_uncurried (E : Bool → Type) :
  (E false) × (E true) → (∀ b, E b).
  intros p b. destruct b; [apply (snd p) | apply (fst p)].
Defined.
Definition Bool_rect_uncurried_inv (E : Bool → Type) :
  (∀ b, E b) → (E false) × (E true).
  intro f. split; [apply (f false) | apply (f true)].
Defined.
Theorem ex5_4 (E : Bool → Type) : IsEquiv (Bool_rect_uncurried E).
Proof.
  refine (isequiv_adjointify _ (Bool_rect_uncurried_inv E) _ _);
    unfold Bool_rect_uncurried, Bool_rect_uncurried_inv.
  intro f. apply path_forall; intro b. destruct b; reflexivity.
  intro p. apply eta_prod.
Qed.
```

Exercise 5.5 (p. 175) Show that the analogous statement to Exercise 5.4 for $\mathbb{N}$ fails.

Solution The analogous statement is that

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

is an equivalence. To show that it fails, note that an element of the domain is a recurrence (e_z, e_s). Recalling the solution to Exercise 5.3, we have recurrences (e_z, e_s) and (e'_z, e'_s) such that $(e_z, e_s) \neq (e'_z, e'_s)$, but such that $\mathsf{ind}_{\mathbb{N}}(E, e_z, e_s) = \mathsf{ind}_{\mathbb{N}}(E, e'_z, e'_s)$. Suppose for contradiction that $\mathsf{ind}_{\mathbb{N}}(E)$ has a quasi-inverse $\mathsf{ind}^{-1}_{\mathbb{N}}(E)$. Then

$$(e_z, e_s) = \mathsf{ind}^{-1}_{\mathbb{N}}(E, \mathsf{ind}_{\mathbb{N}}(E, e_z, e_s)) = \mathsf{ind}^{-1}_{\mathbb{N}}(E, \mathsf{ind}_{\mathbb{N}}(E, e'_z, e'_s)) = (e'_z, e'_s)$$

The first and third equality are from the fact that a quasi-inverse is a left inverse. The second comes from the fact that $\mathsf{ind}_{\mathbb{N}}(E)$ sends the two recurrences to the same function. So we have derived a contradiction.

```
Definition nat_rect_uncurried (E : nat → Type) :
  (E O) × (∀ n, E n → E (S n)) → ∀ n, E n.
  intros p n. induction n. apply (fst p). apply (snd p). apply IHn.
Defined.

Theorem ex5_5 : ¬ IsEquiv (nat_rect_uncurried (fun _ ⇒ Bool)).
Proof.
  intro e. destruct e.
  set (ez := (Ex3.ez Bool true)).
  set (es := (Ex3.es Bool)).
  set (ez' := (Ex3.ez' Bool true)).
  set (es' := (Ex3.es' Bool true)).
  assert ((ez, es) = (ez', es')) as H.
  transitivity (equiv_inv (nat_rect_uncurried (fun _ ⇒ Bool) (ez, es))).
  symmetry. apply eissect.
  transitivity (equiv_inv (nat_rect_uncurried (fun _ ⇒ Bool) (ez', es'))).
  apply (ap equiv_inv). apply path_forall; intro n. induction n.
    reflexivity.
    simpl. rewrite IHn. unfold Ex3.es, Ex3.es'. induction n; reflexivity.
  apply eissect.
  assert (¬ ((ez, es) = (ez', es'))) as nH.
    apply (Ex3.ex5_3 Bool true false). apply true_ne_false.
    contradiction nH.
Qed.
```

***Exercise 5.6 (p. 175)** Show that if we assume simple instead of dependent elimination for W-types, the uniqueness property fails to hold. That is, exhibit a type satisfying the recursion principle of a W-type, but for which functions are not determined uniquely by their recurrence.

***Exercise 5.7 (p. 175)** Suppose that in the "inductive definition" of the type C at the beginning of §5.6, we replace the type $\mathbb{N}$ by $\mathbf{0}$. Analogously to 5.6.1, we might consider a recursion principle for this type with hypothesis

$$h : (C \to \mathbf{0}) \to (P \to \mathbf{0}) \to P.$$

Show that even without a computation rule, this recursion principle is inconsistent, i.e. it allows us to construct an element of $\mathbf{0}$.

Solution The associated recursion principle is

$$\mathsf{rec}_C : \prod_{P:\mathcal{U}} ((C \to \mathbf{0}) \to (P \to \mathbf{0}) \to P) \to C \to P$$

***Exercise 5.8 (p. 175)** Consider now an "inductive type" D with one constructor $\mathsf{scott} : (D \to D) \to D$. The second recursor for C suggested in §5.6 leads to the following recursor for D:

$$\mathsf{rec}_D : \prod_{P:\mathcal{U}} ((D \to D) \to (D \to P) \to P) \to D \to P$$

with computation rule $\mathsf{rec}_D(P, h, \mathsf{scott}(\alpha)) \equiv h(\alpha, (\lambda d.\, \mathsf{rec}_D(P, h, \alpha(d))))$. Show that this also leads to a contradiction.

Exercise 5.9 (p. 176) Let A be an arbitrary type and consider generally an "inductive definition" of a type L_A with constructor $\mathsf{lawvere} : (L_A \to A) \to L_A$. The second recursor for C suggested in §5.6 leads to the following recursor for L_A:

$$\mathsf{rec}_{L_A} : \prod_{P:\mathcal{U}} ((L_A \to A) \to P) \to L_A \to P$$

with computation rule $\mathsf{rec}_{L_A}(P, h, \mathsf{lawvere}(\alpha)) \equiv h(\alpha)$. Using this, show that A has a *fixed-point property*, i.e. for every function $f : A \to A$ there exists an $a : A$ such that $f(a) = a$. In particular, L_A is inconsistent if A is a type without the fixed-point property, such as $\mathbf{0}$, $\mathbf{2}$, $\mathbb{N}$.

Solution This is an instance of Lawvere's fixed-point theorem, which says that in a cartesian closed category, if there is a point-surjective map $T \to A^T$, then every endomorphism $f : A \to A$ has a fixed point. Working at an intuitive level, the recursion principle ensures that we have the required properties of a point-surjective map in a CCC. In particular, we have the map $\phi : (L_A \to A) \to A^{L_A \to A}$ given by

$$\phi :\equiv \lambda(f : L_A \to A).\, \lambda(\alpha : L_A \to A).\, f(\mathsf{lawvere}(\alpha))$$

and for any $h : A^{L_A \to A}$, we have

$$\phi(\mathsf{rec}_{L_A}(A, h)) \equiv \lambda(\alpha : L_A \to A).\, \mathsf{rec}_{L_A}(A, h, \mathsf{lawvere}(\alpha)) \equiv \lambda(\alpha : L_A \to A).\, h(\alpha) = h$$

So we can recap the proof of Lawvere's fixed-point theorem with this ϕ. Suppose that $f : A \to A$, and define

$$q :\equiv \lambda(\alpha : L_A \to A).\, f(\phi(\alpha, \alpha)) : (L_A \to A) \to A$$
$$p :\equiv \mathsf{rec}_{L_A}(A, q) : L_A \to A$$

so that p lifts q:

$$\phi(p) \equiv \lambda(\alpha : L_A \to A).\, \mathsf{rec}_{L_A}(A, q, \mathsf{lawvere}(\alpha)) \equiv \lambda(\alpha : L_A \to A).\, q(\alpha) = q$$

This make $\phi(p, p)$ a fixed point of f:

$$f(\phi(p, p)) = (\lambda(\alpha : L_A \to A).\, f(\phi(\alpha, \alpha)))(p) = q(p) = \phi(p, p)$$

```
Definition onto {X Y} (f : X → Y) := ∀ y : Y, {x : X & f x = y}.
Lemma LawvereFP {X Y} (phi : X → (X → Y)) :
  onto phi → ∀ (f : Y → Y), {y : Y & f y = y}.
Proof.
```

```
  intros Hphi f.
  set (q := fun x ⇒ f (phi x x)).
  set (p := Hphi q). destruct p as [p Hp].
  ∃ (phi p p).
  change (f (phi p p)) with ((fun x ⇒ f (phi x x)) p).
  change (fun x ⇒ f (phi x x)) with q.
  symmetry. apply (apD10 Hp).
Defined.

Module Ex9.
Section Ex9.

Variable (L A : Type).
Variable lawvere : (L → A) → L.
Variable rec : ∀ P, ((L → A) → P) → L → P.
Hypothesis rec_comp : ∀ P h alpha, rec P h (lawvere alpha) = h alpha.

Definition phi : (L → A) → ((L → A) → A) :=
  fun f alpha ⇒ f (lawvere alpha).

Theorem ex5_9 : ∀ (f : A → A), {a : A & f a = a}.
Proof.
  intro f. apply (LawvereFP phi).
  intro q. ∃ (rec A q). unfold phi.
  change q with (fun alpha ⇒ q alpha).
  apply path_forall; intro alpha. apply rec_comp.
Defined.

End Ex9.
End Ex9.
```

Exercise 5.10 (p. 176) Continuing from Exercise 5.9, consider $L_\mathbf{1}$, which is not obviously inconsistent since $\mathbf{1}$ does have the fixed-point property. Formulate an induction principle for $L_\mathbf{1}$ and its computation rule, analogously to its recursor, and using this, prove that it is contractible.

Solution The induction principle for $L_\mathbf{1}$ is

$$\mathsf{ind}_{L_\mathbf{1}} : \prod_{P:L_\mathbf{1}\to\mathcal{U}} \left(\prod_{\alpha:L_\mathbf{1}\to\mathbf{1}} P(\mathsf{lawvere}(\alpha)) \right) \to \prod_{\ell:L_\mathbf{1}} P(\ell)$$

and it has the computation rule

$$\mathsf{ind}_{L_\mathbf{1}}(P, f, \mathsf{lawvere}(\alpha)) \equiv f(\alpha)$$

for all $f : \prod_{(\alpha:L_\mathbf{1}\to\mathbf{1})} P(\mathsf{lawvere}(\alpha))$ and $\alpha : L_\mathbf{1} \to \mathbf{1}$.

Let $! : L_\mathbf{1} \to \mathbf{1}$ be the unique terminal arrow. I claim that $L_\mathbf{1}$ is contractible with center $\mathsf{lawvere}(!)$. By $\mathsf{ind}_{L_\mathbf{1}}$, it suffices to show that $\mathsf{lawvere}(!) = \mathsf{lawvere}(\alpha)$ for any $\alpha : L_\mathbf{1} \to \mathbf{1}$. And by the universal property of the terminal object, $\alpha = !$, so we're done.

