Contents

Chapter 1

Library Functions

`Definition` idmap A := `fun` x : $A \Rightarrow x$.

`Definition` compose $\{A\ B\ C\}$ $(g : B \rightarrow C)$ $(f : A \rightarrow B)$ $(x : A)$:= $g\ (f\ x)$.

Notation "g 'o' f" := (compose $g\ f$) (*left associativity*, `at` *level* 37).

Chapter 2

Library Paths

Basic homotopy-theoretic approach to paths.

```
Require Export Functions.
```

For compatibility with Coq 8.2. `Unset` *Automatic Introduction.*

```
Inductive paths {A} : A → A → Type := idpath : ∀ x, paths x x.
```

We introduce notation $x = y$ for the space **paths** x y of paths from x to y. We can then write $p : x = y$ to indicate that p is a path from x to y.

```
Notation "x == y" := (paths x y) (at level 70).
```

The `Hint Resolve idpath@` line below means that Coq's `auto` tactic will automatically perform `apply idpath` if that leads to a successful solution of the current goal. For example if we ask it to construct a path $x = x$, `auto` will find the identity path `idpath` x, thanks to the `Hint Resolve`.

In general we should declare `Hint Resolve` on those theorems which are not very complicated but get used often to finish off proofs. Notice how we use the non-implicit version `idpath@` (if we try `Hint Resolve idpath` Coq complains that it cannot guess the value of the implicit argument A).

```
Hint Resolve @idpath.
```

The following automated tactic applies induction on paths and then idpath. It can handle many easy statements.

```
Ltac path_induction :=
  intros; repeat progress (
    match goal with
      | [ p : _ = _ ⊢ _ ] ⇒ induction p
      | _ ⇒ idtac
    end
  ); auto.
```

You can read the tactic definition as follows. We first perform `intros` to move hypotheses into the context. Then we repeat while there is still progress: if there is a path p in the context, apply induction to it, otherwise perform the *idtac* which does nothing (and so no progress is made and we stop). After that, we perform an `auto`.

The notation [... $\vdash$ ...] is a pattern for contexts. To the left of the symbol $\vdash$ we list hypotheses and to the right the goal. The underscore means "anything".

In summary *path_induction* performs as many inductions on paths as it can, then it uses `auto`.

We now define the basic operations on paths, starting with concatenation.

```
Definition concat {A} {x y z : A} : (x = y) → (y = z) → (x = z).
Proof.
  intros A x y z p q.
  induction p.
  induction q.
  apply idpath.
Defined.
```

The concatenation of paths p and q is denoted as p @ q.

Notation "p @ q" := (concat p q) (at *level* 60).

A definition like `concat` can be used in two ways. The first and obvious way is as an operation which concatenates together two paths. The second use is a proof tactic when we want to construct a path $x = z$ as a concatenation of paths $x = y = z$. This is done with `apply concat@`, see examples below. We will actually define a tactic *path_via* which uses `concat` but is much smarter than just the direct application `apply concat@`.

Paths can be reversed.

```
Definition opposite {A} {x y : A} : (x = y) → (y = x).
Proof.
  intros A x y p.
  induction p.
  apply idpath.
Defined.
```

Notation for the opposite of a path p is ! p.

Notation "! p" := (opposite p) (at *level* 50).

Next we give names to the basic properties of concatenation of paths. Note that all statements are "up to a higher path", e.g., the composition of p and the identity path is not equal to p but only connected to it with a path.

The following lemmas say that up to higher paths, the paths form a 1-groupoid.

```
Lemma idpath_left_unit A (x y : A) (p : x = y) : idpath x @ p = p.
Proof.
  path_induction.
Defined.
```

```
Lemma idpath_right_unit A (x y : A) (p : x = y) : (p @ idpath y) = p.
```

```
Proof.
  path_induction.
Defined.

Lemma opposite_right_inverse A (x y : A) (p : x = y) : (p @ !p) = idpath x.
Proof.
 path_induction.
Defined.

Lemma opposite_left_inverse A (x y : A) (p : x = y) : (!p @ p) = idpath y.
Proof.
  path_induction.
Defined.

Lemma opposite_concat A (x y z : A) (p : x = y) (q : y = z) : !(p @ q) = !q @ !p.
Proof.
  path_induction.
Defined.

Lemma opposite_idpath A (x : A) : !(idpath x) = idpath x.
Proof.
  path_induction.
Defined.

Lemma opposite_opposite A (x y : A) (p : x = y) : !(! p) = p.
Proof.
  path_induction.
Defined.

Lemma concat_associativity A (w x y z : A) (p : w = x) (q : x = y) (r : y = z) :
  (p @ q) @ r = p @ (q @ r).
Proof.
  path_induction.
Defined.
```

Now we move on to the 2-groupoidal structure of a type. Concatenation of 2-paths along 1-paths is just ordinary concatenation in a path type, but we need a new name and notation for concatenation of 2-paths along points.

```
Definition concat2 {A} {x y z : A} {p p' : x = y} {q q' : y = z} :
  (p = p') -> (q = q') -> (p @ q = p' @ q').
Proof.
  path_induction.
Defined.

Notation "p @@ q" := (concat2 p q) (at level 60).
```

We also have whiskering operations.

```
Definition whisker_right {A} {x y z : A} {p p' : x = y} (q : y = z) :
```

```
  (p = p') → (p @ q = p' @ q).
Proof.
  path_induction.
Defined.

Definition whisker_left {A} {x y z : A} {q q' : y = z} (p : x = y) :
  (q = q') → (p @ q = p @ q').
Proof.
  path_induction.
Defined.

Definition whisker_right_toid {A} {x y : A} {p : x = x} (q : x = y) :
  (p = idpath x) → (p @ q = q).
Proof.
  intros A x y p q a.
  apply @concat with (y := idpath x @ q).
  apply whisker_right. assumption.
  apply idpath_left_unit.
Defined.

Definition whisker_right_fromid {A} {x y : A} {p : x = x} (q : x = y) :
  (idpath x = p) → (q = p @ q).
Proof.
  intros A x y p q a.
  apply @concat with (y := idpath x @ q).
  apply opposite, idpath_left_unit.
  apply whisker_right. assumption.
Defined.

Definition whisker_left_toid {A} {x y : A} {p : y = y} (q : x = y) :
  (p = idpath y) → (q @ p = q).
Proof.
  intros A x y p q a.
  apply @concat with (y := q @ idpath y).
  apply whisker_left. assumption.
  apply idpath_right_unit.
Defined.

Definition whisker_left_fromid {A} {x y : A} {p : y = y} (q : x = y) :
  (idpath y = p) → (q = q @ p).
Proof.
  intros A x y p q a.
  apply @concat with (y := q @ idpath y).
  apply opposite, idpath_right_unit.
  apply whisker_left. assumption.
Defined.
```

The interchange law for whiskering.

```
Definition whisker_interchange A (x y z : A) (p p' : x = y) (q q' : y = z)
  (a : p = p') (b : q = q') :
  (whisker_right q a) @ (whisker_left p' b) = (whisker_left p b) @ (whisker_right q' a).
Proof.
  path_induction.
Defined.
```

The interchange law for concatenation.

```
Definition concat2_interchange A (x y z : A) (p p' p'' : x = y) (q q' q'' : y = z)
  (a : p = p') (b : p' = p'') (c : q = q') (d : q' = q'') :
  (a @@ c) @ (b @@ d) = (a @ b) @@ (c @ d).
Proof.
  path_induction.
Defined.
```

Taking opposites of 1-paths is functorial on 2-paths.

```
Definition opposite2 A {x y : A} (p q : x = y) (a : p = q) : (!p = !q).
Proof.
  path_induction.
Defined.
```

Now we consider the application of functions to paths.

A path $p : x = y$ in a space A is mapped by $f : A \to B$ to a path map $f\ p : f\ x = f\ y$ in B.

```
Lemma map {A B} {x y : A} (f : A -> B) : (x = y) -> (f x = f y).
Proof.
  path_induction.
Defined.
```

The next two lemmas state that map $f\ p$ is "functorial" in the path p.

```
Lemma idpath_map A B (x : A) (f : A -> B) : map f (idpath x) = idpath (f x).
Proof.
  path_induction.
Defined.

Lemma concat_map A B (x y z : A) (f : A -> B) (p : x = y) (q : y = z) :
  map f (p @ q) = (map f p) @ (map f q).
Proof.
  path_induction.
Defined.

Lemma opposite_map A B (f : A -> B) (x y : A) (p : x = y) :
  map f (! p) = ! map f p.
Proof.
```

```
  path_induction.
Defined.
```

It is also the case that map f p is functorial in f.

```
Lemma idmap_map A (x y : A) (p : x = y) : map (idmap A) p = p.
Proof.
  path_induction.
Defined.

Lemma compose_map A B C (f : A → B) (g : B → C) (x y : A) (p : x = y) :
  map (g ∘ f) p = map g (map f p).
Proof.
  path_induction.
Defined.
```

We can also map paths between paths.

```
Definition map2 {A B} {x y : A} {p q : x = y} (f : A → B) :
  p = q → (map f p = map f q)
  := map (map f).
```

The type of "homotopies" between two functions f and g is $\forall\ x, f\ x = g\ x$. These can be derived from "paths" between functions $f = g$; the converse is function extensionality.

```
Definition happly {A B} {f g : A → B} : (f = g) → (∀ x, f x = g x) :=
  fun p x ⇒ map (fun h ⇒ h x) p.
```

Similarly, happly for dependent functions.

```
Definition happly_dep {A} {P : A → Type} {f g : ∀ x, P x} :
  (f = g) → (∀ x, f x = g x) :=
  fun p x ⇒ map (fun h ⇒ h x) p.
```

We declare some more `Hint Resolve` hints, now in the "hint database" *path_hints*. In general various hints (resolve, rewrite, unfold hints) can be grouped into "databases". This is necessary as sometimes different kinds of hints cannot be mixed, for example because they would cause a combinatorial explosion or rewriting cycles.

A specific `Hint Resolve` database *db* can be used with `auto with` *db*.

```
Hint Resolve
  @idpath @opposite
  idpath_left_unit idpath_right_unit
  opposite_right_inverse opposite_left_inverse
  opposite_concat opposite_idpath opposite_opposite
  @concat2
  @whisker_right @whisker_left
  @whisker_right_toid @whisker_right_fromid
  @whisker_left_toid @whisker_left_fromid
  opposite2
```

```
  @map idpath_map concat_map idmap_map compose_map opposite_map
  @map2
 : path_hints.
```

We can add more hints to the database later.

For some reason, apply happly and apply happly_dep often seem to fail unification. This tactic does the work that I think they should be doing.

```
Ltac apply_happly :=
  match goal with
    | ⊢ ?f' ?x = ?g' ?x ⇒
      first [
          apply @happly with (f := f') (g := g')
        | apply @happly_dep with (f := f') (g := g')
      ]
  end.
```

The following tactic is intended to be applied when we want to find a path between two expressions which are largely the same, but differ in the value of some subexpression. Therefore, it does its best to "peel off" all the parts of both sides that are the same, repeatedly, until only the "core" bit of difference is left. Then it performs an auto using the *path_hints* database.

```
Ltac path_simplify :=
  repeat progress first [
      apply whisker_left
    | apply whisker_right
    | apply @map
    ]; auto with path_hints.
```

The following variant allows the caller to supply an additional lemma to be tried (for instance, if the caller expects the core difference to be resolvable by using a particular lemma).

```
Ltac path_simplify' lem :=
  repeat progress first [
      apply whisker_left
    | apply whisker_right
    | apply @map
    | apply lem
    | apply opposite; apply lem
    ]; auto with path_hints.
```

These tactics are used to construct a path $a = b$ as a composition of paths $a = x$ and $x = b$. They then apply *path_simplify* to both paths, along with possibly an additional lemma supplied by the user.

```
Ltac path_via mid :=
```

```
  apply @concat with (y := mid); path_simplify.
Ltac path_using mid lem :=
  apply @concat with (y := mid); path_simplify' lem.
```

This variant does not call path_simplify.

```
Ltac path_via' mid :=
  apply @concat with (y := mid).
```

Here are some tactics for reassociating concatenations. The tactic *associate_right* associates both source and target of the goal all the way to the right, and dually for *associate_left*.

```
Ltac associate_right_in s :=
  match s with
    context cxt [ (?a @ ?b) @ ?c ] ⇒
    let mid := context cxt[a @ (b @ c)] in
      path_using mid concat_associativity
  end.
Ltac associate_right :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ associate_right_in s | associate_right_in t ]
    end
  ).
Ltac associate_left_in s :=
  match s with
    context cxt [ ?a @ (?b @ ?c) ] ⇒
    let mid := context cxt[(a @ b) @ c] in
      path_using mid concat_associativity
  end.
Ltac associate_left :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ associate_left_in s | associate_left_in t ]
    end
  ).
```

This tactic unwhiskers by paths on both sides, reassociating as necessary.

```
Ltac unwhisker :=
  associate_left;
  repeat progress apply whisker_right;
  associate_right;
  repeat progress apply whisker_left.
```

Here are some tactics for eliminating identities. The tactic *cancel_units* tries to remove all identity paths and functions from both source and target of the goal.

```
Ltac cancel_units_in s :=
  match s with
    | context cxt [ idpath ?a @ ?p ] ⇒
      let mid := context cxt[p] in path_using mid idpath_left_unit
    | context cxt [ ?p @ idpath ?a ] ⇒
      let mid := context cxt[p] in path_using mid idpath_right_unit
    | context cxt [ map ?f (idpath ?x) ] ⇒
      let mid := context cxt[idpath (f x)] in path_using mid idpath_map
    | context cxt [ map (idmap _) ?p ] ⇒
      let mid := context cxt[p] in path_using mid idmap_map
    | context cxt [ ! (idpath ?a) ] ⇒
      let mid := context cxt[idpath a] in path_using mid opposite_idpath
  end.

Ltac cancel_units :=
  repeat (
    match goal with
      ⊢ ?s = ?t ⇒ first [ cancel_units_in s | cancel_units_in t ]
    end
  ).
```

And some tactics for eliminating matched pairs of opposites.