```
Module Ex10.
Section Ex10.

Variable L : Type.
Variable lawvere : (L → Unit) → L.

Variable indL : ∀ P, (∀ alpha, P (lawvere alpha)) → ∀ l, P l.
Hypothesis ind_comp : ∀ P f alpha, indL P f (lawvere alpha) = f alpha.
```

```
Theorem ex5_10 : Contr L.
Proof.
  apply (BuildContr L (lawvere (fun _ ⇒ tt))).
  apply indL; intro alpha.
  apply (ap lawvere).
  apply allpath_hprop.
Defined.

End Ex10.
End Ex10.
```

Exercise 5.11 (p. 176) In §5.1 we defined the type $\mathsf{List}(A)$ of finite lists of elements of some type A. Consider a similiar inductive definition of a type $\mathsf{Lost}(A)$, whose only constructor is

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

Show that $\mathsf{Lost}(A)$ is equivalent to $\mathbf{0}$.

Solution Consider the recursor for $\mathsf{Lost}(A)$, given by

$$\mathsf{rec}_{\mathsf{Lost}(A)} : \prod_{P:\mathcal{U}} (A \to \mathsf{Lost}(A) \to P \to P) \to \mathsf{Lost}(A) \to P$$

with computation rule

$$\mathsf{rec}_{\mathsf{Lost}(A)}(P, f, \mathsf{cons}(h, t)) \equiv f(h, \mathsf{rec}_{\mathsf{Lost}(A)}(P, f, t))$$

Now $\mathsf{rec}_{\mathsf{Lost}(A)}(\mathbf{0}, \lambda a.\, \lambda \ell.\, \mathsf{id}_{\mathbf{0}}) : \mathsf{Lost}(A) \to \mathbf{0}$, so $\neg\mathsf{Lost}(A)$ is inhabited, thus $\mathsf{Lost}(A) \simeq \mathbf{0}$.

```
Theorem not_equiv_empty (A : Type) : ¬ A → (A ≃ Empty).
Proof.
  intro nA. refine (equiv_adjointify nA (Empty_rect (fun _ ⇒ A)) _ _);
  intro; contradiction.
Defined.

Module Ex11.

Inductive lost (A : Type) := cons : A → lost A → lost A.

Theorem ex5_11 (A : Type) : lost A ≃ Empty.
Proof.
  apply not_equiv_empty.
  intro l.
  apply (lost_rect A). auto. apply l.
Defined.

End Ex11.
```

6 Higher Inductive Types

Exercise 6.1 (p. 217) Define concatenation of dependent paths, prove that application of dependent functions preserves concatenation, and write out the precise induction principle for the torus T^2 with its computation rules.

Solution I found Kristina Sojakova's answer posted to the HoTT Google group helpful here, though I think my answer differs.

Let $W : \mathcal{U}$, $P : W \to \mathcal{U}$, $x, y, z : W$, $u : P(x)$, $v : P(y)$, and $w : P(z)$ with $p : x = y$ and $q : y = z$. We define the map

$$\centerdot_D : (u =^P_p v) \to (v =^P_q w) \to (u =^P_{p \centerdot q} w)$$

by path induction.

```
Definition concatD {A} {P : A → Type} {x y z : A}
                   {u : P x} {v : P y} {w : P z}
                   {p : x = y} {q : y = z} :
  (p # u = v) → (q # v = w) → ((p @ q) # u = w).
  by path_induction.
Defined.

Notation "p @D q" := (concatD p q)%path (at level 20) : path_scope.
```

To prove that application of dependent functions preserves concatenation, we must show that for any $f : \prod_{(x:A)} P(x)$, $p : x = y$, and $q : y = z$,

$$\mathsf{apd}_f(p \centerdot q) = \mathsf{apd}_f(p) \centerdot_D \mathsf{apd}_f(q)$$

which is immediate by path induction.

```
Theorem apD_pp {A} {P : A → Type} (f : ∀ x : A, P x)
               {x y z : A} (p : x = y) (q : y = z) :
  apD f (p @ q) = (apD f p) @D (apD f q).
Proof.
  by path_induction.
Defined.
```

Suppose that we have a family $P : T^2 \to \mathcal{U}$, a point $b' : P(b)$, paths $p' : b' =^P_p b'$ and $q' : b' =^P_q b'$ and a 2-path $t' : p' \centerdot_D q' =^P_t q' \centerdot_D p'$. Then the induction principle gives a section $f : \prod_{(x:T^2)} P(x)$ such that $f(b) \equiv b'$, $f(p) = p'$, and $f(q) = q'$. As discussed in the text, we should also have $\mathsf{apd}^2_f(t) = t'$, but this is not well-typed. This is because $\mathsf{apd}^2_f(t) : f(p \centerdot q) =^P_t f(q \centerdot p)$, in contrast to the type of t', and the two types are not judgementally equal.

To cast $\mathsf{apd}^2_f(t)$ as the right type, note first that, as just proven, $f(p \centerdot q) = f(p) \centerdot_D f(q)$, and $f(q \centerdot p) = f(q) \centerdot_D f(p)$. The computation rules for the 1-paths can be lifted as follows. Let $r, r' : u =^P_p v$, and $s, s' : v =^P_q w$. Then we define a map

$$\centerdot^2_D : (r = r') \to (s = s') \to (r \centerdot_D s = r' \centerdot_D s')$$

by path induction.

```
Definition concat2D {A : Type} {P : A → Type}
                    {x y z : A} {p : x = y} {q : y = z}
                    {u : P x} {v : P y} {w : P z}
                    {r r' : p # u = v} {s s' : q # v = w} :
  (r = r') → (s = s') → (r @D s = r' @D s').
  by path_induction.
Defined.

Notation "p @@D q" := (concat2D p q)%path (at level 20) : path_scope.
```

Thus by the computation rules for p and q, we have for $\alpha : f(p) = p'$ and $\beta : f(q) = q'$,

$$\alpha \centerdot^2_D \beta : f(p) \centerdot_D f(q) = p' \centerdot_D q'$$
$$\beta \centerdot^2_D \alpha : f(q) \centerdot_D f(p) = q' \centerdot_D p'$$

At this point it's pretty clear how to assemble the computation rule. Let $N_1 : f(p \cdot q) = f(p) \cdot_D f(q)$ and $N_2 : f(q \cdot p) = f(q) \cdot_D f(p)$ be two instances of the naturality proof just given. Then we have

$$(\alpha \cdot^2_D \beta)^{-1} \cdot N_1^{-1} \cdot \mathsf{apd}^2_f(t) \cdot (\mathsf{transport}^2(t, b') \cdot_{\mathsf{l}} N_2) \cdot (\mathsf{transport}^2(t, b') \cdot_{\mathsf{l}} (\beta \cdot^2_D \alpha)) : p' \cdot_D q' =^P_t q' \cdot_D p'$$

which is the type of t'.

```
Module Torus_ex.

Private Inductive T2 : Type :=
| Tb : T2.

Axiom Tp : Tb = Tb.
Axiom Tq : Tb = Tb.
Axiom Tt : Tp @ Tq = Tq @ Tp.

Definition T2_rect (P : T2 → Type)
           (b' : P Tb) (p' : Tp # b' = b') (q' : Tq # b' = b')
           (t' : p' @D q' = (transport2 P Tt b') @ (q' @D p'))
  : ∀ (x : T2), P x :=
  fun x ⇒ match x with Tb ⇒ fun _ _ _ ⇒ b' end p' q' t'.

Axiom T2_rect_beta_Tp :
  ∀ (P : T2 → Type)
         (b' : P Tb) (p' : Tp # b' = b') (q' : Tq # b' = b')
         (t' : p' @D q' = (transport2 P Tt b') @ (q' @D p')),
    apD (T2_rect P b' p' q' t') Tp = p'.

Axiom T2_rect_beta_Tq :
  ∀ (P : T2 → Type)
         (b' : P Tb) (p' : Tp # b' = b') (q' : Tq # b' = b')
         (t' : p' @D q' = (transport2 P Tt b') @ (q' @D p')),
    apD (T2_rect P b' p' q' t') Tq = q'.

Axiom T2_rect_beta_Tt :
  ∀ (P : T2 → Type)
         (b' : P Tb) (p' : Tp # b' = b') (q' : Tq # b' = b')
         (t' : p' @D q' = (transport2 P Tt b') @ (q' @D p')),
    (T2_rect_beta_Tp P b' p' q' t' @@D T2_rect_beta_Tq P b' p' q' t')^
    @ (apD_pp (T2_rect P b' p' q' t') Tp Tq)^
    @ (apD02 (T2_rect P b' p' q' t') Tt)
    @ (whiskerL (transport2 P Tt (T2_rect P b' p' q' t' Tb))
                (apD_pp (T2_rect P b' p' q' t') Tq Tp))
    @ (whiskerL (transport2 P Tt (T2_rect P b' p' q' t' Tb))
                (T2_rect_beta_Tq P b' p' q' t' @@D T2_rect_beta_Tp P b' p' q' t'))
    = t'.

End Torus_ex.
```

Exercise 6.2 (p. 217) Prove that $\Sigma\mathbb{S}^1 \simeq \mathbb{S}^2$, using the explicit definition of $\mathbb{S}^2$ in terms of base and surf given in §6.4.

Solution $\mathbb{S}^2$ is generated by

- $\mathsf{base}_2 : \mathbb{S}^2$
- $\mathsf{surf} : \mathsf{refl}_{\mathsf{base}_2} = \mathsf{refl}_{\mathsf{base}_2}$

and $\Sigma\mathbb{S}^1$ is generated by

- $\mathsf{N} : \Sigma\mathbb{S}^1$
- $\mathsf{S} : \Sigma\mathbb{S}^1$
- $\mathsf{merid} : \mathbb{S}^1 \to (\mathsf{N} = \mathsf{S})$.

To define a map $f : \Sigma\mathbb{S}^1 \to \mathbb{S}^2$, we need a map $m : \mathbb{S}^1 \to (\mathsf{base}_2 = \mathsf{base}_2)$, which we define by circle recursion such that $m(\mathsf{base}_1) \equiv \mathsf{refl}_{\mathsf{base}_2}$ and $m(\mathsf{loop}) = \mathsf{surf}$. Then recursion on $\Sigma\mathbb{S}^1$ gives us our f, and we have $f(\mathsf{N}) \equiv \mathsf{base}_2$; $f(\mathsf{S}) \equiv \mathsf{base}_2$; and for all $x : \mathbb{S}^1$, $f(\mathsf{merid}(x)) = m(x)$.

To go the other way, we use the recursion principle for the 2-sphere to obtain a function $g : \mathbb{S}^2 \to \Sigma\mathbb{S}^1$ such that $g(\mathsf{base}_2) \equiv \mathsf{N}$ and $\mathsf{ap}^2_g(\mathsf{surf}) = \mathsf{merid}(\mathsf{loop}) \cdot_{\mathsf{r}} \mathsf{merid}(\mathsf{base}_1)^{-1}$, conjugated with proofs that $\mathsf{merid}(\mathsf{base}_1) \cdot \mathsf{merid}(\mathsf{base}_1)^{-1} = \mathsf{refl}_{\mathsf{N}}$.

Now, to show that this is an equivalence, we must show that the second map is a quasi-inverse to the first. First we show $g \circ f \sim \mathsf{id}_{\Sigma\mathbb{S}^1}$. For the poles we have

$$g(f(\mathsf{N})) \equiv g(\mathsf{base}_2) \equiv \mathsf{N}$$
$$g(f(\mathsf{S})) \equiv g(\mathsf{base}_2) \equiv \mathsf{N}$$

and concatenating the latter with $\mathsf{merid}(\mathsf{base}_1)$ gives $g(f(\mathsf{S})) = \mathsf{S}$. Now we must show that for all $y : \mathbb{S}^1$, these equalities hold as x varies along $\mathsf{merid}(y)$. That is, we must produce a path

$$\mathsf{transport}^{x \mapsto g(f(x))=x}(\mathsf{merid}(y), \mathsf{refl}_{\mathsf{N}}) = \mathsf{merid}(\mathsf{base}_1)$$

or, by Theorem 2.11.3 and a bit of path algebra,

$$g(m(y)) = \mathsf{merid}(y) \cdot \mathsf{merid}(\mathsf{base}_1)^{-1}$$

We do this by induction on $\mathbb{S}^1$. When $y \equiv \mathsf{base}_1$, we have

$$g(m(\mathsf{base}_1)) = g(\mathsf{refl}_{\mathsf{base}_2}) = \mathsf{refl}_{\mathsf{base}_2} = \mathsf{merid}(\mathsf{base}_1) \cdot \mathsf{merid}(\mathsf{base}_1)^{-1}$$

When y varies along loop, we have to show that this proof continues to hold. By Theorem 2.11.3 and some path algebra, this in fact reduces to

$$\mathsf{ap}_{y \mapsto g(m(y))}\mathsf{loop} = \mathsf{merid}(\mathsf{loop}) \cdot_{\mathsf{r}} \mathsf{merid}(\mathsf{base}_1)^{-1}$$

modulo the proofs of $\mathsf{merid}(\mathsf{base}_1) \cdot \mathsf{merid}(\mathsf{base}_1)^{-1} = \mathsf{refl}_{\mathsf{N}}$. And this is essentially the computation rule for g. Since the computation rules are propositional some extra proofs have to be carried around, though; see the second part of isequiv_SS1_to_S2 for the gory details.

To show that $f \circ g \sim \mathsf{id}_{\mathbb{S}^2}$, note that

$$f(g(\mathsf{base}_2)) \equiv f(\mathsf{N}) \equiv \mathsf{base}_2$$

so we only need to show that as x varies over the surface,

$$\mathsf{refl}_{\mathsf{refl}_{\mathsf{base}_2}} =^{x \mapsto f(g(x))=x}_{\mathsf{surf}} \mathsf{refl}_{\mathsf{refl}_{\mathsf{base}_2}}$$

which means

$$\mathsf{refl}_{\mathsf{refl}_{\mathsf{base}_2}} = \mathsf{transport}^2(\mathsf{surf}, \mathsf{refl}_{\mathsf{base}_2}) \equiv \mathsf{ap}_{p \mapsto p_* \mathsf{refl}_{\mathsf{base}_2}} \mathsf{surf}$$

So we need to show that

$$\mathsf{refl}_{\mathsf{refl}_{\mathsf{base}_2}} = \mathsf{ap}_{p \mapsto f(g(p^{-1})) \cdot p} \mathsf{surf}$$

which by naturality of ap and the computation rule for $\mathbb{S}^1$ is

$$\mathsf{refl}_{\mathsf{refl}_{\mathsf{base}_2}} = \left(\mathsf{ap}^2_f\left(\mathsf{ap}^2_g(\mathsf{surf})\right)\right)^{-1} \cdot \mathsf{surf}$$

Naturality and the computation rules then give

$$\mathsf{ap}_f^2\left(\mathsf{ap}_g^2(\mathsf{surf})\right) = \mathsf{ap}_f^2(\mathsf{merid}(\mathsf{loop})) = m(\mathsf{loop}) = \mathsf{surf}$$

and we're done. The computation in Coq is long, since all of the homotopic corrections have to be done by hand. I also spend a lot of moves on making the interactive version of the proof clear, and those could probably be eliminated.

```
Lemma ap_ap__ap02_ap {A B C : Type} {a : A} {b : B}
        (p : a = a) (f : B → C) (m : A → (b = b)) :
  ap (fun x : A ⇒ ap f (m x)) p = ap02 f (ap m p).
Proof. by path_induction. Defined.

Lemma whiskerR_ap {A B : Type} {a : A} {b b' : B}
        (f : A → (b = b')) (p : a = a) (q : b' = b) :
  whiskerR (ap f p) q = ap (fun x ⇒ f x @ q) p.
Proof. by path_induction. Defined.

Definition SS1_to_S2 := Susp_rect_nd base base (S1_rectnd (base = base) 1 surf).

Definition S2_to_SS1 :=
  S2_rectnd (Susp S1) North
                 ((concat_pV (merid Circle.base))^
                  @ (whiskerR (ap merid loop) (merid Circle.base)^)
                  @ (concat_pV (merid Circle.base)))^.

Lemma ap_concat (A B : Type) (a : A) (b b' b'' : B) (p : a = a)
        (f : A → (b = b')) (g : A → (b' = b'')) :
  ap (fun x ⇒ f x @ g x) p = (ap f p) @@ (ap g p).
Proof. by path_induction. Defined.

Definition concat2_V2p {A : Type} {x y : A} {p q : x = y} (r : p = q) :
  (concat_Vp _)^ @ (inverse2 r @@ r) @ (concat_Vp _) = 1.
Proof. by path_induction. Defined.

Definition ap_inv {A B : Type} {a : A} {b : B} (p : a = a) (m : A → (b = b)):
  ap (fun x : A ⇒ (m x)^) p = inverse2 (ap m p).
Proof. by path_induction. Defined.

Lemma bar (A B C : Type) (b b' : B) (m : A → (b = b')) (f : B → C)
        (a : A) (p : a = a) :
  ap (fun x : A ⇒ ap f (m x)) p = ap02 f (ap m p).
Proof.
  by path_induction.
Defined.

Definition ap03 {A B : Type} (f : A → B) {x y : A}
              {p q : x = y} {r s : p = q} (t : r = s) :
  ap02 f r = ap02 f s
  := match t with idpath ⇒ 1 end.

Definition ap_pp_pV {A B : Type} (f : A → B)
              {x y : A} (p : x = y) :
  ap_pp f p p^
  = (ap02 f (concat_pV p)^)^ @ (concat_pV (ap f p))^ @ (1 @@ (ap_V f p))^.
Proof.
  by path_induction.
Defined.

Definition ap02_V {A B : Type} (f : A → B)
```

```
             {x y : A} {p q : x = y} (r : p = q) :
  ap02 f r^ = (ap02 f r)^.
Proof. by path_induction. Defined.

Definition inv_p2p {A : Type} {x y z : A}
             {p p' : x = y} {q q' : y = z} (r : p = p') (s : q = q') :
  (r @@ s)^ = r^ @@ s^.
Proof. by path_induction. Defined.

Definition baz {A : Type} {x y : A} {p q : x = y} (r : p = q) :
  concat_pV p = (r @@ inverse2 r) @ concat_pV q.
Proof. by path_induction. Defined.

Theorem isequiv_SS1_to_S2 : IsEquiv (SS1_to_S2).
Proof.
  apply isequiv_adjointify with S2_to_SS1.

  (*  SS1_to_S2 o S2_to_SS1 == id *)
  refine (S2_rect (fun x => SS1_to_S2 (S2_to_SS1 x) = x) 1 _).
  unfold transport2.
  transparent assert (H : (forall p : base = base,
      (fun p' : base = base => transport (fun x => SS1_to_S2 (S2_to_SS1 x) = x) p' 1) p
      =
      (fun p' : base = base => ap SS1_to_S2 (ap S2_to_SS1 p'^) @ p') p
  )).
  intros. refine ((transport_paths_FFlr _ _) @ _).
  apply whiskerR. refine ((concat_p1 _) @ _).
  refine ((ap_V SS1_to_S2 (ap S2_to_SS1 p))^ @ _). f_ap.
  apply (ap_V _ _)^.
  transitivity ((H 1)
                 @ ap (fun p : base = base => ap SS1_to_S2 (ap S2_to_SS1 p^) @ p) surf
                 @ (H 1)^).
  apply moveL_pV.
  apply (@concat_Ap _ _ _ _ H _ _ _).
  hott_simpl. clear H.
  transitivity (ap (fun p : base = base => ap SS1_to_S2 (ap S2_to_SS1 p^) @ p)
                  (ap (S1_rectnd (base = base) 1 surf) loop)).
  f_ap. apply (S1_rectnd_beta_loop _ _ _)^.
  refine ((ap_compose _ _ _)^ @ _). unfold compose.
  refine ((ap_concat S1 _ _ _ _ _ _ _ _ _) @ _).
  transitivity (ap (fun x : S1 =>
                    ap SS1_to_S2 (ap S2_to_SS1 (S1_rectnd (base = base) 1 surf x)^)) loop
                  @@ surf).
  f_ap. apply S1_rectnd_beta_loop.
  refine (_ @ (concat2_V2p surf)). hott_simpl. f_ap.