This is an auxiliary tactic which performs one step of a reassociation of s (which is the source or target of a path) so as to get $!p$ to be closer to being concatenated on the left with something irreducible. If there is more than one copy of $!p$ in s, then this tactic finds the first one which is concatenated on the left with anything (irreducible or not), or if there is no such occurrence of $!p$, then finds the first one overall. If this $!p$ is already concatenated on the left with something irreducible, then if that something is a p, it cancels them. If that something is not a p, then it fails.

```
Ltac partly_cancel_left_opposite_of_in p s :=
  match s with
    | context cxt [ @concat _ ?trg _ _ (!p) p ] ⇒
      let mid := context cxt[ idpath trg ] in path_using mid opposite_left_inverse
    | context cxt [ !p @ (?a @ ?b) ] ⇒
      let mid := context cxt[ (!p @ a) @ b ] in path_using mid concat_associativity
    | context cxt [ !p @ _ ] ⇒ fail 1
    | context cxt [ (?a @ !p) @ ?b ] ⇒
      let mid := context cxt[ a @ (!p @ b) ] in path_using mid concat_associativity
    | context cxt [ ?a @ (?b @ !p) ] ⇒
      let mid := context cxt[ (a @ b) @ !p ] in path_using mid concat_associativity
  end;
  cancel_units.
```

This tactic simply calls the previous one for the source and the target, repeatedly, until it can no longer make progress. Ltac *cancel_left_opposite_of p* :=

```
repeat progress (
  match goal with
    ⊢ ?s = ?t ⇒ first [
        partly_cancel_left_opposite_of_in p s
      | partly_cancel_left_opposite_of_in p t
    ]
  end
).
```

Now the same thing on the right

```
Ltac partly_cancel_right_opposite_of_in p s :=
  match s with
    | context cxt [ @concat _ ?src _ _ p (!p) ] ⇒
      let mid := context cxt[ idpath src ] in path_using mid opposite_right_inverse
    | context cxt [ (?a @ ?b) @ !p ] ⇒
      let mid := context cxt[ a @ (b @ !p) ] in path_using mid concat_associativity
    | context cxt [ _ @ !p ] ⇒ fail 1
    | context cxt [ ?a @ (!p @ ?b) ] ⇒
      let mid := context cxt[ (a @ !p) @ b ] in path_using mid concat_associativity
    | context cxt [ (!p @ ?a) @ ?b ] ⇒
      let mid := context cxt[ !p @ (a @ b) ] in path_using mid concat_associativity
  end;
  cancel_units.
```

```
Ltac cancel_right_opposite_of p :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [
          partly_cancel_right_opposite_of_in p s
        | partly_cancel_right_opposite_of_in p t
      ]
    end
  ).
```

This tactic tries to cancel $!p$ on both the left and the right. Ltac *cancel_opposite_of p* :=

```
  cancel_left_opposite_of p;
  cancel_right_opposite_of p.
```

This tactic looks in s for an opposite of anything, and for the first one it finds, it tries to cancel it on both sides. Ltac *cancel_opposites_in s* :=

```
  match s with
    context cxt [ !(?p) ] ⇒ cancel_opposite_of p
  end.
```

Finally, this tactic repeats the previous one as long as it gets us somewhere. This is most often the easiest of these tactics to call in an interactive proof.

This tactic is not the be-all and end-all of opposite-canceling, however; it only works until it runs into an opposite that it can't cancel. It can get stymied by something like $!p$ @ $!q$ @ q, which should simplify to $!p$, but the tactic simply tries to cancel $!p$, makes no progress, and stops. In such a situation one must call *cancel_opposite_of q* directly (or figure out how to write a smarter tactic!).

```
Ltac cancel_opposites :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ cancel_opposites_in s | cancel_opposites_in t ]
    end
  ).
```

Now we have a sequence of fairly boring tactics, each of which corresponds to a simple lemma. Each of these tactics repeatedly looks for occurrences, in either the source or target of the goal, of something whose form can be changed by the lemma in question, then calls *path_using* to change it.

For each lemma the basic tactic is called *do_LEMMA*. If the lemma can sensibly be applied in two directions, there is also an *undo_LEMMA* tactic.

Tactics for opposite_opposite

```
Ltac do_opposite_opposite_in s :=
  match s with
    | context cxt [ ! (! ?p) ] ⇒
      let mid := context cxt [ p ] in path_using mid opposite_opposite
  end.
```

```
Ltac do_opposite_opposite :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ do_opposite_opposite_in s | do_opposite_opposite_in t ]
    end
  ).
```

Tactics for opposite_map.

```
Ltac apply_opposite_map :=
  match goal with
    | ⊢ map ?f' (! ?p') = ! map ?f' ?p' ⇒
      apply opposite_map with (f := f') (p := p')
    | ⊢ ! map ?f' ?p' = map ?f' (! ?p') ⇒
      apply opposite, opposite_map with (f := f') (p := p')
  end.
```

```
Ltac do_opposite_map_in s :=
```

```
  match s with
    | context cxt [ map ?f (! ?p) ] ⇒
      let mid := context cxt [ ! map f p ] in path_using mid opposite_map
  end.

Ltac do_opposite_map :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ do_opposite_map_in s | do_opposite_map_in t ]
    end
  ); do_opposite_opposite.

Ltac undo_opposite_map_in s :=
  match s with
    | context cxt [ ! (map ?f ?p) ] ⇒
      let mid := context cxt [ map f (! p) ] in path_using mid opposite_map
  end.

Ltac undo_opposite_map :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ undo_opposite_map_in s | undo_opposite_map_in t ]
    end
  ); do_opposite_opposite.
```

Tactics for opposite_concat.

```
Ltac do_opposite_concat_in s :=
  match s with
    | context cxt [ (! ?p) @ (! ?q) ] ⇒
      let mid := context cxt [ ! (q @ p) ] in path_using mid opposite_concat
  end.

Ltac do_opposite_concat :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ do_opposite_concat_in s | do_opposite_concat_in t ]
    end
  ); do_opposite_opposite.

Ltac undo_opposite_concat_in s :=
  match s with
    | context cxt [ ! (?q @ ?p) ] ⇒
      let mid := context cxt [ (! p) @ (! q) ] in path_using mid opposite_concat
  end.

Ltac undo_opposite_concat :=
  repeat progress (
```

```
    match goal with
      ⊢ ?s = ?t ⇒ first [ undo_opposite_concat_in s | undo_opposite_concat_in t ]
    end
  ); do_opposite_opposite.
```

Tactics for compose_map. As with happly, apply compose_map often fail to unify, so we define a separate tactic.

```
Ltac apply_compose_map :=
  match goal with
    | ⊢ map (?g' ∘ ?f') ?p' = map ?g' (map ?f' ?p') ⇒
      apply compose_map with (g := g') (f := f') (p := p')
    | ⊢ map ?g' (map ?f' ?p') = map (?g' ∘ ?f') ?p' ⇒
      apply opposite; apply compose_map with (g := g') (f := f') (p := p')
  end.

Ltac do_compose_map_in s :=
  match s with
    | context cxt [ map (?f ∘ ?g) ?p ] ⇒
      let mid := context cxt [ map f (map g p) ] in
        path_via mid; try apply_compose_map
  end.

Ltac do_compose_map :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ do_compose_map_in s | do_compose_map_in t ]
    end
  ).

Ltac undo_compose_map_in s :=
  match s with
    | context cxt [ map ?f (map ?g ?p) ] ⇒
      let mid := context cxt [ map (f ∘ g) p ] in
        path_via mid; try apply_compose_map
  end.

Ltac undo_compose_map :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ undo_compose_map_in s | undo_compose_map_in t ]
    end
  ).
```

Tactics for concat_map.

```
Ltac do_concat_map_in s :=
  match s with
```

```
    | context cxt [ map ?f (?p @ ?q) ] ⇒
      let mid := context cxt [ map f p @ map f q ] in path_using mid concat_map
  end.

Ltac do_concat_map :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ do_concat_map_in s | do_concat_map_in t ]
    end
  ).

Ltac undo_concat_map_in s :=
  match s with
    | context cxt [ map ?f ?p @ map ?f ?q ] ⇒
      let mid := context cxt [ map f (p @ q) ] in path_using mid concat_map
  end.

Ltac undo_concat_map :=
  repeat progress (
    match goal with
      ⊢ ?s = ?t ⇒ first [ undo_concat_map_in s | undo_concat_map_in t ]
    end
  ).
```

Now we return to proving lemmas about paths. We show that homotopies are natural with respect to paths in the domain.

```
Lemma homotopy_naturality A B (f g : A → B) (p : ∀ x, f x = g x) (x y : A) (q : x = y) :
  map f q @ p y = p x @ map g q.
Proof.
  induction q.
  cancel_units.
Defined.

Hint Resolve homotopy_naturality : path_hints.

Lemma homotopy_naturality_toid A (f : A → A) (p : ∀ x, f x = x) (x y : A) (q : x = y) :
  map f q @ p y = p x @ q.
Proof.
  induction q.
  cancel_units.
Defined.

Hint Resolve homotopy_naturality_toid : path_hints.

Lemma homotopy_naturality_fromid A (f : A → A) (p : ∀ x, x = f x) (x y : A) (q : x = y) :
  q @ p y = p x @ map f q.
Proof.
  induction q.
```

cancel_units.
`Defined.`

`Hint Resolve` homotopy_naturality_fromid : *path_hints*.

Cancellability of concatenation on both sides.

`Lemma` concat_cancel_right A $(x\ y\ z : A)$ $(p\ q : x = y)$ $(r : y = z)$: $(p$ `@` $r = q$ `@` $r) \rightarrow (p = q)$.
`Proof.`
  `intros` $A\ x\ y\ z\ p\ q\ r$.
  `intro` a.
  `induction` p.
  `induction` r.
  path_via (q `@` idpath x).
`Defined.`

`Lemma` concat_cancel_left A $(x\ y\ z : A)$ $(p : x = y)$ $(q\ r : y = z)$: $(p$ `@` $q = p$ `@` $r) \rightarrow (q = r)$.
`Proof.`
  `intros` $A\ x\ y\ z\ p\ q\ r$.
  `intro` a.
  `induction` p.
  `induction` r.
  path_via (idpath x `@` q).
`Defined.`

If a function is homotopic to the identity, then that homotopy makes it a "well-pointed" endofunctor in the following sense.

`Lemma` htoid_well_pointed A $(f : A \rightarrow A)$ $(p : \forall\ x, f\ x = x)$ $(x : A)$:
  map $f\ (p\ x) = p\ (f\ x)$.
`Proof.`
  `intros` $A\ f\ p\ x$.
  apply concat_cancel_right `with` $(r := p\ x)$.
  apply homotopy_naturality_toid.
`Defined.`

Mates

`Lemma` concat_moveright_onright A $(x\ y\ z : A)$ $(p : x = z)$ $(q : x = y)$ $(r : z = y)$:
  $(p = q$ `@ !`$r) \rightarrow (p$ `@` $r = q)$.
`Proof.`
  `intros` $A\ x\ y\ z\ p\ q\ r$.
  `intro` a.
  path_via (q `@ (!`r `@` r`)`).
  associate_left.
`Defined.`

```
Ltac moveright_onright :=
  match goal with
    | ⊢ (?p @ ?r = ?q) ⇒
      apply concat_moveright_onright
    | ⊢ (?r = ?q) ⇒
      path_via (idpath _ @ r); apply concat_moveright_onright
  end; do_opposite_opposite.

Lemma concat_moveleft_onright A (x y z : A) (p : x = y) (q : x = z) (r : z = y) :
  (p @ !r = q) → (p = q @ r).
Proof.
  intros A x y z p q r.
  intro a.
  path_via (p @ (!r @ r)).
  associate_left.
Defined.

Ltac moveleft_onright :=
  match goal with
    | ⊢ (?p = ?q @ ?r) ⇒
      apply concat_moveleft_onright
    | ⊢ (?p = ?r) ⇒
      path_via (idpath _ @ r); apply concat_moveleft_onright
  end; do_opposite_opposite.

Lemma concat_moveleft_onleft A (x y z : A) (p : y = z) (q : x = z) (r : y = x) :
  (!r @ p = q) → (p = r @ q).
Proof.
  intros A x y z p q r.
  intro a.
  path_via ((r @ !r) @ p).
  associate_right.
Defined.

Ltac moveleft_onleft :=
  match goal with
    | ⊢ (?p = ?r @ ?q) ⇒
      apply concat_moveleft_onleft
    | ⊢ (?p = ?r) ⇒
      path_via (r @ idpath _); apply concat_moveleft_onleft
  end; do_opposite_opposite.

Lemma concat_moveright_onleft A (x y z : A) (p : x = z) (q : y = z) (r : y = x) :
  (p = !r @ q) → (r @ p = q).
Proof.
  intros A x y z p q r.
```

```
  intro a.
  path_via ((r @ !r) @ q).
  associate_right.
Defined.
```

```
Ltac moveright_onleft :=
  match goal with
    | ⊢ (?r @ ?p = ?q) ⇒
      apply concat_moveright_onleft
    | ⊢ (?r = ?q) ⇒
      path_via (r @ idpath _); apply concat_moveright_onleft
  end; do_opposite_opposite.
```

Chapter 3

Library Fibrations

```
Require Export Paths.
```

For compatibility with Coq 8.2. Unset *Automatic Introduction*.

In homotopy type theory, We think of elements of Type as spaces or homotopy types, while a type family $P : A \to$ Type corresponds to a fibration whose base is A and whose fiber over x is $P\ x$.

From such a P we can build a total space over the base space A so that the fiber over x : A is $P\ x$. This is just Coq's dependent sum construction, written as $\{x : A \ \&\ P\ x\}$. The elements of $\{x : A \ \&\ P\ x\}$ are pairs, written existT $P\ x\ y$ in Coq, where $x : A$ and $y : P\ x$.

The primitive notation for dependent sum is **sigT** P. Note, though, that in the absence of definitional eta expansion, this is not actually identical with $\{x : A \ \&\ P\ x\}$, since the latter desugars to **sigT** fun($x \Rightarrow P\ x$).

Finally, the base and fiber components of a point in the total space are extracted with projT1 and projT2.

We can also define more familiar homotopy-looking aliases for all of these functions.

```
Notation "( x ; y )" := (existT _ x y).
Notation pr1 := (@projT1 _ _).
Notation pr2 := (@projT2 _ _).
```

An element of section P is a global section of fibration P.

```
Definition section {A} (P : A → Type) := ∀ x : A, P x.
```

We now study how paths interact with fibrations. The most basic fact is that we can transport points in the fibers along paths in the base space. This is actually a special case of the paths_rect induction principle in which the fibration P does not depend on paths in the base space but rather just on points of the base space.

```
Theorem transport {A} {P : A → Type} {x y : A} (p : x = y) : P x → P y.
Proof.
  path_induction.
Defined.
```

A homotopy fiber for a map f at y is the space of paths of the form $f\ x = y$.

```
Definition hfiber {A B} (f : A → B) (y : B) := {x : A & f x = y}.
```

We prove a lemma that explains how to transport a point in the homotopy fiber along a path in the domain of the map.

```
Lemma transport_hfiber A B (f : A → B) (x y : A) (z : B) (p : x = y) (q : f x = z) :
  transport (P := fun x ⇒ f x = z) p q = !(map f p) @ q.
Proof.
  intros A B f x y z p q.
  induction p.
  cancel_units.
Defined.
```

The following lemma tells us how to construct a path in the total space from a path in the base space and a path in the fiber.

```
Lemma total_path (A : Type) (P : A → Type) (x y : sigT P) (p : projT1 x = projT1 y) :
  (transport p (projT2 x) = projT2 y) → (x = y).
Proof.
  intros A P x y p.
  intros q.
  destruct x as [x H].
  destruct y as [y G].
  simpl in × ⊢ ×.
  induction p.
  simpl in q.
  path_induction.
Defined.
```

Conversely, a path in the total space can be projected down to the base.

```
Definition base_path {A} {P : A → Type} {u v : sigT P} :
  (u = v) → (projT1 u = projT1 v) :=
  map pr1.
```

And similarly to the fiber.

```
Definition fiber_path {A} {P : A → Type} {u v : sigT P}
  (p : u = v) : (transport (map pr1 p) (projT2 u) = projT2 v).
Proof.
  path_induction.
Defined.
```

And these operations are inverses. See total_paths_equiv, later on, for a more precise statement.

```
Lemma total_path_reconstruction (A : Type) (P : A → Type) (x y : sigT P) (p : x = y) :
  total_path A P x y (base_path p) (fiber_path p) = p.
```

```
Proof.
  intros A P x y p.
  induction p.
  induction x.
  auto.
Defined.
```

```
Lemma base_total_path (A : Type) (P : A → Type) (x y : sigT P)
  (p : projT1 x = projT1 y) (q : transport p (projT2 x) = projT2 y) :
  (base_path (total_path A P x y p q)) = p.
Proof.
  destruct x as [x H]. destruct y as [y K]. intros p q.
  simpl in p. induction p. simpl in q. induction q.
  auto.
Defined.
```

```
Lemma fiber_total_path (A : Type) (P : A → Type) (x y : sigT P)
  (p : projT1 x = projT1 y) (q : transport p (projT2 x) = projT2 y) :
  transport (P := fun p' : pr1 x = pr1 y ⇒ transport p' (pr2 x) = pr2 y)
  (base_total_path A P x y p q) (fiber_path (total_path A P x y p q))
  = q.
Proof.
  destruct x as [x H]. destruct y as [y K]. intros p q.
  simpl in p. induction p. simpl in q. induction q.
  auto.
Defined.
```

This lemma tells us how to extract a commutative triangle in the base from a path in the homotopy fiber.

```
Lemma hfiber_triangle {A B} {f : A → B} {z : B} {x y : hfiber f z} (p : x = y) :
  (map f (base_path p)) @ (projT2 y) = (projT2 x).
Proof.
  intros. induction p.
  unfold base_path.
  cancel_units.
Defined.
```

Transporting a path along another path is equivalent to concatenating the two paths.

```
Lemma trans_is_concat {A} {x y z : A} (p : x = y) (q : y = z) :
  (transport q p) = p @ q.
Proof.
  path_induction.
Defined.
```

```
Lemma trans_is_concat_opp {A} {x y z : A} (p : x = y) (q : x = z) :
  (transport (P := fun x' ⇒ (x' = z)) p q) = !p @ q.
```

Proof.
  path_induction.
Defined.

Transporting along a concatenation is transporting twice.

Lemma trans_concat $\{A\}$ $\{P : A \to \texttt{Type}\}$ $\{x\ y\ z : A\}$ $(p : x = y)$ $(q : y = z)$ $(z : P\ x)$:
  transport $(p$ @ $q)$ $z =$ transport q (transport p z).
Proof.
  path_induction.
Defined.

Transporting commutes with pulling back along a map.

Lemma map_trans $\{A\ B\}$ $\{x\ y : A\}$ $(P : B \to \texttt{Type})$ $(f : A \to B)$ $(p : x = y)$ $(z : P\ (f\ x))$:
 (transport $(P := (\texttt{fun}\ x \Rightarrow P\ (f\ x)))$ p z) $=$ (transport (map f p) z).
Proof.
  path_induction.
Defined.

And also with applying fiberwise functions.

Lemma trans_map $\{A\}$ $\{P\ Q : A \to \texttt{Type}\}$ $\{x\ y : A\}$ $(p : x = y)$ $(f : \forall\ x, P\ x \to Q\ x)$ $(z : P\ x)$:
  $f\ y$ (transport p z) $=$ (transport p $(f\ x\ z)$).
Proof.
  path_induction.
Defined.

A version of map for dependent functions.

Lemma map_dep $\{A\}$ $\{P : A \to \texttt{Type}\}$ $\{x\ y : A\}$ $(f : \forall\ x, P\ x)$ $(p\!: x = y)$:
  transport p $(f\ x) = f\ y$.
Proof.
  path_induction.
Defined.

Lemma trans_trivial $\{A\ B : \texttt{Type}\}$ $\{x\ y : A\}$ $(p : x = y)$ $(z : B)$:
  transport $(P := \texttt{fun}\ x \Rightarrow B)$ p $z = z$.
Proof.
  path_induction.
Defined.

Lemma map_dep_trivial $\{A\ B\}$ $\{x\ y : A\}$ $(f : A \to B)$ $(p\!: x = y)$:
  map_dep f p $=$ trans_trivial p $(f\ x)$ @ map f p.
Proof.
  path_induction.
Defined.

Lemma map_twovar $\{A : \texttt{Type}\}$ $\{P : A \to \texttt{Type}\}$ $\{B : \texttt{Type}\}$ $\{x\ y : A\}$ $\{a : P\ x\}$ $\{b : P\ y\}$

```
  (f : ∀ x : A, P x → B) (p : x = y) (q : transport p a = b) :
  f x a = f y b.
Proof.
  intros A P B x y a b f p q.
  induction p.
  simpl in q.
  induction q.
  apply idpath.
Defined.

Lemma total_path2 (A : Type) (P : A → Type) (x y : sigT P)
  (p q : x = y) (r : base_path p = base_path q) :
  (transport (P := fun s ⇒ transport s (pr2 x) = (pr2 y)) r (fiber_path p) = fiber_path q)
→ (p = q).
Proof.
  intros A P x y p q r H.
  path_via (total_path A P x y (base_path p) (fiber_path p)) ;
  [ apply opposite, total_path_reconstruction | ].
  path_via (total_path A P x y (base_path q) (fiber_path q)) ;
  [ | apply total_path_reconstruction ].
  apply @map_twovar with
    (f := total_path A P x y)
    (p := r).
  assumption.
Defined.
```

Chapter 4

Library Contractible

Require Export Paths Fibrations.

For compatibility with Coq 8.2. Unset *Automatic Introduction*.