  (*  invert the equation *)
  transparent assert (H : (
    forall x : S1,
      ap SS1_to_S2 (ap S2_to_SS1 (S1_rectnd (base = base) 1 surf x)^)
      =
      (ap SS1_to_S2 (ap S2_to_SS1 (S1_rectnd (base = base) 1 surf x)))^
  )).
  intro x.
  refine (_ @ (ap_V _ _)). f_ap.
```

```
refine (_ @ (ap_V _ _)). reflexivity.
transitivity (
    (H Circle.base)
    @ ap (fun x : S1 ⇒ (ap SS1_to_S2 (ap S2_to_SS1 (S1_rectnd (base = base) 1 surf x)))^)
      loop
    @ (H Circle.base)^
).
apply moveL_pV. apply (concat_Ap H loop).
simpl. hott_simpl. clear H.
refine ((ap_inv loop
                (fun x ⇒ ap SS1_to_S2 (ap S2_to_SS1 (S1_rectnd (base = base) 1 surf x)))
                ) @ _).
f_ap.
(*  reduce to SS1_to_S2 (S2_to_SS1 surf) = surf *)
refine ((bar _ _ _ _ _ _ _ _ loop) @ _).
refine ((ap03 _ (bar _ _ _ _ _ _ _ _ loop)) @ _).
refine ((ap03 _ (ap03 _ (S1_rectnd_beta_loop _ _ _))) @ _).
(*  compute the action of S2_to_SS1 *)
refine ((ap03 _ (S2_rectnd_beta_surf _ _ _)) @ _).
hott_simpl.
refine ((ap02_pp _ _ _) @ _). apply moveR_pM.
refine ((ap02_pp _ _ _) @ _). apply moveR_Mp.
refine ((ap03 _ (whiskerR_ap _ _ _)) @ _).
refine ((ap03 _ (ap_concat _ _ _ _ _ _ _ merid _)) @ _).
hott_simpl.
refine ((ap02_p2p _ _ _) @ _). hott_simpl. apply moveR_pV. apply moveR_pM.
(*  eliminate ap_pps *)
refine ((ap_pp_pV _ _) @ _). apply moveL_pV. apply moveL_Mp.
refine (_ @ (ap_pp_pV _ _)^). repeat (apply moveR_pM).
refine ((inv_pp _ _) @ _). apply moveR_Vp.
refine ((inv_pp _ _) @ _). apply moveR_pV. hott_simpl.
repeat (apply moveL_pM). apply moveR_pM. hott_simpl.
apply moveL_Mp. refine (_ @ (ap02_V _ _)).
refine (_ @ (ap03 _ (inv_V _)^)). apply moveR_Vp. hott_simpl.
apply moveR_pM. hott_simpl.
repeat (refine ((concat_pp_p _ _ _) @ _)). apply moveR_Vp. apply moveR_Vp.
apply moveR_Vp. refine ((concat_concat2 _ _ _ _) @ _).
apply moveL_Mp. apply moveR_pM. refine ((inv_p2p _ _) @ _). apply moveL_pV.
refine ((concat_concat2 _ _ _ _) @ _). hott_simpl.
(*  eliminate the concat_pVs *)
transparent assert (r : (
  ap SS1_to_S2 (merid Circle.base) = 1
)).
change 1 with (S1_rectnd (base = base) 1 surf Circle.base).
apply (Susp_comp_nd_merid Circle.base).
apply moveL_pV. apply moveL_pM.
refine (_ @ (baz r)^). hott_simpl.
apply moveR_pV. apply moveR_Mp.
refine ((baz r) @ _). apply moveL_Vp. hott_simpl.
refine ((concat_concat2 _ r 1 (inverse2 r)) @ _). simpl. hott_simpl.
```

```
apply moveL_Mp. apply moveR_pM.
refine ((inv_p2p r (inverse2 r)) @ _).
apply moveL_pV.
refine ((concat_concat2 r^ (ap02 SS1_to_S2 (ap merid loop) @ r) (inverse2 r)^ (inverse2 r)) @ _).
hott_simpl.

(*  de-whisker *)
refine ((concat2_p1 _) @ _).
transitivity ((concat_p1 _)
                @ ((r^ @ ap02 SS1_to_S2 (ap merid loop)) @ r)
                @ (concat_p1 _)^).
apply moveL_pV. apply moveL_Mp.
refine ((concat_p_pp _ _ _) @ _).
apply whiskerR_p1. simpl. hott_simpl.
apply moveR_pM. apply moveR_Vp.
unfold r. clear r.

(*  compute the action of SS1_to_S2 *)
refine ((bar _ _ _ _ _ merid _ _ _)^ @ _).
transparent assert (H : (
  ∀ x : S1, ap SS1_to_S2 (merid x) = S1_rectnd _ 1 surf x
)).
apply Susp_comp_nd_merid.
change (Susp_comp_nd_merid Circle.base) with (H Circle.base).
refine (_ @ (concat_pp_p _ _ _)). apply moveL_pV.
refine ((concat_Ap H loop) @ _). f_ap.
apply S1_rectnd_beta_loop.

(*  S2_to_SS2 o SS1_to_S2 == id *)
unfold Sect.
refine (Susp_rect (fun x ⇒ S2_to_SS1 (SS1_to_S2 x) = x) 1 (merid Circle.base) _).

(*  make the goal g(m(y)) = merid(y) @ merid(Circle.base)^ *)
intro x.
  refine ((transport_paths_FFlr _ _) @ _). hott_simpl.
  transitivity (ap S2_to_SS1 (S1_rectnd (base = base) 1 surf x)^ @ merid x).
  repeat f_ap.
  transitivity (ap SS1_to_S2 (merid x))^. hott_simpl.
  apply inverse2.
  apply Susp_comp_nd_merid.
  apply moveR_pM.
  transitivity (ap S2_to_SS1 (S1_rectnd (base = base) 1 surf x))^. hott_simpl.
  symmetry. transitivity (merid x @ (merid Circle.base)^)^.
  symmetry. apply inv_pV. apply inverse2. symmetry.
generalize dependent x.

(*  now compute *)
refine (S1_rect _ _ _).
refine (_ @ (concat_pV _)^). reflexivity.
refine ((@transport_paths_FlFr S1 _ _ _ _ _ loop _) @ _).
hott_simpl.
apply moveR_pM. apply moveR_pM. hott_simpl. refine (_ @ (inv_V _)).
apply inverse2.
transitivity ((concat_pV (merid Circle.base))^
                @ (whiskerR (ap merid loop) (merid Circle.base)^)
```

```
                  @ (concat_pV (merid Circle.base))).
  refine (_ @ (inv_V _)).
  refine (_ @ (S2_rectnd_beta_surf _ _ _)).
  refine ((ap_ap__ap02_ap _ _ _) @ _).
  f_ap. apply S1_rectnd_beta_loop.
  refine (_ @ (inv_pp _ _)^). refine ((concat_pp_p _ _ _) @ _). apply whiskerL.
  refine (_ @ (inv_pp _ _)^). hott_simpl. apply whiskerR.
  apply whiskerR_ap.
Defined.
```

*__Exercise 6.3 (p. 217)__ Prove that the torus T^2 as defined in §6.6 is equivalent to $\mathbb{S}^1 \times \mathbb{S}^1$.

Solution We first define $f : T^2 \to \mathbb{S}^1 \times \mathbb{S}^1$ by torus recursion, using the maps

$$b \mapsto (\mathsf{base}, \mathsf{base})$$
$$p \mapsto \mathsf{pair}^=(\mathsf{refl}_{\mathsf{base}}, \mathsf{loop})$$
$$q \mapsto \mathsf{pair}^=(\mathsf{loop}, \mathsf{refl}_{\mathsf{base}})$$
$$\Phi(\alpha, \alpha) : \lambda(\alpha : x = x'). \lambda(\alpha' : y = y'). \left(\mathsf{pair}^=(\mathsf{refl}_x, \alpha') \cdot \mathsf{pair}^=(\alpha, \mathsf{refl}_{y'})\right) = \left(\mathsf{pair}^=(\alpha, \mathsf{refl}_y) \cdot \mathsf{pair}^=(\mathsf{refl}_{x'}, \alpha')\right)$$
$$t \mapsto \Phi(\mathsf{loop}, \mathsf{loop})$$

Where Φ is defined by recursion on α and α'. To define a function $f : \mathbb{S}^1 \times \mathbb{S}^1 \to T^2$, we need a function $\tilde{f} : \mathbb{S}^1 \to \mathbb{S}^1 \to T^2$, which we'll define by double circle recursion. $\tilde{f}' : \mathbb{S}^1 \to T^2$ is given by $b \mapsto \mathsf{base}$ and $\mathsf{loop} \mapsto p$. Then $\tilde{f}$ is defined by $b \mapsto \tilde{f}'$ and

$$\mathsf{loop} \mapsto$$

```
Definition Phi {A : Type} {x x' y y' : A} (alpha : x = x') (alpha' : y = y')
: ((path_prod (x, y) (x, y') 1 alpha') @ (path_prod (x, y') (x', y') alpha 1))
  = ((path_prod (x, y) (x', y) alpha 1) @ (path_prod (x', y) (x', y') 1 alpha')).
  induction alpha.
  induction alpha'.
  reflexivity.
Defined.
```

*__Exercise 6.4 (p. 217)__ Define dependent n-loops and the action of dependent functions on n-loops, and write down the induction principle for the n-spheres as defined at the end of §6.4.

*__Exercise 6.5 (p. 217)__ Prove that $\Sigma\mathbb{S}^n \simeq \mathbb{S}^{n+1}$, using the definition of $\mathbb{S}^n$ in terms of Ω^n from §6.4.

Solution This definition defines $\mathbb{S}^n$ as the higher inductive type generated by

- $\mathsf{base}_n : \mathbb{S}^n$
- $\mathsf{loop}_n : \Omega^n(\mathbb{S}^n, \mathsf{base})$.

To define a function $\Sigma\mathbb{S}^n \to \mathbb{S}^{n+1}$, we send both N and S to base_{n+1}. So we need a function $m : \mathbb{S}^n \to (\mathsf{base}_{n+1} = \mathsf{base}_{n+1})$, for which we use $\mathbb{S}^n$-recursion.

*__Exercise 6.6 (p. 217)__ Prove that if the type $\mathbb{S}^2$ belongs to some universe $\mathcal{U}$, then $\mathcal{U}$ is not a 2-type.

Exercise 6.7 (p. 217) Prove that if G is a monoid and $x : G$, then $\sum_{(y:G)}((x \cdot y = e) \times (y \cdot x = e))$ is a mere proposition. Conclude, using the principle of unique choice, that it would be equivalent to define a group to be a monoid such that for every $x : G$, there merely exists a $y : G$ such that $x \cdot y = e$ and $y \cdot x = e$.

Solution Suppose that G is a monoid and $x : G$. Since G is a set, each of $x \cdot y = e$ and $y \cdot x = e$ are mere propositions. The product preserves this, so our type is of the form $\sum_{(y:G)} P(y)$ for a family of mere propositions $P : G \to \mathcal{U}$. Now, suppose that there is a point $u : \sum_{(y:G)} P(y)$; we show that this implies that this type is contractible, hence the type is a mere proposition. Since $P(y)$ is a mere proposition, we just need to show that for any point $v : \sum_{(y:G)} P(y)$, $\mathsf{pr}_1 u = \mathsf{pr}_1 v$. But this is just to say that if $\mathsf{pr}_1 u$ has an inverse it is unique, and this is a basic fact about inverses.

A group is defined to be a monoid together with an inversion function $i : G \to G$ such that for all $x : G$, $x \cdot i(x) = e$ and $i(x) \cdot x = e$. That is, the following type is inhabited:

$$\sum_{(i:G\to G)} \prod_{(x:G)} ((x \cdot i(x) = e) \times (i(x) \cdot x = e))$$

but this type is equivalent to the type

$$\prod_{(x:G)} \sum_{(y:G)} ((x \cdot y = e) \times (y \cdot x = e))$$

And as we have just shown, this is of the form $\prod_{(x:G)} Q(x)$ for Q a family of mere propositions. Thus, by the principle of unique choice, it suffices to demand that for each $x : G$ we have $\|Q(x)\|$. Thus these two requirements are equivalent.

```
Class IsMonoid (A : Type) (m : A → A → A) (e : A)
  := BuildIsMonoid {
          m_isset : IsHSet A ;
          m_unitr : ∀ a : A, m a e = a ;
          m_unitl : ∀ a : A, m e a = a ;
          m_assoc : ∀ x y z : A, m x (m y z) = m (m x y) z
       }.

Record Monoid
  := BuildMonoid {
          m_set :> Type ;
          m_mult :> m_set → m_set → m_set ;
          m_unit :> m_set ;
          m_ismonoid :> IsMonoid m_set m_mult m_unit
       }.

Lemma hprop_prod :
  ∀ A, IsHProp A → ∀ B, IsHProp B → IsHProp (A × B).
Proof.
  intros A HA B HB z z'.
  apply (trunc_equiv (equiv_path_prod z z')).
Defined.

Theorem hprop_inverse_exists (G : Monoid) (x : G) :
  IsHProp {y : G & (G x y = G) × (G y x = G)}.
Proof.
  (*  reduce to uniqueness of inverse *)
  assert (∀ y : G, IsHProp ((G x y = G) × (G y x = G))). intro y.
  apply hprop_prod; intros p q; apply G.
  apply hprop_inhabited_contr. intro u. ∃ u.
  intro v. apply path_sigma_hprop.
```

```
  (*  inverse is unique *)
  refine ((@m_unitr _ G G G _)^ @ _).
  refine (_ @ (@m_unitl _ G G G _)).
  transitivity (G u.1 (G x v.1)). f_ap. symmetry. apply (fst v.2).
  transitivity (G (G u.1 x) v.1). refine (@m_assoc G G G G _ _ _).
  f_ap. apply (snd u.2).
Defined.

Class IsGroup (A : Monoid) (i : A → A)
  := BuildIsGroup {
        g_invr : ∀ a : A, (m_mult A) a (i a) = (m_unit A) ;
        g_invl : ∀ a : A, (m_mult A) (i a) a = (m_unit A)
      }.

Record Group
  := BuildGroup {
        g_monoid :> Monoid ;
        g_inv :> (m_set g_monoid) → (m_set g_monoid) ;
        g_isgroup :> IsGroup g_monoid g_inv
      }.

Theorem issig_group :
  {G : Monoid & {i : G → G & ∀ a, (G a (i a) = G) × (G (i a) a = G)}}
    ≃
    Group.
Proof.
  apply (@equiv_compose' _ {G : Monoid & {i : G → G & IsGroup G i}} _).
  issig BuildGroup g_monoid g_inv g_isgroup.
  apply equiv_functor_sigma_id. intro G.
  apply equiv_functor_sigma_id. intro i.
  apply (@equiv_compose' _
                        {_ : ∀ a, (G a (i a) = G)
                             & (∀ a : G, G (i a) a = G)}
                        _).
  issig (BuildIsGroup G i) (@g_invr G i) (@g_invl G i).
  refine (equiv_adjointify _ _ _ _); intro z.
    apply (fun a ⇒ fst (z a); fun a ⇒ snd (z a)).
    apply (fun a ⇒ (z.1 a, z.2 a)).
    destruct z as [g h]. apply path_sigma_uncurried. ∃ 1. reflexivity.
    apply path_forall; intro a. apply eta_prod.
Defined.

(*  this is a bad name for this, but I don't know what to call it and I
   also don't think it's in the HoTT library, at least according to
   HoTTBook.v *)
Definition Book_2_15_6 (X : Type) (A : X → Type) (P : ∀ x : X, A x → Type) :
  (∀ x, {a : A x & P x a}) → {g : ∀ x, A x & ∀ x, P x (g x)}.
Proof.
  intro g. ∃ (fun x ⇒ (g x).1). intro x. apply (g x).2.
Defined.

Theorem Book_2_15_7 (X : Type) (A : X → Type) (P : ∀ x : X, A x → Type) :
  IsEquiv (Book_2_15_6 X A P).
Proof.
  refine (isequiv_adjointify _ _ _ _); intro z.
```

```
    destruct z as [g h]. intro x. apply (g x; h x).
    unfold Book_2_15_6. destruct z as [g h].
    apply path_sigma_uncurried. simpl. ∃ 1. reflexivity.
    apply path_forall; intro x. apply eta_sigma.
Defined.

Theorem ex6_7 :
  {G : Monoid & ∀ x, Brck {y : G & (G x y = G) × (G y x = G)}}
  ≃
  Group.
Proof.
  apply (@equiv_compose' _
                          {G : Monoid &
                           ∀ x : G, {y : G & (G x y = G) × (G y x = G)}}
                          _).
  apply (@equiv_compose' _
                          {G : Monoid &
                          {i : G → G &
                           ∀ a, (G a (i a) = G) × (G (i a) a = G)}}
                          _).
  apply issig_group.
  apply equiv_functor_sigma_id. intro G.
  apply (BuildEquiv _ _
                     (Book_2_15_6 _ _ (fun x y ⇒ (G x y = G) × (G y x = G)))).
  apply Book_2_15_7.
  apply equiv_functor_sigma_id. intro G.
  apply equiv_functor_forall_id. intro x.
  apply equiv_inverse.
  apply (BuildEquiv _ _ min1).
  refine IsEquivmin1.
  apply hprop_inverse_exists.
Defined.
```

Exercise 6.8 (p. 217) Prove that if A is a set, then $\mathsf{List}(A)$ is a monoid. Then complete the proof of Lemma 6.11.5.

Solution We first characterise the path space of $\mathsf{List}(A)$, which goes just as $\mathbb{N}$. We define the codes

$$\begin{aligned}
\mathsf{code}(\mathsf{nil}, \mathsf{nil}) &:\equiv \mathbf{1} \\
\mathsf{code}(\mathsf{cons}(h, t), \mathsf{nil}) &:\equiv \mathbf{0} \\
\mathsf{code}(\mathsf{nil}, \mathsf{cons}(h', t')) &:\equiv \mathbf{0} \\
\mathsf{code}(\mathsf{cons}(h, t), \mathsf{cons}(h', t')) &:\equiv (h = h') \times \mathsf{code}(t, t')
\end{aligned}$$

and the function $r : \prod_{(\ell:\mathsf{List}(A))} \mathsf{code}(\ell, \ell)$ by

$$\begin{aligned}
r(\mathsf{nil}) &:\equiv \star \\
r(\mathsf{cons}(h, t)) &:\equiv (\mathsf{refl}_h, r(t))
\end{aligned}$$

Now, for all $\ell, \ell' : \mathsf{List}(A)$, $(\ell = \ell') \simeq \mathsf{code}(\ell, \ell')$. To prove this, we define

$$\mathsf{encode}(\ell, \ell', p) :\equiv \mathsf{transport}^{\mathsf{code}(\ell, -)}(p, r(\ell))$$

and we define decode by double induction on ℓ, ℓ'. When they're both nil, send everything to $\mathsf{refl}_{\mathsf{nil}}$. When one is nil and the other a cons, we use the eliminator for $\mathbf{0}$. When they're both a cons, we define

$$\begin{aligned}\mathsf{code}(\mathsf{cons}(h,t),\mathsf{cons}(h',t')) &\equiv (h=h')\times \mathsf{code}(t,t') \\ &\xrightarrow{\mathsf{id}_{h=h'}\times\mathsf{decode}(t,t')} (h=h')\times(t=t') \\ &\xrightarrow{\mathsf{pair}^=} ((h,t)=(h',t')) \\ &\xrightarrow{\mathsf{ap}_{\lambda z.\,\mathsf{cons}(\mathsf{pr}_1 z,\mathsf{pr}_2 z)}} (\mathsf{cons}(h,t)=\mathsf{cons}(h',t'))\end{aligned}$$

It follows easily from induction on everything and naturality that these are quasi-inverses. The only point of note is that the fact that A is a set is required in the proof of

$$\mathsf{encode}(\ell,\ell',\mathsf{decode}(\ell,\ell',z)) = z$$

This is because our definition of encode involved an arbitrary choice in how r acts on cons, and this choice is only preserved up to homotopy.

```
Local Open Scope list_scope.

Fixpoint list_code {A : Type} (l l' : list A) : Type :=
  match l with
    | nil ⇒ match l' with
              | nil ⇒ Unit
              | h' :: t' ⇒ Empty
            end
    | h :: t ⇒ match l' with
                | nil ⇒ Empty
                | h' :: t' ⇒ (h = h') × (list_code t t')
              end
  end.

Fixpoint list_r {A : Type} (l : list A) : list_code l l :=
  match l with
    | nil ⇒ tt
    | h :: t ⇒ (1, list_r t)
  end.

Definition list_encode {A : Type} (l l' : list A) (p : l = l') :=
  transport (fun x ⇒ list_code l x) p (list_r l).

Definition list_decode {A : Type} :
  ∀ (l l' : list A) (z : list_code l l'), l = l'.
  induction l as [| h t]; destruct l' as [| h' t']; intros.
    reflexivity. contradiction. contradiction.
    apply (@ap _ _ (fun x ⇒ cons (fst x) (snd x)) (h, t) (h', t')).
    apply path_prod. apply (fst z). apply IHt. apply (snd z).
Defined.