A space A is contractible if there is a point $x : A$ and a (pointwise) homotopy connecting the identity on A to the constant map at x. Thus an element of is_contr A is a pair whose first component is a point x and the second component is a pointwise retraction of A to x.

```
Definition is_contr A := {x : A & ∀ y : A, y = x}.
```

If a space is contractible, then any two points in it are connected by a path in a canonical way.

```
Lemma contr_path {A} (x y : A) : (is_contr A) → (x = y).
Proof.
  intros A x y.
  intro H.
  destruct H as (z,p).
  path_via z.
Defined.
```

Similarly, any two parallel paths in a contractible space are homotopic.

```
Lemma contr_path2 {A} {x y : A} (p q : x = y) : (is_contr A) → (p = q).
Proof.
  intros X x y p q.
  intro ctr.
  destruct ctr as (c, ret).
  path_via (ret x @ !ret y).
  moveleft_onright.
  moveright_onleft.
  apply opposite.
  exact (! trans_is_concat_opp p (ret x) @ map_dep ret p ).
  moveright_onright.
  moveleft_onleft.
```

```
  exact (! trans_is_concat_opp q (ret x) @ map_dep ret q).
Defined.
```

It follows that any space of paths in a contractible space is contractible.

```
Lemma contr_pathcontr {A} (x y : A) : is_contr A → is_contr (x = y).
Proof.
  intros A x y.
  intro ctr.
  ∃ (contr_path x y ctr).
  intro p.
  apply contr_path2.
  assumption.
Defined.
```

The total space of any based path space is contractible.

```
Lemma pathspace_contr {X} (x:X) : is_contr (sigT (paths x)).
Proof.
  intros X x.
  ∃ (x ; idpath x).
  intros [y p].
  path_induction.
Defined.

Lemma pathspace_contr' {X} (x:X) : is_contr { y:X & x = y }.
Proof.
  intros X x.
  ∃ (existT (fun y ⇒ x = y) x (idpath x)).
  intros [y p].
  path_induction.
Defined.
```

The unit type is contractible.

```
Lemma unit_contr : is_contr unit.
Proof.
  ∃ tt.
  intro y.
  induction y.
  auto.
Defined.

Hint Resolve unit_contr.
```

Chapter 5

Library Equivalences

```
Require Export Paths Fibrations Contractible.
```

For compatibility with Coq 8.2. `Unset Automatic Introduction.`

An equivalence is a map whose homotopy fibers are contractible.

```
Definition is_equiv {A B} (f : A -> B) := forall y : B, is_contr (hfiber f y).
```

equiv A B is the space of equivalences from A to B.

```
Definition equiv A B := { w : A -> B & is_equiv w }.
Notation "A <~> B" := (equiv A B) (at level 55).
```

Strictly speaking, an element w of $A \xrightarrow{\sim} B$ is a *pair* consisting of a map projT1 w and the proof projT2 w that it is an equivalence. Thus, in order to apply w to x we must write projT1 w x. Coq is able to do this automatically if we declare that projT1 is a *coercion* from equiv A B to $A \to B$.

```
Definition equiv_coerce_to_function A B (w : A <~> B) : (A -> B)
  := projT1 w.
Coercion equiv_coerce_to_function : equiv >-> Funclass.
```

Here is a tactic which helps us prove that a homotopy fiber is contractible. This will be useful for showing that maps are equivalences.

```
Ltac contract_hfiber y p :=
  match goal with
    | [ |- is_contr (@hfiber _ _ ?f ?x) ] =>
      eexists (existT (fun z => f z = x) y p);
        let z := fresh "z" in
        let q := fresh "q" in
          intros [z q]
  end.
```

Let us explain the tactic. It accepts two arguments y and p and attempts to contract a homotopy fiber to existT _ y p. It first looks for a goal of the form is_contr hfiber(f x), where the question marks in $f?$ and $?x$ are pattern variables that Coq should match against the actual values. If the goal is found, then we use *eexists* to specify that the center of retraction is at the element existT _ y p of hfiber provided by the user. After that we generate some fresh names and perfrom intros.

The identity map is an equivalence.

```
Definition idequiv A : A ≃ A.
Proof.
  intro A.
  ∃ (idmap A).
  intros x.
  contract_hfiber x (idpath x).
  apply total_path with (p := q).
  simpl.
  compute in q.
  path_induction.
Defined.
```

From an equivalence from U to V we can extract a map in the inverse direction.

```
Definition inverse {U V} (w : U ≃ V) : (V → U) :=
  fun y ⇒ pr1 (pr1 ((pr2 w) y)).
Notation "w ^-1" := (inverse w) (at level 40).
```

The extracted map in the inverse direction is actually an inverse (up to homotopy, of course).

```
Definition inverse_is_section {U V} (w : U ≃ V) y : w (w^-1 y) = y :=
  pr2 (pr1 ((pr2 w) y)).
Definition inverse_is_retraction {U V} (w : U ≃ V) x : (w^-1 (w x)) = x :=
  !base_path (pr2 ((pr2 w) (w x)) (x ; idpath (w x))).
```

Here are some tactics to use for canceling inverses, and for introducing them.

```
Ltac cancel_inverses_in s :=
  match s with
    | context cxt [ equiv_coerce_to_function _ _ ?w (?w ^-1 ?x) ] ⇒
      let mid := context cxt [ x ] in
        path_using mid inverse_is_section
    | context cxt [ ?w ^-1 (equiv_coerce_to_function _ _ ?w ?x) ] ⇒
      let mid := context cxt [ x ] in
        path_using mid inverse_is_retraction
  end.
Ltac cancel_inverses :=
  repeat progress (
    match goal with
```

```
      | ⊢ ?s = ?t ⇒ first [ cancel_inverses_in s | cancel_inverses_in t ]
    end
  ).

Ltac expand_inverse_src w x :=
  match goal with
    | ⊢ ?s = ?t ⇒
      match s with
        | context cxt [ x ] ⇒
          first [
            let mid := context cxt [ w (w⁻¹ x) ] in
              path_via' mid;
              [ path_simplify' inverse_is_section | ]
            |
            let mid := context cxt [ w⁻¹ (w x) ] in
              path_via' mid;
              [ path_simplify' inverse_is_retraction | ]
          ]
      end
  end.

Ltac expand_inverse_trg w x :=
  match goal with
    | ⊢ ?s = ?t ⇒
      match t with
        | context cxt [ x ] ⇒
          first [
            let mid := context cxt [ w (w⁻¹ x) ] in
              path_via' mid;
              [ | path_simplify' inverse_is_section ]
            |
            let mid := context cxt [ w⁻¹ (w x) ] in
              path_via' mid;
              [ | path_simplify' inverse_is_retraction ]
          ]
      end
  end.
```

These tactics change between goals of the form $w\ x = y$ and the form $x = w^{-1}\ y$, and dually.

```
Ltac equiv_moveright :=
  match goal with
    | ⊢ equiv_coerce_to_function _ _ ?w ?a = ?b ⇒
      apply @concat with (y := w (w⁻¹ b));
```

```
        [ apply map | apply inverse_is_section ]
    | ⊢ (?w ⁻¹) ?a = ?b ⇒
      apply @concat with (y := w⁻¹ (w b));
        [ apply map | apply inverse_is_retraction ]
  end.
```

```
Ltac equiv_moveleft :=
  match goal with
    | ⊢ ?a = equiv_coerce_to_function _ _ ?w ?b ⇒
      apply @concat with (y := w (w⁻¹ a));
        [ apply opposite, inverse_is_section | apply map ]
    | ⊢ ?a = (?w ⁻¹) ?b ⇒
      apply @concat with (y := w⁻¹ (w a));
        [ apply opposite, inverse_is_retraction | apply map ]
  end.
```

This is one of the "triangle identities" for the preceeding two homotopies. (It doesn't look like a triangle since we've inverted one of the homotopies.)

```
Definition inverse_triangle {A B} (w : A ≃ B) x :
  (map w (inverse_is_retraction w x)) = (inverse_is_section w (w x)).
Proof.
  intros.
  unfold inverse_is_retraction.
  do_opposite_map.
  apply (concat (!idpath_right_unit _ _ _ _)).
  moveright_onleft.
  apply opposite.
  exact (hfiber_triangle (pr2 (pr2 w (w x)) (x ; idpath _))).
Defined.
```

Equivalences are "injective on paths".

```
Lemma equiv_injective U V (w : U ≃ V) x y : (w x = w y) → (x = y).
Proof.
  intros U V w x y.
  intro p.
  expand_inverse_src w x.
  equiv_moveright.
  assumption.
Defined.
```

Anything contractible is equivalent to the unit type.

```
Lemma contr_equiv_unit (A : Type) :
  is_contr A → (A ≃ unit).
Proof.
  intros A H.
```

```
  ∃ (fun x ⇒ tt).
  intro y. destruct y.
  contract_hfiber (pr1 H) (idpath tt).
  apply @total_path with (p := pr2 H z).
  apply contr_path2.
  auto.
Defined.
```

And conversely, anything equivalent to a contractible type is contractible.

```
Lemma contr_equiv_contr (A B : Type) :
  A ≃ B → is_contr A → is_contr B.
Proof.
  intros A B f Acontr.
  destruct Acontr.
  ∃ (f x).
  intro y.
  equiv_moveleft.
  apply p.
Defined.
```

The free path space of a type is equivalent to the type itself.

```
Definition free_path_space A := {xy : A × A & fst xy = snd xy}.

Definition free_path_source A : free_path_space A ≃ A.
Proof.
  intro A.
  ∃ (fun p ⇒ fst (projT1 p)).
  intros x.
  eexists (existT _ (existT (fun (xy : A × A) ⇒ fst xy = snd xy) (x,x) (idpath x)) _).
  intros [[[u v] p] q].
  simpl in × ⊢ ×.
  induction q as [a].
  induction p as [b].
  apply idpath.
Defined.

Definition free_path_target A : free_path_space A ≃ A.
Proof.
  intro A.
  ∃ (fun p ⇒ snd (projT1 p)).
  intros x.
  eexists (existT _ (existT (fun (xy : A × A) ⇒ fst xy = snd xy) (x,x) (idpath x)) _).
  intros [[[u v] p] q].
  simpl in × ⊢ ×.
  induction q as [a].
```

```
  induction p as [b].
  apply idpath.
Defined.
```

We have proven that every equivalence has an inverse up to homotopy. In fact, having an inverse up to homotopy is also enough to characterize a map as being an equivalence. However, the data of an inverse up to homotopy is not equivalent to the data in is_equiv unless we add one more piece of coherence data. This is a homotopy version of the category-theoretic notion of "adjoint equivalence".

```
Definition is_adjoint_equiv {A B} (f : A → B) :=
  { g : B → A &
    { is_section : ∀ y, (f (g y)) = y &
      { is_retraction : ∀ x, (g (f x)) = x &
        ∀ x, (map f (is_retraction x)) = (is_section (f x))
          }}}.

Definition is_equiv_to_adjoint {A B} (f: A → B) (E : is_equiv f) : is_adjoint_equiv f :=
  let w := (f ; E) in
    (w⁻¹ ; (inverse_is_section w; (inverse_is_retraction w ; inverse_triangle w))).

Definition adjoint_equiv (A B : Type) := { f: A → B & is_adjoint_equiv f }.

Theorem is_adjoint_to_equiv {A B} (f: A → B) : is_adjoint_equiv f → is_equiv f.
Proof.
  intros A B f [g [is_section [is_retraction triangle]]].
  intro y.
  contract_hfiber (g y) (is_section y).
  apply (total_path _
    (fun x ⇒ f x = y)
    (existT _ z q)
    (existT _ (g y) (is_section y))
    (!is_retraction z @ (map g q))).
  simpl.
  path_via (!(map f (!is_retraction z @ map g q)) @ q).
  apply transport_hfiber.
  do_concat_map.
  do_opposite_map.
  undo_opposite_concat.
   Here is where we use triangle.      path_via (!map f (map g q) @ is_section (f z) @ q).
   Now it's just naturality of 'is_section'.      associate_right.
  moveright_onleft.
  undo_compose_map.
  apply opposite, homotopy_naturality_toid with (f := f ∘ g).
Defined.
```

Probably equiv_to_adjoint and adjoint_to_equiv are actually inverse equivalences, at least if we assume function extensionality.

```
Lemma equiv_pointwise_idmap A (f : A → A) (p : ∀ x, f x = x) : is_equiv f.
Proof.
  intros.
  apply is_adjoint_to_equiv.
  ∃ (idmap A).
  ∃ p.
  ∃ p.
  apply htoid_well_pointed.
Defined.
```

A central fact about adjoint equivalences is that any "incoherent" equivalence can be improved to an adjoint equivalence by changing one of the natural isomorphisms. We now prove a corresponding result in homotopy type theory. The proof is exactly the same as the usual proof for adjoint equivalences in 2-category theory.

```
Definition adjointify {A B} (f : A → B) (g : B → A) :
  (∀ y, f (g y) = y) → (∀ x, g (f x) = x ) →
  is_adjoint_equiv f.
Proof.
  intros A B f g is_section is_retraction.
  set (is_retraction' := fun x ⇒
    ( map g (map f (!is_retraction x)))
    @ (map g (is_section (f x)))
    @ (is_retraction x)).
  ∃ g.
  ∃ is_section.
  ∃ is_retraction'.
  intro x.
```

Now we just play with naturality until things cancel. `unfold is_retraction'.`

```
  do_concat_map.
  undo_compose_map.
  moveleft_onleft.
  associate_left.
  path_via ((!is_section (f x) @ map (f ∘ g) (map f (!is_retraction x))
    @ map (f ∘ g) (is_section (f x))) @ map f (is_retraction x)).
  unwhisker.
  do_compose_map; auto.
  path_via (map f (!is_retraction x) @ (!is_section (f (g (f x))))
    @ map (f ∘ g) (is_section (f x)) @ map f (is_retraction x)).
  unwhisker.
  apply opposite, (homotopy_naturality_fromid B _ (fun y ⇒ !is_section y)).
  path_via (map f (!is_retraction x) @ (is_section (f x) @ (!is_section (f x))
    @ map f (is_retraction x)).
```

```
  unwhisker.
  apply opposite, (homotopy_naturality_fromid B _ (fun y ⇒ !is_section y)).
  do_opposite_map.
  cancel_right_opposite_of (is_section (f x)).
Defined.
```

Therefore, "any homotopy equivalence is an equivalence."

```
Definition hequiv_is_equiv {A B} (f : A → B) (g : B → A)
  (is_section : ∀ y, f (g y) = y) (is_retraction : ∀ x, g (f x) = x) :
  is_equiv f := is_adjoint_to_equiv f (adjointify f g is_section is_retraction).
```

All sorts of nice things follow from this theorem.