Definition path_list {A : Type} : ∀ (h h' : A) (t t' : list A),
  h = h' → t = t' → h :: t = h' :: t'.
Proof.
  intros h h' t t' ph pt.
  apply (list_decode _ _). split.
    apply ph.
    apply (list_encode _ _). apply pt.
Defined.
```

```
Theorem equiv_path_list {A : Type} {H : IsHSet A} (l l' : list A) :
  (list_code l l') ≃ (l = l').
Proof.
  refine (equiv_adjointify (list_decode l l') (list_encode l l') _ _).
  (*  lst_decode o lst_encode == id *)
  intro p. induction p.
  induction l as [|h t]. reflexivity. simpl in *.
  refine (_ @ (ap_1 _ _)). f_ap.
  transitivity (path_prod (h, t) (h, t) 1 1). f_ap. reflexivity.
  (*  lst_encode o lst_decode == id *)
  generalize dependent l'.
  induction l as [|h t], l' as [|h' t']; intro z.
    apply contr_unit. contradiction. contradiction.
    simpl. unfold list_encode.
    refine ((transport_compose _ _ _ _)^ @ _).
    refine ((transport_prod
              (path_prod (h, t) (h', t') (fst z) (list_decode t t' (snd z)))
              (1, list_r t)) @ _).
    destruct z as [p c].
    apply path_prod. apply H.
    refine ((transport_path_prod _ _ _ _ _ _ _ _) @ _).
    induction p. apply (IHt t').
Defined.
```

It's now easy to see that $\mathsf{List}(A)$ is a set, by induction. If $\ell \equiv \ell' \equiv \mathsf{nil}$, then $(\ell = \ell') \simeq \mathbf{1}$, which is contractible. Similarly, if only one is nil then $(\ell = \ell') \simeq \mathbf{0}$, which is contractible. Finally, if both are conses, then the path space is $(h = h') \times \mathsf{code}(t, t')$. The former is contractible because A is a set, and the latter is contractible by the induction hypothesis. Contractibility is preserved by products, so the path space is contractible.

```
Theorem set_list_is_set (A : Type) : IsHSet A → IsHSet (list A).
Proof.
  intros HA l.
  induction l as [|h t].
    intro l'; destruct l' as [|h' t'].
    apply (trunc_equiv (equiv_path_list nil nil)).
    apply (trunc_equiv (equiv_path_list nil (h' :: t'))).
    intro l'; destruct l' as [|h' t'].
    apply (trunc_equiv (equiv_path_list (h :: t) nil)).
    transparent assert (r : (IsHProp (list_code (h :: t) (h' :: t')))).
    simpl. apply hprop_prod.
    apply hprop_allpath. apply HA.
    apply (trunc_equiv (equiv_path_list t t')^-1).
    apply (trunc_equiv (equiv_path_list (h :: t) (h' :: t'))).
Defined.
```

Now, to show that $\mathsf{List}(A)$ is a monoid, we must equip it with a multiplication function and a unit element. For the multiplication function we use append, and for the unit we use nil. These must satisfy two properties. First we must have, for all $\ell : \mathsf{List}(A)$, $\ell \cdot \mathsf{nil} = \mathsf{nil} \cdot \ell = \ell$, which we clearly do. Second, append must be associative, which it clearly is.

```
(*  move these elsewhere *)

Theorem app_nil_r {A : Type} : ∀ l : list A, l ++ nil = l.
Proof. induction l. reflexivity. simpl. f_ap. Defined.
```

```
Theorem app_assoc {A : Type} : ∀ x y z : list A,
  x ++ (y ++ z) = (x ++ y) ++ z.
Proof.
  intros x y z. induction x. reflexivity.
  simpl. apply path_list. reflexivity. apply IHx.
Defined.

Theorem set_list_is_monoid {A : Type} {HA : IsHSet A} :
  IsMonoid (list A) (@app A) nil.
Proof.
  apply BuildIsMonoid.
  apply set_list_is_set. apply HA.
  apply app_nil_r. reflexivity.
  apply app_assoc.
Defined.
```

Now, Lemma 6.11.5 states that for any set A, the type $\mathsf{List}(A)$ is the free monoid on A. That is, there is an equivalence

$$\hom_{\mathrm{Monoid}}(\mathsf{List}(A), G) \simeq (A \to G)$$

There is an obvious inclusion $\eta : A \to \mathsf{List}(A)$ defined by $a \mapsto \mathsf{cons}(a, \mathsf{nil})$, and this defines a map $(- \circ \eta)$ giving the forward direction of the equivalence. For the other direction, suppose that $f : A \to G$. We lift this to a map $\bar{f} : \mathsf{List}(A) \to G$ by recursion:

$$\begin{aligned} \bar{f}(\mathsf{nil}) &:\equiv e \\ \bar{f}(\mathsf{cons}(h, t)) &:\equiv f(h) \cdot \bar{f}(t) \end{aligned}$$

To show that this is a monoid homomorphism, we must show

$$\begin{aligned} \bar{f}(\mathsf{nil}) &= e \\ \bar{f}(\ell \cdot \ell') &= \bar{f}(\ell) \cdot \bar{f}(\ell') \end{aligned}$$

The first is a judgemental equality, so we just need to show the second, which we do by induction on ℓ. When $\ell \equiv \mathsf{nil}$ we have

$$\bar{f}(\mathsf{nil} \cdot \ell') \equiv \bar{f}(\ell') = e \cdot \bar{f}(\ell') \equiv \bar{f}(\mathsf{nil}) \cdot \bar{f}(\ell')$$

and when it is a cons,

$$\bar{f}(\mathsf{cons}(h, t) \cdot \ell') = \bar{f}(\mathsf{cons}(h, t \cdot \ell')) = f(h) \cdot \bar{f}(t \cdot \ell') = f(h) \cdot \bar{f}(t) \cdot \bar{f}(\ell') = f(\mathsf{cons}(h, t)) \cdot \bar{f}(\ell')$$

by the induction hypothesis in the third equality. So $\bar{f}$ is a monoid homomorphism.

To show that these are quasi-inverses, suppose that $f : \hom_{\mathrm{Monoid}}(\mathsf{List}(A), G)$. Then we must show that

$$\overline{f \circ \eta} = f$$

which we do by function extensionality and induction. When $\ell \equiv \mathsf{nil}$, we have

$$\overline{f \circ \eta}(\mathsf{nil}) \equiv e = f(\mathsf{nil})$$

Since f is a monoid homomorphism. When $\ell \equiv \mathsf{cons}(h, t)$,

$$\overline{f \circ \eta}(\mathsf{cons}(h, t)) \equiv f(\eta(h)) \cdot \overline{f \circ \eta}(t) = f(\eta(h)) \cdot f(t) = f(\mathsf{cons}(h, \mathsf{nil}) \cdot t) \equiv f(\mathsf{cons}(h, t))$$

So by function extensionality $\overline{f \circ \eta} = f$. We must also show that the proofs that $\overline{f \circ \eta}$ and f are monoid homomorphisms are equal. This turns out to be trivial, however: since monoids are structures on sets, all of the relevant proofs are of equalities in sets, so the structures are mere propositions, and equality of the underlying maps is equivalent to equality of the homomorphisms.

For the other direction, suppose that $f : A \to G$. We show that

$$\bar{f} \circ \eta = f$$

again by function extensionality. Suppose that $a : A$; then

$$\bar{f}(\eta(a)) \equiv \bar{f}(\mathsf{cons}(a, \mathsf{nil})) \equiv f(a) \cdot \bar{f}(\mathsf{nil}) \equiv f(a) \cdot e = f(a)$$

and we're done.

```
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
Class IsMonoidHom {A B : Monoid} (f : A → B) :=
  BuildIsMonoidHom {
      hunit : f (m_unit A) = m_unit B ;
      hmult : ∀ a a' : A, f ((m_mult A) a a') = (m_mult B) (f a) (f a')
    }.
Record MonoidHom (A B : Monoid) :=
  BuildMonoidHom {
      mhom_fun :> A → B ;
      mhom_ismhom :> IsMonoidHom mhom_fun
    }.
Definition homLAG_to_AG (A : Type) (HA : IsHSet A) (G : Monoid) :
  MonoidHom (BuildMonoid (list A) _ _ set_list_is_monoid) G → (A → G)
  := fun f a ⇒ (mhom_fun _ G f) [a].
Definition AG_to_homLAG (A : Type) (HA : IsHSet A) (G : Monoid) :
  (A → G) → MonoidHom (BuildMonoid (list A) _ _ set_list_is_monoid) G.
Proof.
  (*  lift f by recursion *)
  intro f.
  refine (BuildMonoidHom _ G _ _).
  intro l. induction l as [ | h t].
  apply (m_unit G).
  apply ((m_mult _) (f h) IHt).

  apply BuildIsMonoidHom.
  (*  takes the unit to the unit *)
  reflexivity.

  (*  respects multiplication *)
  simpl. intro l. induction l. intro l'. simpl.
  refine (_ @ (m_unitl _)^). reflexivity. apply G.

  intro l'. simpl. refine (_ @ (m_assoc _ _ _)).
  f_ap. apply (m_unit _). apply G.
Defined.
Theorem isprod_ismonoidhom {A B : Monoid} (f : A → B) :
  (f (m_unit A) = m_unit B)
  × (∀ a a', f ((m_mult A) a a') = (m_mult B) (f a) (f a'))
  ≃
  IsMonoidHom f.
Proof.
```

```
  (*  I think this should be a judgemental equality, but it's not *)
  etransitivity {_ : f A = B & ∀ a a' : A, f (A a a') = B (f a) (f a')}.
  refine (equiv_adjointify _ _ _ _); intro z.
    ∃ (fst z). apply (snd z). apply (z.1, z.2).
    apply eta_sigma. apply eta_prod.

  issig (BuildIsMonoidHom A B f) (@hunit A B f) (@hmult A B f).
Defined.

Theorem hprop_dependent (A : Type) (P : A → Type) :
  (∀ a, IsHProp (P a)) → IsHProp (∀ a, P a).
Proof.
  intro HP. apply hprop_allpath. intros p p'.
  apply path_forall; intro a. apply HP.
Defined.

Theorem hprop_ismonoidhom {A B : Monoid} (f : A → B) : IsHProp (IsMonoidHom f).
Proof.
  refine (trunc_equiv' (isprod_ismonoidhom f)).
  apply hprop_prod.
  intros p q. apply B.
  repeat (apply hprop_dependent; intro). intros p q. apply B.
Defined.

Theorem issig_monoidhom (A B : Monoid) :
  {f : A → B & IsMonoidHom f} ≃ MonoidHom A B.
Proof.
  issig (BuildMonoidHom A B) (@mhom_fun A B) (@mhom_ismhom A B).
Defined.

Theorem equiv_path_monoidhom {A B : Monoid} {f g : MonoidHom A B} :
  ((mhom_fun _ _ f) = (mhom_fun _ _ g)) ≃ f = g.
Proof.
  equiv_via ((issig_monoidhom A B)^-1 f = (issig_monoidhom A B)^-1 g).
  refine (@equiv_path_sigma_hprop
              (A → B) IsMonoidHom hprop_ismonoidhom
              ((issig_monoidhom A B)^-1 f) ((issig_monoidhom A B)^-1 g)).
  apply equiv_inverse. apply equiv_ap. refine _.
Defined.

Theorem list_is_free_monoid (A : Type) (HA : IsHSet A) (G : Monoid) :
  MonoidHom (BuildMonoid (list A) _ _ set_list_is_monoid) G ≃ (A → G).
Proof.
  transparent assert (HG : (IsMonoid G G G)). apply G.
  refine (equiv_adjointify (homLAG_to_AG _ _ _) (AG_to_homLAG _ _ _) _ _).
  intro f. apply path_forall; intro a.
  simpl. apply (m_unitr _).

  intro f. apply equiv_path_monoidhom. apply path_forall; intro l.
  induction l as [|h t]; simpl.
  symmetry. apply (@hunit _ _ f). apply f.
  transitivity (G (homLAG_to_AG A HA G f h) (f t)). f_ap.
  unfold homLAG_to_AG. refine (@hmult _ _ f _ [h] t)^. apply f.
Defined.

Local Close Scope list_scope.
```

*Exercise 6.9 (p. 217)** Assuming LEM, construct a family $f : \prod_{(X:\mathcal{U})}(X \to X)$ such that $f_{\mathbf{2}} : \mathbf{2} \to \mathbf{2}$ is the nonidentity automorphism.

*Exercise 6.10 (p. 218)** Show that the map constructed in Lemma 6.3.2 is in fact a quasi-inverse to happly, so that the interval type implies the full function extensionality axiom.

Solution Of course, it's easiest to prove the full function extensionality axiom by referring to Exercise 2.16. But we want to show something more: that this map is an inverse to happly. Let $f g : A \to B$, and suppose that $p : f = g$. Then $\mathsf{happly}(p) : \prod_{(x:A)} f(x) = g(x)$. For all $x : A$ we define a function $\tilde{h} : I \to B$ by

$$
\begin{aligned}
\tilde{h}_x(0_I) &:\equiv f(x), \\
\tilde{h}_x(1_I) &:\equiv g(x), \\
\mathsf{ap}_{\tilde{h}_x}(\mathsf{seg}) &:\equiv \mathsf{happly}(p, x)
\end{aligned}
$$

and we define $q : I \to (A \to B)$ by $q(i) :\equiv \lambda x. \tilde{h}_x(i)$. Thus

$$\mathsf{ap}_q(\mathsf{seg}) \equiv \mathsf{ap}_{\lambda x. \tilde{h}_x}(\mathsf{seg})$$

```
Module Exercise6_10.

Module Interval.

Private Inductive interval : Type :=
  | zero : interval
  | one : interval.

Axiom seg : zero = one.

Definition interval_rect (P : interval → Type)
  (a : P zero) (b : P one) (p : seg # a = b)
  : ∀ x : interval, P x
  := fun x ⇒ (match x return _ → P x with
                | zero ⇒ fun _ ⇒ a
                | one ⇒ fun _ ⇒ b
              end) p.

Axiom interval_rect_beta_seg : ∀ (P : interval → Type)
  (a : P zero) (b : P one) (p : seg # a = b),
  apD (interval_rect P a b p) seg = p.

End Interval.

Definition interval_rectnd (P : Type) (a b : P) (p : a = b)
  : interval → P
  := interval_rect (fun _ ⇒ P) a b (transport_const _ _ @ p).

Definition interval_rectnd_beta_seg (P : Type) (a b : P) (p : a = b)
  : ap (interval_rectnd P a b p) seg = p.
Proof.
  refine (cancelL (transport_const seg a) _ _ _).
  refine ((apD_const (interval_rect (fun _ ⇒ P) a b _) seg)^ @ _).
  refine (interval_rect_beta_seg (fun _ ⇒ P) _ _ _).
Defined.

Definition Lemma6_3_2 {A B : Type} (f g : A → B) : (f ∼ g) → (f = g).
Proof.
```

```
  intro p.
  transparent assert (pt : (∀ x:A, interval → B)).
    intro x. apply (interval_rectnd B (f x) (g x) (p x)).
  transparent assert (q : (interval → (A → B))).
    intro i. apply (fun x:A ⇒ pt x i).
  apply (ap q seg).
Defined.

End Exercise6_10.
```

Exercise 6.11 (p. 218) Prove the universal property of suspension:

$$(\Sigma A \to B) \simeq \left(\sum_{(b_n:B)} \sum_{(b_s:B)} (A \to (b_n = b_s)) \right)$$

Solution To construct an equivalence, suppose that $f : \Sigma A \to B$. Then there are two elements $f(\mathsf{N}), f(\mathsf{S}) : B$ such that there is a map $A \to (f(\mathsf{N}) = f(\mathsf{S}))$; in particular for any element $a : A$ we have the element $\mathsf{merid}(a) : (\mathsf{N} = \mathsf{S})$ which may be pushed forward to give $f(\mathsf{merid}(a)) : f(\mathsf{N}) = f(\mathsf{S})$. For the other direction, suppose that we have elements $b_n, b_s : B$ such that $f : A \to (b_n = b_s)$. Then by suspension recursion we define a function $g : \Sigma A \to B$ such that $g(\mathsf{N}) \equiv b_n$, $g(\mathsf{S}) \equiv b_s$, and $g(\mathsf{merid}(a)) = f(a)$ for all $a : \Sigma A$.

To show that these are quasi-inverses, suppose that $f : \Sigma A \to B$. We then construct the element $(f(\mathsf{N}), f(\mathsf{S}), \lambda a. f(\mathsf{merid}(a)))$ of the codomain, and going back gives a function $g : \Sigma A \to B$ such that $g(\mathsf{N}) \equiv f(\mathsf{N})$, $g(\mathsf{S}) \equiv f(\mathsf{S})$, and $g(\mathsf{merid}(a)) = f(\mathsf{merid}(a))$ for all $a : \Sigma A$. But this just means that g and f have the same recurrence relation, so we're back where we started.

For the other loop, suppose that we have an element (b_n, b_s, f) of the right. Then we get an arrow $g : \Sigma A \to B$ on the left such that $g(\mathsf{N}) = b_n$, $g(\mathsf{S}) = b_s$, and $g(\mathsf{merid}(a)) = f(a)$ for all $a : \Sigma A$. Going back to the right, we have the element (b_n, b_s, f), homotopic to the identity function.

```
Theorem univ_prop_susp {A B : Type} :
  (Susp A → B) ≃ {bn : B & {bs : B & A → (bn = bs)}}.
Proof.
  refine (equiv_adjointify _ _ _ _).
  intro f. ∃ (f North). ∃ (f South). intro a. apply (ap f (merid a)).
  intro w. destruct w as [bn [bs f]]. apply (Susp_rect_nd bn bs f).

  intro w. destruct w as [bn [bs f]].
  apply path_sigma_uncurried. ∃ 1.
  apply path_sigma_uncurried. ∃ 1.
  apply path_forall; intro a. simpl.
  apply Susp_comp_nd_merid.

  intro f. apply path_forall.
  refine (Susp_rect _ 1 1 _).
  intro a.
  refine ((trans_paths _ _ _ _ _ _ _ _) @ _).
  apply moveR_pM.
  refine ((concat_p1 _) @ _). refine (_ @ (concat_1p _)^). apply inverse2.
  refine ((Susp_comp_nd_merid _) @ _).
  reflexivity.
Defined.
```

*__Exercise 6.12 (p. 218)__ Show that $\mathbb{Z} \simeq \mathbb{N} + \mathbf{1} + \mathbb{N}$. Show that if we were to define $\mathbb{Z}$ as $\mathbb{N} + \mathbf{1} + \mathbb{N}$, then we could obtain Lemma 6.10.12 with judgmental computation rules.

Solution Let $\mathbb{Z} :\equiv \sum_{(x:\mathbb{N}\times\mathbb{N})}(r(x) = x)$, where

$$r(a,b) = \begin{cases} (a-b,0) & \text{if } a \geq b \\ (0,b-a) & \text{otherwise} \end{cases}$$

To define the forward direction, let $((a,b),p) : \mathbb{Z}$. If $a = b$, then produce $\star$. Otherwise, if $a > b$, produce $\mathsf{pred}(a-b)$ in the right copy of $\mathbb{N}$. Otherwise (i.e., when $a < b$), produce $\mathsf{pred}(b-a)$ in the left copy of $\mathbb{N}$. To go the other way, we have three cases. When n is in the left, send it to $(0, \mathsf{succ}(n))$, along with the appropriate proof. When $n \equiv \star$, produce $(0,0)$. When n is in the right, send it to $(\mathsf{succ}(n), 0)$. Clearly, these two constructions are quasi-inverses, since $\mathsf{succ}(\mathsf{pred}(n)) = n$ for all $n \neq 0$.

```
Module Exercise6_12.

Fixpoint monus n m :=
  match m with
    | O ⇒ n
    | S m' ⇒ pred (monus n m')
  end.

Lemma monus_O_n : ∀ n, monus O n = O.
Proof. induction n. reflexivity. simpl. change O with (pred O). f_ap. Defined.

Lemma n_le_Sn : ∀ n, le n (S n).
Proof. intro n. ∃ (S O). apply (plus_1_r n)^. Defined.

Lemma monus_eq_O__n_le_m : ∀ n m, (monus n m = O) → (le n m).
Admitted.

Lemma monus_self : ∀ n, monus n n = O.
Admitted.

Definition n_le_m__Sn_le_Sm : ∀ (n m : nat), (le n m) → (le (S n) (S m))
  := fun n m H ⇒ (H.1; ap S H.2).