The inverse of an equivalence is an equivalence.

```
Lemma equiv_inverse {A B} (f : A ≃ B) : B ≃ A.
Proof.
  intros.
  destruct (is_equiv_to_adjoint f (pr2 f)) as [g [is_section [is_retraction triangle]]].
  ∃ g.
  exact (hequiv_is_equiv g f is_retraction is_section).
Defined.
```

Anything homotopic to an equivalence is an equivalence.

```
Lemma equiv_homotopic {A B} (f g : A → B) :
  (∀ x, f x = g x) → is_equiv g → is_equiv f.
Proof.
  intros A B f g' p geq.
  set (g := existT is_equiv g' geq : A ≃ B).
  apply @hequiv_is_equiv with (g := g^-1).
  intro y.
  expand_inverse_trg g y; auto.
  intro x.
  equiv_moveright; auto.
Defined.
```

And the 2-out-of-3 property for equivalences.

```
Definition equiv_compose {A B C} (f : A ≃ B) (g : B ≃ C) : (A ≃ C).
Proof.
  intros.
  ∃ (g ∘ f).
  apply @hequiv_is_equiv with (g := (f^-1) ∘ (g^-1)).
  intro y.
  expand_inverse_trg g y.
  expand_inverse_trg f (g^-1 y).
  apply idpath.
```

```
  intro x.
  expand_inverse_trg f x.
  expand_inverse_trg g (f x).
  apply idpath.
Defined.

Definition equiv_cancel_right {A B C} (f : A ≃ B) (g : B -> C) :
  is_equiv (g o f) -> is_equiv g.
Proof.
  intros A B C f g H.
  set (gof := (existT _ (g o f) H) : A ≃ C).
  apply @hequiv_is_equiv with (g := f o (gof^-1)).
  intro y.
  expand_inverse_trg gof y.
  apply idpath.
  intro x.
  change (f (gof^-1 (g x)) = x).
  equiv_moveright; equiv_moveright.
  change (g x = g (f (f^-1 x))).
  cancel_inverses.
Defined.

Definition equiv_cancel_left {A B C} (f : A -> B) (g : B ≃ C) :
  is_equiv (g o f) -> is_equiv f.
Proof.
  intros A B C f g H.
  set (gof := existT _ (g o f) H : A ≃ C).
  apply @hequiv_is_equiv with (g := gof^-1 o g).
  intros y.
  expand_inverse_trg g y.
  expand_inverse_src g (f (((gof ^-1) o g) y)).
  apply map.
  path_via (gof ((gof^-1 (g y)))).
  apply inverse_is_section.
  intros x.
  path_via (gof^-1 (gof x)).
  apply inverse_is_retraction.
Defined.

Definition contr_contr_equiv {A B} (f : A -> B) :
  is_contr A -> is_contr B -> is_equiv f.
Proof.
  intros A B f Acontr Bcontr.
  apply @equiv_cancel_left with
```

```
    (g := contr_equiv_unit B Bcontr).
  exact (pr2 (contr_equiv_unit A Acontr)).
Defined.
```

The action of an equivalence on paths is an equivalence.

```
Theorem equiv_map_inv {A B} {x y : A} (f : A ≃> B) :
  (f x = f y) -> (x = y).
Proof.
  intros A B x y f p.
  path_via (f^-1 (f x)).
  apply opposite, inverse_is_retraction.
  path_via' (f^-1 (f y)).
  apply map. assumption.
  apply inverse_is_retraction.
Defined.

Theorem equiv_map_is_equiv {A B} {x y : A} (f : A ≃> B) :
  is_equiv (@map A B x y f).
Proof.
  intros A B x y f.
  apply @hequiv_is_equiv with (g := equiv_map_inv f).
  intro p.
  unfold equiv_map_inv.
  do_concat_map.
  do_opposite_map.
  moveright_onleft.
  undo_compose_map.
  path_via (map (f o (f ^-1)) p @ inverse_is_section f (f y)).
  apply inverse_triangle.
  path_via (inverse_is_section f (f x) @ p).
  apply homotopy_naturality_toid with (f := f o (f^-1)).
  apply opposite, inverse_triangle.
  intro p.
  unfold equiv_map_inv.
  moveright_onleft.
  undo_compose_map.
  apply homotopy_naturality_toid with (f := (f^-1) o f).
Defined.

Definition equiv_map_equiv {A B} {x y : A} (f : A ≃> B) :
  (x = y) ≃> (f x = f y) :=
  (@map A B x y f ; equiv_map_is_equiv f).
```

Path-concatenation is an equivalence.

```
Lemma concat_is_equiv_left {A} (x y z : A) (p : x = y) :
```

```
  is_equiv (fun q: y = z ⇒ p @ q).
Proof.
  intros A x y z p.
  apply @hequiv_is_equiv with (g := @concat A y x z (!p)).
  intro q.
  associate_left.
  intro q.
  associate_left.
Defined.

Definition concat_equiv_left {A} (x y z : A) (p : x = y) :
  (y = z) ≃ (x = z) :=
  (fun q: y = z ⇒ p @ q ; concat_is_equiv_left x y z p).

Lemma concat_is_equiv_right {A} (x y z : A) (p : y = z) :
  is_equiv (fun q : x = y ⇒ q @ p).
Proof.
  intros A x y z p.
  apply @hequiv_is_equiv with (g := fun r : x = z ⇒ r @ !p).
  intro q.
  associate_right.
  intro q.
  associate_right.
Defined.

Definition concat_equiv_right {A} (x y z : A) (p : y = z) :
  (x = y) ≃ (x = z) :=
  (fun q: x = y ⇒ q @ p ; concat_is_equiv_right x y z p).
```

And we can characterize the path types of the total space of a fibration, up to equivalence.

```
Theorem total_paths_equiv (A : Type) (P : A → Type) (x y : sigT P) :
  (x = y) ≃ { p : pr1 x = pr1 y & transport p (pr2 x) = pr2 y }.
Proof.
  intros A P x y.
  ∃ (fun r ⇒ existT (fun p ⇒ transport p (pr2 x) = pr2 y) (base_path r) (fiber_path r)).
  eapply @hequiv_is_equiv.
  instantiate (1 := fun pq ⇒ let (p,q) := pq in total_path A P x y p q).
  intros [p q].
  eapply total_path.
  instantiate (1 := base_total_path A P x y p q).
  simpl.
  apply fiber_total_path.
  intro r.
  simpl.
```

```
  apply total_path_reconstruction.
Defined.
```

André Joyal suggested the following definition of equivalences, and to call it "h-isomorphism".

```
Definition is_hiso {A B} (f : A → B) :=
  ( { g : B→A & ∀ x, g (f x) = x } ×
    { h : B→A & ∀ y, f (h y) = y } )%type.

Theorem equiv_to_hiso {A B} (f : equiv A B) : is_hiso f.
Proof.
  intros A B f.
  split.
  ∃ (f⁻¹).
  apply inverse_is_retraction.
  ∃ (f⁻¹).
  apply inverse_is_section.
Defined.

Theorem hiso_to_equiv {A B} (f : A → B) : is_hiso f → is_equiv f.
Proof.
  intros A B f H.
  destruct H as ((g, is_retraction), (h, is_section)).
  eapply hequiv_is_equiv.
  instantiate (1 := g).
  intro y.
  path_via (f (h y)).
  path_via (g (f (h (y)))).
  assumption.
Defined.
```

Of course, the harder part is showing that is_hiso is a proposition.

Chapter 6

Library FiberEquivalences

```
Require Export Fibrations Equivalences.
```

For compatibility with Coq 8.2. `Unset` *Automatic Introduction*.

The map on total spaces induced by a map of fibrations

```
Definition total_map {A B : Type} {P : A → Type} {Q : B → Type}
  (f : A → B) (g : ∀ x:A, P x → Q (f x)) :
  sigT P → sigT Q.
Proof.
  intros A B P Q f g.
  intros [x y].
  ∃ (f x).
  exact (g x y).
Defined.
```

We first consider maps between fibrations over the same base space. The theorem is that such a map induces an equivalence on total spaces if and only if it is an equivalence on all fibers.

```
Section FiberMap.

  Variable A : Type.
  Variables P Q : A → Type.
  Variable g : ∀ x, P x → Q x.

  Let tg := total_map (idmap A) g.

  Let tg_is_fiberwise (z : sigT P) : pr1 z = pr1 (tg z).
    intros [x y].
    auto.
  Defined.

  Let tg_isg_onfibers (z : sigT P) :
    g _ (transport (tg_is_fiberwise z) (pr2 z)) = pr2 (tg z).
  Proof.
```

```
  intros [x y].
  auto.
Defined.

Let tg_isfib_onpaths (z w : sigT P) (p : z = w) :
  (tg_is_fiberwise z @ base_path (map tg p) @ !tg_is_fiberwise w) = base_path p.
Proof.
  path_induction.
  destruct x. simpl. auto.
Defined.

Section TotalIsEquiv.

  Hypothesis tot_iseqv : is_equiv tg.

  Let tot_eqv : (sigT P) ≃> (sigT Q) := (tg ; tot_iseqv).

  Let ginv (x:A) (y: Q x) : P x.
  Proof.
    intros x y.
    set (inv1 := pr2 ((tot_eqv^-1) (x ; y))).
    apply (transport (base_path (inverse_is_section tot_eqv (x ; y)))).
    simpl.
    apply (transport (tg_is_fiberwise ((tot_eqv^-1) (x ; y)))).
    assumption.
  Defined.

  Theorem fiber_is_equiv (x:A) : is_equiv (g x).
  Proof.
    intros x.
    set (is_section := inverse_is_section tot_eqv).
    set (is_retraction := inverse_is_retraction tot_eqv).
    set (triangle := inverse_triangle tot_eqv).
    apply @hequiv_is_equiv with (g := ginv x).
    intro y.
    path_via (transport (P := Q)
      (base_path (is_section (x ; y)))
      (pr2 (tot_eqv (tot_eqv^-1 (x ; y))))).
    path_via (transport
      (base_path (is_section (x ; y)))
      (g _ (transport (tg_is_fiberwise (tot_eqv^-1 (x ; y)))
        (pr2 (tot_eqv^-1 (x ; y)))))).
    apply trans_map.
    exact (fiber_path (is_section (existT _ x y))).
    intro y.
    path_via (transport (base_path (map tg (is_retraction (x ; y))))
```

```
    (transport (tg_is_fiberwise (tot_eqv^-1 (x ; (g x y))))
      (pr2 (tot_eqv^-1 (x ; (g x y)))))).
    unfold ginv.
    apply happly, map, map.
    apply opposite, triangle.
    path_via (transport
      (base_path (is_retraction (x ; y)))
      (pr2 (tot_eqv^-1 (x ; (g x y))))).
    path_via (transport
      ((tg_is_fiberwise (tot_eqv^-1 (x ; (g x y))))
        @ (base_path (map tg (is_retraction (x ; y)))))
      (pr2 ((tot_eqv^-1) (x ; (g x y))))).
    apply opposite, trans_concat.
    apply happly, map.
    path_via (tg_is_fiberwise (tot_eqv^-1 (x ; (g x y))) @
      base_path (map tg (is_retraction (x ; y))) @
      !tg_is_fiberwise (x ; y)).
    exact (fiber_path (is_retraction (existT _ x y))).
  Defined.

  Definition fiber_equiv (x:A) : P x ≃ Q x :=
    (g x ; fiber_is_equiv x).

End TotalIsEquiv.

Section FiberIsEquiv.

  Hypothesis fiber_iseqv : ∀ x, is_equiv (g x).

  Let fiber_eqv x : P x ≃ Q x := (g x ; fiber_iseqv x).

  Let total_inv : sigT Q → sigT P.
  Proof.
    intros [x y].
    ∃ x.
    apply ((fiber_eqv x)^-1).
    assumption.
  Defined.

  Theorem total_is_equiv : is_equiv tg.
  Proof.
    eapply hequiv_is_equiv.
    instantiate (1 := total_inv).
    intros [x y].
    eapply total_path.
    instantiate (1 := idpath x).
    path_via (fiber_eqv x ((fiber_eqv x ^-1) y)).
```

```
    apply inverse_is_section.
    intros [x y].
    eapply total_path.
    instantiate (1 := idpath x).
    path_via (fiber_eqv x ^-1 (fiber_eqv x y)).
    apply inverse_is_retraction.
  Defined.

  Definition total_equiv : sigT P ≃> sigT Q :=
    (tg ; total_is_equiv).

End FiberIsEquiv.

End FiberMap.

Implicit Arguments fiber_equiv [A].
Implicit Arguments fiber_is_equiv [A].
Implicit Arguments total_equiv [A].
Implicit Arguments total_is_equiv [A].
```

Next we consider a fibration over one space and its pullback along a map from another base space. The theorem is that if the map we pull back along is an equivalence, so is the induced map on total spaces.

```
Section PullbackMap.

  Variables A B : Type.
  Variable Q : B -> Type.
  Variable f : A ≃> B.

  Let pbQ : A -> Type := Q o f.

  Let g (x:A) : pbQ x -> Q (f x) := idmap (Q (f x)).

  Let tg := total_map f g.

  Let tginv : sigT Q -> sigT pbQ.
  Proof.
    intros [x z].
    exists (f^-1 x).
    apply (transport (! inverse_is_section f x)).
    assumption.
  Defined.

  Theorem pullback_total_is_equiv : is_equiv tg.
  Proof.
    apply @hequiv_is_equiv with (g := tginv).
    intros [x z].
    apply total_path with (p := inverse_is_section f x).
    simpl.
    path_via (transport (! inverse_is_section f x @ inverse_is_section f x) z).
```

```
    apply opposite, trans_concat.
    path_via (transport (idpath x) z).
    apply @map with (f := fun p ⇒ transport p z).
    cancel_opposites.
    intros [x z].
    apply total_path with (p := inverse_is_retraction f x).
    simpl.
    path_via (transport (map f (inverse_is_retraction f x))
      (transport (!inverse_is_section f (f x)) z)).
    apply map_trans.
    path_via (transport (!inverse_is_section f (f x) @ map f (inverse_is_retraction f x)) z).
    apply opposite, trans_concat.
    path_via (transport (idpath (f x)) z).
    assert (p : (!inverse_is_section f (f x) @ map f (inverse_is_retraction f x)) = idpath (f
x)).
    moveright_onleft.
    cancel_units.
    apply inverse_triangle.
    exact (@map _ _ (!inverse_is_section f (f x) @ map f (inverse_is_retraction f x))
      (idpath (f x))
      (fun p ⇒ transport p z) p).
  Defined.

  Definition pullback_total_equiv : sigT pbQ ≃ sigT Q :=
    existT _ tg pullback_total_is_equiv.

End PullbackMap.

Implicit Arguments pullback_total_is_equiv [A B].
Implicit Arguments pullback_total_equiv [A B].
```

Finally, we can put these together to prove that given a map of fibrations lying over an equivalence of base spaces, the induced map on total spaces is an equivalence if and only if the map on each fiber is an equivalence.

```
Section FibrationMap.

  Variables A B : Type.
  Variable P : A → Type.
  Variable Q : B → Type.

  Variable f : A ≃ B.
  Variable g : ∀ x:A, P x → Q (f x).

  Let tg := total_map f g.

  Let pbQ := Q ∘ f.

  Let pbg (x : A) : P x → pbQ x := g x.

  Theorem fibseq_fiber_is_equiv :
```

```
    is_equiv tg → ∀ x, is_equiv (g x).
  Proof.
    intro H.
    set (pbmap_equiv := pullback_total_is_equiv Q f).
    apply fiber_is_equiv.
    apply @equiv_cancel_left with (C := sigT Q) (g := pullback_total_equiv Q f).
    apply @equiv_homotopic with (g := tg).
    intros [x y].
    auto.
    assumption.
  Defined.

  Definition fibseq_fiber_equiv :
    is_equiv tg → ∀ x, P x ≃> Q (f x) :=
      fun H x ⇒ (g x ; fibseq_fiber_is_equiv H x).

  Let fibseq_a_totalequiv :
    (∀ x, is_equiv (g x)) → (sigT P ≃> sigT Q).
  Proof.
    intro H.
    apply @equiv_compose with (B := sigT pbQ).
    ∃ (total_map (idmap A) pbg).
    apply @total_is_equiv.
    apply H.
    apply pullback_total_equiv.
  Defined.

  Theorem fibseq_total_is_equiv :
    (∀ x, is_equiv (g x)) → is_equiv tg.
  Proof.
    intro H.
    apply @equiv_homotopic with (g := fibseq_a_totalequiv H).
    intros [x y].
    auto.
    exact (pr2 (fibseq_a_totalequiv H)).
  Defined.

  Definition fibseq_total_equiv :
    (∀ x, is_equiv (g x)) → (sigT P ≃> sigT Q) :=
    fun H ⇒ (tg ; fibseq_total_is_equiv H).

End FibrationMap.

Implicit Arguments fibseq_fiber_is_equiv [A B].
Implicit Arguments fibseq_fiber_equiv [A B].
Implicit Arguments fibseq_total_is_equiv [A B].
Implicit Arguments fibseq_total_equiv [A B].
```

Chapter 7

Library Funext

```
Require Export Fibrations Contractible Equivalences FiberEquivalences.
```

Much of the content here is closely related to Richard Garner's paper "On the strength of dependent products…". We use different terminology in places, but recall his for comparison.

7.1 Naive functional extensionality

The simplest notion we call "naive functional extensionality". This is what a type theorist would probably write down when thinking of types as sets and identity types as equalities: it says that if two functions are equal pointwise, then they are equal. It comes in both ordinary and dependent versions.

From an HoTT point of view, the type of *extensional equality* or *pointwise equality* between two functions can also be seen as the type of *homotopies* between them.

```
Definition ext_dep_eq {X} {P : X → Type} (f g : ∀ x, P x)
  := ∀ x : X, f x = g x.

Notation "f === g" := (ext_dep_eq f g) (at level 50).

Definition funext_statement : Type :=
  ∀ (X Y : Type) (f g: X → Y), f === g → f = g.

Definition funext_dep_statement : Type :=
  ∀ (X : Type) (P : X → Type) (f g : section P), f === g → (f = g).
```

This is the rule 'Pi-ext' in Garner.

However, there are clearly going to be problems with this in the homotopy world, since "being equal" is not merely a property, but being equipped with a path is structure. We should expect some sort of coherence or canonicity of the path from f to g relating it to the pointwise homotopy we started with.

There are (at least) two natural "computation principles" one might consider. The first fits with thinking of funext as an *eliminator*: it tells us what happens if we apply funext to a term of canonical form.

```
Definition funext_comp1_statement (funext : funext_dep_statement)
  := (∀ X P f, funext X P f f (fun x ⇒ idpath (f x)) = idpath f).
```
A propositional form of Garner's 'Pi-ext-comp'.

Does this rule follow automatically? Yes and no. Given a witness funext : funext_dep_statement, this does not necessarily hold for funext itself; but we can always find a better witness which it does hold:

```
Definition funext_correction : funext_dep_statement → funext_dep_statement
  := (fun funext ⇒
        fun X P f g h ⇒
            (funext X P f g h)
          @
            ! (funext X P g g (fun x ⇒ idpath (g x)))).
Lemma funext_correction_comp1 :
  ∀ (funext : funext_dep_statement),
  funext_comp1_statement (funext_correction funext).
Proof.
  unfold funext_comp1_statement.
  unfold funext_correction.
  auto with path_hints.
Defined.
```

On the other hand, if we think of funext as more like a *1-dimensional constructor* for Pi-types, we can be led to the following rule, telling us what happens to it under the destructor for Pi-types, function application (bumped up to dimension 1 via happly):

```
Definition funext_comp2_statement (funext : funext_dep_statement)
  := (∀ X P f g p x,
       happly_dep (funext X P f g p) x = p x).
```
'Pi-ext-app' in Garner.

Does this rule follow automatically? *Yes*, and in fact for a given witness funext, it's equivalent to funext_comp1_statement above. However, this seems quite non-trivial to prove; it will follow eventually from the comparision with "contractible functional extensionality". So we leave this for now, and will return to it later.

7.2 Strong functional extensionality

Alternatively, a natural way to state a "homotopically good" notion of function extensionality is to observe that there is a canonical map in the other direction, taking paths between functions to pointwise homotopies. We can thus just ask for that map to be an equivalence. We call this "strong functional extensionality." Of course, it also comes in ordinary and dependent versions.

```
Definition strong_funext_statement : Type :=
  ∀ (X Y : Type) (f g : X → Y), is_equiv (@happly X Y f g).
```

```
Definition strong_funext_dep_statement : Type :=
  ∀ (X : Type) (P : X → Type) (f g : section P),
    is_equiv (@happly_dep X P f g).
```

Of course, strong functional extensionality implies naive functional extensionality, along with both computation rules.

```
Theorem strong_to_naive_funext :
  strong_funext_statement → funext_statement.
Proof.
  intros H X Y f g.
  exact ((@happly X Y f g ; H X Y f g) ⁻¹).
Defined.

Theorem strong_funext_compute
  (strong_funext : strong_funext_statement)
  (X Y:Type) (f g : X → Y) (p : f === g) (x : X) :
  happly (strong_to_naive_funext strong_funext X Y f g p) x = p x.
Proof.
  intros.
  unfold strong_to_naive_funext.
  unfold inverse.
  simpl.
  exact (happly_dep (pr2 (pr1 (strong_funext X Y f g p))) x).
Defined.

Theorem strong_to_naive_funext_dep :
  strong_funext_dep_statement → funext_dep_statement.
Proof.
  intros H X Y f g.
  exact ((@happly_dep X Y f g ; H X Y f g) ⁻¹).
Defined.

Theorem strong_funext_dep_comp1
  (strong_funext_dep : strong_funext_dep_statement)
: funext_comp1_statement (strong_to_naive_funext_dep strong_funext_dep).
Proof.
  unfold funext_comp1_statement.
  intros.
  unfold strong_to_naive_funext_dep.
  unfold inverse.
  simpl.
  unfold strong_funext_dep_statement in ×.
  apply (@base_path _ _ (pr1 (strong_funext_dep X P f f (fun x : X ⇒ idpath (f x))))
(idpath f; idpath _)).
  symmetry.
```

```
  apply (pr2 (strong_funext_dep X P f f (fun x : X ⇒ idpath (f x)))).
Defined.

Theorem strong_funext_dep_comp2
  (strong_funext_dep : strong_funext_dep_statement)
  : funext_comp2_statement (strong_to_naive_funext_dep strong_funext_dep).
Proof.
  unfold funext_comp2_statement.
  intros.
  unfold strong_to_naive_funext_dep.
  unfold inverse.
  simpl.
  exact (happly_dep (pr2 (pr1 (strong_funext_dep X P f g p))) x).
Defined.

Definition strong_funext_dep_compute := strong_funext_dep_comp2.
```

Conversely, does naive functional extensionality imply the strong form? Assuming *both* computation rules, this is not hard to show: *comp1* says that funext gives a left inverse to happly, *comp2* that it gives a right inverse.

```
Lemma funext_both_comps_to_strong
  (funext : funext_dep_statement)
  (funext_comp1 : funext_comp1_statement funext)
  (funext_comp2 : funext_comp2_statement funext)
: strong_funext_dep_statement.
Proof.
  intros. unfold strong_funext_dep_statement.
  intros.
  apply (hequiv_is_equiv happly_dep (funext _ _ f g)).
  intro h_fg. apply funext.
  intro x. apply (funext_comp2 X P).
  intro p. destruct p. apply funext_comp1.
Defined.
```

But can we do better, getting to strong functional extensionality from just naive functional extensionality alone? At first the prospects don't look good; naive functional extensionality provides us with paths, but doesn't tell us anything about the behaviour of those paths under elimination, so it seems unlikely that it would be an inverse to happly.