Lemma order_partitions : ∀ (n m : nat), (le n m) + (lt m n).
Proof.
  induction n.
    intro m. left. ∃ m. reflexivity.
  induction m.
    right. ∃ n. reflexivity.
    destruct IHm.
      left. destruct l. ∃ (S x).
      simpl. apply (ap S). apply ((plus_n_Sm _ _)^ @ p).
      destruct (IHn m).
        left. apply n_le_m__Sn_le_Sm. apply l0.
        right. destruct l0. ∃ x. simpl. apply (ap S). apply p.
Defined.

Definition r : nat × nat → nat × nat.
  intro z. destruct z as [a b].
  destruct (order_partitions b a).
  apply (monus a b, O).
  apply (O, monus b a).
Defined.

Definition int := {x : nat × nat & r x = x}.

Definition int_to_nat_1_nat : int → (nat + Unit + nat).
  intro z. destruct z as [[a b] p]. destruct (nat_eq_decidable a b).
```

```
  left. right. apply tt.
  destruct (order_partitions b a).
  right. apply (pred (monus a b)).
  left. left. apply (pred (monus b a)).
Defined.

Definition nat_1_nat_to_int : (nat + Unit + nat) → int :=
  fun z ⇒
    match z with
      | inl a ⇒ match a with
                    | inl n ⇒ ((O, S n) ; 1)
                    | inr _ ⇒ ((O, O) ; 1)
                  end
      | inr n ⇒ ((S n, O) ; 1)
    end.

Lemma lt_le : ∀ n m, (lt n m) → (le n m).
Proof.
  intros n m p. destruct p as [k p]. ∃ (S k).
  apply p.
Defined.

Lemma hset_prod : ∀ A, IsHSet A → ∀ B, IsHSet B → IsHSet (A × B).
Proof.
  intros A HA B HB.
  intros z z'. apply hprop_allpath. apply allpath_hprop.
Defined.

Theorem hset_nat : IsHSet nat.
Proof.
  apply hset_decidable. intros n.
  induction n; intro m; destruct m.
    left. reflexivity.
    right. intro p. apply equiv_path_nat in p. contradiction.
    right. intro p. apply equiv_path_nat in p. contradiction.
    destruct (IHn m).
      left. apply (ap S). apply p.
      right. intro p. apply S_inj in p. contradiction.
Defined.

Theorem ex6_12 : int ≃ (nat + Unit + nat).
Proof.
  refine (equiv_adjointify int_to_nat_1_nat nat_1_nat_to_int _ _).
  intro z. destruct z as [[n | n] | n].
  reflexivity. simpl. repeat f_ap. apply contr_unit. reflexivity.
  intro z. destruct z as [[a b] p].
  apply path_sigma_uncurried.
  assert (
    (nat_1_nat_to_int (int_to_nat_1_nat ((a, b) ; p))) .1 = ((a, b) ; p) .1
  ) as H.
  unfold nat_1_nat_to_int, int_to_nat_1_nat.
  destruct (nat_eq_decidable a b).
  unfold r in p. simpl. destruct (order_partitions b a); refine (_ @ p);
    apply path_prod.
    assert (b = O). apply (ap snd p)^.
```

```
    assert (a = O). apply (p0 @ X).
    simpl. transitivity (monus a 0). simpl. apply X0^. f_ap. apply X^.
    reflexivity.
    reflexivity.
    assert (a = O). apply (ap fst p)^. assert (b = O). apply (p0^ @ X).
    simpl. transitivity (monus b 0). simpl. apply X0^. f_ap. apply X^.
  unfold r in p. simpl. destruct (order_partitions b a); refine (_ @ p);
    apply path_prod.
    simpl. refine ((S_pred_inv _ _) @ _).
    intro H. apply monus_eq_O__n_le_m in H.
    apply le_antisymmetric in H. symmetry in H. apply n in H. contradiction.
    apply l. reflexivity. reflexivity. reflexivity.
    simpl. refine ((S_pred_inv _ _) @ _).
    intro H.
    assert (a = b). refine ((ap fst p)^ @ H^ @ (ap snd p)). apply n in X.
    contradiction. reflexivity. simpl in *.
    ∃ H.
    assert (IsHSet (nat × nat)) as Hn. apply hset_prod; apply hset_nat.
    apply set_path2.
Defined.
End Exercise6_12.
```

7 Homotopy n-types

***Exercise 7.1 (p. 250)** (i) Use Theorem 7.2.2 to show that if $\|A\| \to A$ for every type A, then every type is a set. (ii) Show that if every surjective function (purely) splits, i.e. if $\prod_{(b:B)} \left\| \mathsf{fib}_f(b) \right\| \to \prod_{(b:B)} \mathsf{fib}_f(b)$ for every $f : A \to B$, then every type is a set.

Solution (i) Suppose that $f : \prod_{(A:\mathcal{U})} \|A\| \to A$, and let X be some type. To apply Theorem 7.2.2, we need a reflexive mere relation on X that implies identity. Let $R(x,y) :\equiv \|x = y\|$, which is clearly a reflexive mere relation. Moreoever, it implies the identity, since for any $x, y : X$ we have

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

So by Theorem 7.2.2, X is a set.

(ii) Again, let X be a typeV and R the relation $R(x,y) :\equiv \|x = y\|$. Suppose that $p : R(x,y)$. Then for all $a : A$, there merely exists an $a' : A$ such that $a' = a$. But since every surjective function purely splits, that means that for every $a : A$ there exists some $a' : A$ such that $a' = a$. So, in particular,

***Exercise 7.2 (p. 250)**

***Exercise 7.3 (p. 250)**

***Exercise 7.4 (p. 250)**

***Exercise 7.5 (p. 250)**

***Exercise 7.6 (p. 250)**

***Exercise 7.7 (p. 250)**

***Exercise 7.8 (p. 250)**

***Exercise 7.9 (p. 251)**

***Exercise 7.10 (p. 251)**

***Exercise 7.11 (p. 251)**

Exercise 7.12 (p. 251) Show that $X \mapsto (\neg\neg X)$ is a modality.

Solution We must show that there are

(i) functions $\eta_A^{\neg\neg} : A \to \neg\neg A$ for every type A,
(ii) for every $A : \mathcal{U}$ and every family $B : \neg\neg A \to \mathcal{U}$, a function

$$\mathsf{ind}_{\neg\neg} : \left(\prod_{a:A} \neg\neg(B(\eta_A^{\neg\neg}(a)))\right) \to \prod_{z:\neg\neg A} \neg\neg(B(z))$$

(iii) A path $\mathsf{ind}_{\neg\neg}(f)(\eta_A^{\neg\neg}(a)) = f(a)$ for each $f : \prod_{(a:A)} \neg\neg(B(\eta_A^{\neg\neg}(a)))$, and
(iv) For any $z, z' : \neg\neg A$, the function $\eta_{z=z'}^{\neg\neg} : (z = z') \to \neg\neg(z = z')$ is an equivalence.

Some of this has already been done. The functions $\eta_A^{\neg\neg}$ were given in Exercise 1.12; recall:

$$\eta_A^{\neg\neg} :\equiv \lambda a. \lambda f. f(a)$$

Now suppose that we have an element f of the antecedent of $\mathsf{ind}_{\neg\neg}$, and let $z : \neg\neg A$. Suppose furthermore that $g : \neg B(z)$; we derive a contradiction. It suffices to construct an element of $\neg A$, for then we can apply z to obtain an element of $\mathbf{0}$. So suppose that $a : A$. Then

$$f(a) : \neg\neg B(\eta_A^{\neg\neg}(a))$$

but $\neg X$ is a mere proposition for any type X, so $z = \eta_A^{\neg\neg}(a)$, and transporting gives an element of $\neg\neg B(z)$. Applying this to g gives a contradiction.

For (iii), suppose that $a : A$ and $f : \prod_{(a:A)} \neg\neg(B(\eta_A^{\neg\neg}(a)))$. Then $f(a) : \neg\neg(B(\eta_A^{\neg\neg}(a)))$, making this type contractible, giving a canonical path $\mathsf{ind}_{\neg\neg}(f)(\eta_A^{\neg\neg}(a)) = f(a)$.

Finally, for (iv), let $z, z' : \neg\neg A$. Since $\neg\neg A$ is a mere proposition, so is $z = z'$. $\neg\neg(z = z')$ is also a mere proposition, so if there is an arrow $\neg\neg(z = z') \to (z = z')$, then it is automatically a quasi-inverse to $\eta_{z=z'}^{\neg\neg}$. Such an arrow is immediate from the fact that $\neg\neg A$ is a mere proposition.

```
Definition eta_nn (A : Type) : A → ¬ ¬ A := fun a f ⇒ f a.

Lemma hprop_arrow (A B : Type) : IsHProp B → IsHProp (A → B).
Proof.
  intro HB.
  apply hprop_allpath. intros f g. apply path_forall. intro a. apply HB.
Defined.

Lemma hprop_neg {A : Type} : IsHProp (¬ A).
Proof.
  apply hprop_arrow. apply hprop_Empty.
Defined.

Definition ind_nn (A : Type) (B : ¬ ¬ A → Type) :
```

```
  (∀ a : A, ¬ ¬ (B (eta_nn A a))) → ∀ z : ¬ ¬ A, ¬ ¬ (B z).
Proof.
  intros f z. intro g.
  apply z. intro a.
  apply ((transport (fun x ⇒ ¬ ¬ (B x))
                    (@allpath_hprop _ hprop_neg _ _)
                    (f a)) g).
Defined.

Definition nn_modality_iii (A : Type) (B : ¬ ¬ A → Type)
  (a : A) (f : ∀ a, ¬ ¬ (B (eta_nn A a))) :
    ind_nn A B f (eta_nn A a) = f a.
Proof.
  apply allpath_hprop.
Defined.

Lemma isequiv_hprop (P Q : Type) (HP : IsHProp P) (HQ : IsHProp Q)
      (f : P → Q) :
  (Q → P) → IsEquiv f.
Proof.
  intro g.
  refine (isequiv_adjointify f g _ _).
  intro q. apply allpath_hprop.
  intro p. apply allpath_hprop.
Defined.

Lemma hprop_hprop_path (A : Type) (HA : IsHProp A) (x y : A) : IsHProp (x = y).
Proof.
  apply hprop_allpath.
  apply set_path2.
Defined.

Definition nn_modality_iv (A : Type) (z z' : ¬ ¬ A) : IsEquiv (eta_nn (z = z')).
Proof.
  apply isequiv_hprop.
  apply hprop_hprop_path. apply hprop_neg. apply hprop_neg.
  intro f. apply hprop_neg.
Defined.
```

***Exercise 7.13 (p. 251)**

***Exercise 7.14 (p. 251)**