However, it turns out that we can do it! It's easiest to go via another extensionality statement: *contractible functional extensionality*, contr_funext_statement below. Before that, though, we need a quick technical digression on eta rules.

7.3 Eta rules and tactics

Another (very) weak type of functional extensionality is the (propositional) eta rule, which is implied by naive functional extensionality.

```
Definition eta {A B} (f : A → B) :=
  fun x ⇒ f x.
```

```
Definition eta_statement :=
  ∀ (A B:Type) (f : A → B), eta f = f.
```

```
Theorem naive_funext_implies_eta : funext_statement → eta_statement.
Proof.
  intros funext A B f.
  apply funext.
  intro x.
  auto.
Defined.
```

Here is the dependent version.

```
Definition eta_dep {A} {P : A → Type} (f : ∀ x, P x) :=
  fun x ⇒ f x.
```

```
Definition eta_dep_statement :=
  ∀ (A:Type) (P : A → Type) (f : ∀ x, P x), eta_dep f = f.
```

```
Theorem naive_funext_dep_implies_eta : funext_dep_statement → eta_dep_statement.
Proof.
  intros funext_dep A P f.
  apply funext_dep.
  intro x.
  auto.
Defined.
```

A "mini" form of the main theorem (naive => strong) is that the eta rule implies directly that the eta map is an equivalence.

```
Lemma eta_is_equiv : eta_statement → ∀ (A B : Type),
  is_equiv (@eta A B).
Proof.
  intros H A B.
  apply equiv_pointwise_idmap.
  intro f.
  apply H.
Defined.
```

```
Definition eta_equiv (Heta : eta_statement) (A B : Type) :
  (A → B) ≃ (A → B) :=
```

```
  existT is_equiv (@eta A B) (eta_is_equiv Heta A B).
```

And the dependent version.

```
Lemma eta_dep_is_equiv : eta_dep_statement → ∀ (A:Type) (P : A → Type),
   is_equiv (@eta_dep A P).
Proof.
  intros H A P.
  apply equiv_pointwise_idmap.
  intro f.
  apply H.
Defined.

Definition eta_dep_equiv (Heta : eta_dep_statement) (A : Type) (P : A → Type) :
  (∀ x, P x) ≃ (∀ x, P x) :=
  existT is_equiv (@eta_dep A P) (eta_dep_is_equiv Heta A P).
```

Some tactics for working with eta-expansion.

```
Ltac eta_intro f :=
  match goal with
    | [ eta_rule : eta_dep_statement ⊢ ∀ (f : ∀ x:_, _), @?Q f] ⇒
        intro f;
        apply (@transport _ Q _ _ (eta_rule _ _ f));
        unfold eta_dep
    | ⊢ ∀ f, @?Q f ⇒
      let eta_rule := fresh "eta_rule"
      in
        intro f;
        cut eta_dep_statement;

          [ intro eta_rule;
            apply (@transport _ Q _ _ (eta_rule _ _ f));
            unfold eta_dep
          | try auto ]
    | ⊢ _ ⇒
      idtac "Goal not quantified over a function; cannot eta-introduce."
end.

Ltac eta_expand f :=
  revert dependent f;
  eta_intro f.
```

Possible improvements to these tactics:

- At end of *eta_expand*, reintroduce any other hypotheses generalized at the beginning of it.

- Make *eta_expand* work without reverting and re-introducing f?
- In particular, it would be really nice if some form of it could work for arbitrary terms, not just variables; I tried using variations of `match` *goal* `with` $\vdash Q@?\ f$ to do this, but couldn't get it to work.
- Write "plural" versions of these tactics, so one can write i.e. *eta_intros* $f\ g\ h$ to abbreviate *eta_intro* f; *eta_intro* g; *eta_intro* h.

Now we're equipped to tackle the main theorem.

7.4 Contractible functional extensionality, and the proof of strong from naive.

We start by considering yet another version of functional extensionality: that given a function f, the space of functions together with a homotopy to f is contractible. For the sake of cleaner terms, we give a slightly more specific statement than just is_contr $(\ldots)$:

```
Definition contr_funext_statement :=
    ∀ A (B : A → Type) (f : ∀ x:A, B x),
    ∀ (g : ∀ x:A, B x) (h : f === g),
    (g ; h) = (existT (fun g ⇒ f === g) f (fun x ⇒ idpath (f x))).
```

The analogous statement with paths in place of homotopies is, of course, always true. (I'd recalled it being in the library somewhere, but I can't find it now?)

```
Lemma contract_cone {A} {x:A} (yp : { y:A & x = y })
  : yp = (x ; idpath x).
Proof.
  destruct yp as [y p]. path_induction.
Defined.
```

Now, by naive extensionality, the product of all these cones is again contractible:

```
Lemma contract_product_of_cones_from_naive_funext
  {A} {B : A → Type} {f : ∀ x:A, B x}
  : funext_dep_statement →
    ∀ (gh : ∀ x:A, { y:B x & f x = y }),
    gh = (fun x:A ⇒ ( f x ; (idpath (f x)) ) ).
Proof.
  intros funext gh.
  apply funext. intro x.
  apply contract_cone.
Defined.
```

But the type of "functions homotopic to f" is an up-to-eta-expansion retract of this product of cones. So, we define this retraction:

```
Lemma pair_fun_to_fun_pair
  {A} {B : A → Type} {f : ∀ x:A, B x}
  (gh : {g : ∀ x : A, B x & ∀ x : A, f x = g x})
  : ∀ x:A, { y:(B x) & f x = y }.
Proof.
  exact (match gh with
           (g ; h) ⇒ (fun x:A ⇒ (g x ; h x)) end ).
Defined.
```

```
Lemma fun_pair_to_pair_fun
  {A} {B : A → Type} {f : ∀ x:A, B x}
  (k : ∀ x:A, { y:(B x) & f x = y })
  : {g : ∀ x : A, B x & ∀ x : A, f x = g x}.
Proof.
  ∃ (fun x:A ⇒ match (k x) with (gx ; _) ⇒ gx end).
  intro x. destruct (k x) as [gx hx]. exact hx.
Defined.
```

...and now we have all the ingredients for proving contractible funext from naive funext (or alternatively from weak funext + dependent eta):

```
Theorem naive_to_contr_funext
  : funext_dep_statement
    → contr_funext_statement.
Proof.
  intros funext.
  unfold contr_funext_statement. intros A B.
  eta_intro f. eta_intro g. eta_intro h.
  path_via (fun_pair_to_pair_fun (pair_fun_to_fun_pair (g ; h))).
  path_via (@fun_pair_to_pair_fun _ _ (fun x ⇒ f x) (fun x ⇒ (f x ; idpath (f x)))).
  apply contract_product_of_cones_from_naive_funext. assumption.
  apply naive_funext_dep_implies_eta; auto.
Defined.
```

```
Lemma contr_funext_to_comp2 (funext : funext_dep_statement)
  : (funext_comp1_statement funext)
  → contr_funext_statement
  → (funext_comp2_statement funext).
Proof.
  intros funext_comp1 contr_funext.
  unfold funext_comp2_statement. intros X P f g h.
  apply (@transport _
          (fun (g0h0 : { g : section P & f === g })
```

```
            ⇒ match g0h0 with (g0;h0)
            ⇒ (∀ x : X, happly_dep (funext X P f g0 h0) x = h0 x) end)
          (existT (fun g ⇒ f === g) f (fun x ⇒ idpath (f x)))
          (g ; h)).
  symmetry. apply contr_funext.
  clear g h. intro x.
  path_via (happly_dep (idpath f) x).
  apply_happly. path_simplify.
  apply funext_comp1.
Defined.

Theorem funext_comp1_to_comp2 (funext : funext_dep_statement)
  : (funext_comp1_statement funext) → (funext_comp2_statement funext).
Proof.
  intro funext_comp1.
  apply contr_funext_to_comp2; auto.
  apply naive_to_contr_funext; auto.
Defined.

Lemma funext_correction_comp2 (funext : funext_dep_statement)
  : funext_comp2_statement (funext_correction funext).
Proof.
  apply funext_comp1_to_comp2.
  apply funext_correction_comp1.
Defined.

Theorem naive_to_strong_funext
  : funext_dep_statement → strong_funext_dep_statement.
Proof.
  intro funext.
  apply (funext_both_comps_to_strong (funext_correction funext)).
  apply funext_correction_comp1.
  apply funext_correction_comp2.
Defined.
```

Alternatively, we can show strong funext entirely from contractible funext, without ever invoking naive:

```
Theorem contr_to_strong_funext :
  contr_funext_statement → strong_funext_dep_statement.
Proof.
  intros contr_funext X P f g.
  set (A := ∀ x, P x).
  set (Q := (fun h ⇒ f = h) : A → Type).
  set (R := (fun h ⇒ ∀ x, f x = h x) : A → Type).
  set (fibhap := (@happly_dep X P f) : ∀ h, Q h → R h).
```

```
  apply (fiber_is_equiv _ _ fibhap). clear g.
  apply contr_contr_equiv.
  apply pathspace_contr'.
  unfold is_contr. ∃ (existT R f (fun x ⇒ idpath (f x))).
  intros [g h]. apply contr_funext.
Defined.
```

7.5 Weak functional extensionality

Inspection of the proof of naive_to_contr_funext shows that it only uses functional extensionality via two simpler statements: eta_dep_statement, and the fact that a product of contractible types is contractible.

This latter statement is interesting in its own right; we call it *weak functional extensionality*.

Among other things, it can be seen from the model category point of view as saying that the dependent product functor preserves trivial fibrations, which is exactly (the non-trivial part of) what's needed to make pullback/dependent-product a Quillen adjunction!

```
Definition weak_funext_statement := ∀ (X : Type) (P : X → Type),
  (∀ x : X, is_contr (P x)) → is_contr (∀ x : X, P x).
```

It is easy to see that naive dependent functional extensionality implies weak functional extensionality.

```
Theorem funext_dep_to_weak :
  funext_dep_statement → weak_funext_statement.
Proof.
  intros H X P H1.
  ∃ (fun x ⇒ projT1 (H1 x)).
  intro f.
  assert (p : ∀ (x:X) (y:P x), y = ((fun x ⇒ projT1 (H1 x)) x)).
  intros. apply contr_path, H1.
  apply H. intro x. apply p.
Defined.
```

Now we can give an alternative form of the main theorem: the fact that weak functional extensionality implies *strong* (dependent) functional extensionality, at least in the presence of the dependent eta rule.

```
Lemma is_contr_product_of_cones_from_weak_funext
  {A} {B : A → Type} {f : ∀ x:A, B x}
  : weak_funext_statement →
    is_contr (∀ x:A, { y:B x & f x = y }).
Proof.
  intro weak_funext. apply weak_funext.
```

```
  intro x. ∃ ((f x ; idpath (f x)) : {y : B x & f x = y}).
  intros [y p]. path_induction.
Defined.
```

We can now essentially repeat the proof of naive_to_contr_funext: `Theorem weak_plus_eta_to_contr_fune`

```
  : eta_dep_statement → weak_funext_statement → contr_funext_statement.
Proof.
  intros eta_dep weak_funext.
  unfold contr_funext_statement. intros A B.
  eta_intro f. eta_intro g. eta_intro h.
  path_via (fun_pair_to_pair_fun (pair_fun_to_fun_pair (g ; h))).
  path_via (@fun_pair_to_pair_fun _ _ (fun x ⇒ f x) (fun x ⇒ (f x ; idpath (f x)))).
  apply contr_path.
  apply is_contr_product_of_cones_from_weak_funext. assumption.
Defined.
```

```
Theorem weak_to_strong_funext_dep :
  eta_dep_statement → weak_funext_statement → strong_funext_dep_statement.
Proof.
  intros eta_dep weak_funext.
  apply contr_to_strong_funext.
  apply weak_plus_eta_to_contr_funext; assumption.
Defined.
```

Therefore, all of the following are equivalent, in their dependent forms:

- naive functional extensionality;
- naive functional extensionality with either or both comp rules;
- strong functional extensionality;
- contractible functional extensionality;
- weak functional extensionality + dependent eta.

7.6 Comparing dependent and non-dependent forms.

We also observe that for both strong and naive functional extensionality, the dependent version implies the non-dependent version.

```
Theorem strong_funext_dep_to_nondep :
  strong_funext_dep_statement → strong_funext_statement.
Proof.
  intros H X Y f g.
```

```
  exact (H X (fun x ⇒ Y) f g).
Defined.

Theorem funext_dep_to_nondep :
  funext_dep_statement → funext_statement.
Proof.
  intros H X Y f g.
  exact (H X (fun x ⇒ Y) f g).
Defined.
```

One can prove similar things for the other variants considered. Can we go the other way, for any of the variants?

Chapter 8

Library Univalence

```
Require Import Paths Fibrations Contractible Equivalences.
```

For compatibility with Coq 8.2. `Unset` *Automatic Introduction.*

Every path between spaces gives an equivalence.

```
Definition path_to_equiv {U V} : (U = V) → (U ≃ V).
Proof.
  intros U V.
  path_induction.
  apply idequiv.
Defined.
```

This is functorial in the appropriate sense.

```
Lemma path_to_equiv_map {A} (P : A → Type) (x y : A) (p : x = y) :
  projT1 (path_to_equiv (map P p)) = transport (P := P) p.
Proof.
  path_induction.
Defined.

Lemma concat_to_compose {A B C} (p : A = B) (q : B = C) :
  path_to_equiv q ○ path_to_equiv p = projT1 (path_to_equiv (p @ q)).
Proof.
  path_induction.
Defined.

Ltac undo_concat_to_compose_in s :=
  match s with
    | context cxt [ equiv_coerce_to_function _ _ (path_to_equiv ?p) ○ equiv_coerce_to_function
_ _ (path_to_equiv ?q) ] ⇒
      let mid := context cxt [ equiv_coerce_to_function _ _ (path_to_equiv (q @ p)) ] in
        path_via mid;
        [ repeat first [ apply happly | apply map | apply concat_to_compose ] | ]
  end.
```

```
Ltac undo_concat_to_compose :=
  repeat progress (
    match goal with
      | ⊢ ?s = ?t ⇒
        first [ undo_concat_to_compose_in s | undo_concat_to_compose_in t ]
    end
  ).
```

```
Lemma opposite_to_inverse {A B} (p : A = B) :
  (path_to_equiv p)^-1 = path_to_equiv (!p).
Proof.
  path_induction.
Defined.
```

```
Ltac undo_opposite_to_inverse_in s :=
  match s with
    | context cxt [ (path_to_equiv ?p) ^-1 ] ⇒
      let mid := context cxt [ equiv_coerce_to_function _ _ (path_to_equiv (! p)) ] in
        path_via mid;
        [ repeat apply map; apply opposite_to_inverse | ]
  end.
```

```
Ltac undo_opposite_to_inverse :=
  repeat progress (
    match goal with
      | ⊢ ?s = ?t ⇒
        first [ undo_opposite_to_inverse_in s | undo_opposite_to_inverse_in t ]
    end
  ).
```

The statement of the univalence axiom.

```
Definition univalence_statement := ∀ (U V : Type), is_equiv (@path_to_equiv U V).
```

```
Section Univalence.
```

```
  Hypothesis univalence : univalence_statement.
```

```
  Definition path_to_equiv_equiv (U V : Type) := (@path_to_equiv U V ; univalence U
V).
```

Assuming univalence, every equivalence yields a path.

```
  Definition equiv_to_path {U V} : U ≃ V → U = V :=
    inverse (path_to_equiv_equiv U V).
```

The map equiv_to_path is a section of path_to_equiv.

```
  Definition equiv_to_path_section U V :
    ∀ (w : U ≃ V), path_to_equiv (equiv_to_path w) = w :=
    inverse_is_section (path_to_equiv_equiv U V).
```

```
Definition equiv_to_path_retraction U V :
  ∀ (p : U = V), equiv_to_path (path_to_equiv p) = p :=
  inverse_is_retraction (path_to_equiv_equiv U V).
```

```
Definition equiv_to_path_triangle U V : ∀ (p : U = V),
    map path_to_equiv (equiv_to_path_retraction U V p) = equiv_to_path_section U V
(path_to_equiv p) :=
  inverse_triangle (path_to_equiv_equiv U V).
```

We can do better than equiv_to_path: we can turn a fibration fibered over equivalences to one fiberered over paths.

```
Definition pred_equiv_to_path U V : (U ≃ V → Type) → (U = V → Type).
Proof.
  intros U V.
  intros Q p.
  apply Q.
  apply path_to_equiv.
  exact p.
Defined.
```

The following theorem is of central importance. Just like there is an induction principle for paths, there is a corresponding one for equivalences. In the proof we use pred_equiv_to_path to transport the predicate P of equivalences to a predicate P' on paths. Then we use path induction and transport back to P.

```
Theorem equiv_induction (P : ∀ U V, U ≃ V → Type) :
  (∀ T, P T T (idequiv T)) → (∀ U V (w : U ≃ V), P U V w).
Proof.
  intros P.
  intro r.
  pose (P' := (fun U V ⇒ pred_equiv_to_path U V (P U V))).
  assert (r' : ∀ T : Type, P' T T (idpath T)).
  intro T.
  exact (r T).
  intros U V w.
  apply (transport (equiv_to_path_section _ _ w)).
  exact (paths_rect _ P' r' U V (equiv_to_path w)).
Defined.
```

```
End Univalence.
```

Chapter 9

Library UnivalenceImpliesFunext

`Require Import` Paths Fibrations Contractible Equivalences Univalence Funext.

For compatibility with Coq 8.2. `Unset` *Automatic Introduction*.

Here we prove that univalence implies function extensionality. We keep this file separate from the statements of Univalence and Funext, since it has a tendency to produce universe inconsistencies. With truly polymorphic universes this ought not to be a problem.

Since this file makes the point that univalence implies funext, further development can avoid including this file and simply assume function extensionality as an axiom alongside univalence, in the knowledge that it is actually no additional requirement.

`Section` UnivalenceImpliesFunext.

`Hypothesis` *univalence* : univalence_statement.

`Hypothesis` *eta_rule* : eta_statement.

Exponentiation preserves equivalences, i.e., if w is an equivalence then so is post-composition by w.

```
Theorem equiv_exponential : ∀ {A B} (w : A ≃> B) C,
  (C → A) ≃> (C → B).
Proof.
  intros A B w C.
  ∃ (fun h ⇒ w ∘ h).
  generalize A B w.
  apply equiv_induction.
  assumption.
  intro D.
  apply (projT2 (eta_equiv eta_rule C D)).
Defined.
```

We are ready to prove functional extensionality, starting with the naive non-dependent version.

`Theorem` univalence_implies_funext : funext_statement.

```
Proof.
  intros A B f g p.
  apply equiv_injective with (w := eta_equiv eta_rule A B).
  simpl.
  pose (d := fun x : A ⇒ existT (fun xy ⇒ fst xy = snd xy) (f x , f x) (idpath (f x))).
  pose (e := fun x : A ⇒ existT (fun xy ⇒ fst xy = snd xy) (f x , g x) (p x)).
  pose (src_compose := equiv_exponential (free_path_source B) A).
  pose (trg_compose := equiv_exponential (free_path_target B) A).
  path_via (projT1 trg_compose e).
  path_via (projT1 trg_compose d).
  apply equiv_injective with (w := src_compose).
  apply idpath.
Defined.
```

Now we use this to prove weak funext, which as we know implies (with dependent eta) also the strong dependent funext.

```
Theorem univalence_implies_weak_funext : weak_funext_statement.
Proof.
  intros X P allcontr.
  assert (eqpt : @paths (X → Type) (fun x ⇒ unit) P).
  apply univalence_implies_funext.
  intro x.
  apply opposite, equiv_to_path, contr_equiv_unit, allcontr.
  assumption.
  assert (contrunit : is_contr (∀ x:X, unit)).
  ∃ (fun _ ⇒ tt).
  intro f.
  apply univalence_implies_funext.
  intro x.
  assert (alltt : ∀ y:unit, y = tt).
  induction y; apply idpath.
  apply alltt.
  exact (transport (P := fun Q: X → Type ⇒ is_contr (∀ x, Q x)) eqpt contrunit).
Admitted.

End UnivalenceImpliesFunext.
```

Chapter 10

Library UnivalenceAxiom

```
Require Import Paths Univalence Funext.
```

This file asserts univalence as a global axiom, along with its basic consequences, including function extensionality. Since the proof that univalence implies funext has a tendency to create universe inconsistencies, we actually assume funext as a separate axiom rather than actually deriving it from univalence.

```
Axiom univalence : univalence_statement.
```

```
Set Implicit Arguments.
Definition equiv_to_path := @equiv_to_path univalence.
Definition equiv_to_path_section := @equiv_to_path_section univalence.
Definition equiv_to_path_retraction := @equiv_to_path_retraction univalence.
Definition equiv_to_path_triangle := @equiv_to_path_triangle univalence.
Definition equiv_induction := @equiv_induction univalence.
```

```
Axiom strong_funext_dep : strong_funext_dep_statement.
Definition strong_funext := strong_funext_dep_to_nondep strong_funext_dep.
Definition funext_dep := strong_to_naive_funext_dep strong_funext_dep.
Definition funext := strong_to_naive_funext strong_funext.
Definition weak_funext := funext_dep_to_weak funext_dep.
Definition funext_dep_compute := strong_funext_dep_compute strong_funext_dep.
Definition funext_compute := strong_funext_compute strong_funext.
```

Chapter 11

Library HLevel

```
Require Import Paths Fibrations Contractible Equivalences Funext.
Require Import UnivalenceAxiom.
```

For compatibility with Coq 8.2. `Unset` *Automatic Introduction*.

Some more stuff about contractibility.

```
Theorem contr_contr {X} : is_contr X → is_contr (is_contr X).
  intros X ctr1.
  ∃ ctr1. intros ctr2.
  apply @total_path with (p := pr2 ctr1 (pr1 ctr2)).
  apply funext_dep.
  intro x.
  apply contr_path2.
  assumption.
Defined.
```

H-levels.

```
Fixpoint is_hlevel (n : nat) : Type → Type :=
  match n with
    | 0 ⇒ is_contr
    | S n' ⇒ fun X ⇒ ∀ (x y:X), is_hlevel n' (x = y)
  end.

Theorem hlevel_inhabited_contr {n X} : is_hlevel n X → is_contr (is_hlevel n X).
Proof.
  intros n.
  induction n.
  intro X.
  apply contr_contr.
  intro X.
  simpl.
  intro H.
```

```
  apply weak_funext.
  intro x.
  apply weak_funext.
  intro y.
  apply IHn.
  apply H.
Defined.
```

H-levels are increasing with n.

```
Theorem hlevel_succ {n X} : is_hlevel n X → is_hlevel (S n) X.
Proof.
  intros n.
  induction n.
  intros X H x y.
  apply contr_pathcontr.
  assumption.
  intros X H x y.
  apply IHn.
  apply H.
Defined.
```

H-level is preserved under equivalence.

```
Theorem hlevel_equiv {n A B} : (A ≃ B) → is_hlevel n A → is_hlevel n B.
Proof.
  intro n.
  induction n.
  simpl.
  apply @contr_equiv_contr.
  simpl.
  intros A B f H x y.
  apply IHn with (A := f (f⁻¹ x) = y).
  apply concat_equiv_left.
  apply opposite, inverse_is_section.
  apply IHn with (A := f (f⁻¹ x) = f (f⁻¹ y)).
  apply concat_equiv_right.
  apply inverse_is_section.
  apply IHn with (A := (f⁻¹ x) = (f⁻¹ y)).
  apply equiv_map_equiv.
  apply H.
Defined.
```

Propositions are of h-level 1.

```
Definition is_prop := is_hlevel 1.
```

Here is an alternate characterization of propositions.

```
Theorem prop_inhabited_contr {A} : is_prop A → A → is_contr A.
Proof.
  intros A H x.
  ∃ x.
  intro y.
  apply H.
Defined.

Theorem inhabited_contr_isprop {A} : (A → is_contr A) → is_prop A.
Proof.
  intros A H x y.
  apply contr_pathcontr.
  apply H.
  assumption.
Defined.

Theorem hlevel_isprop {n A} : is_prop (is_hlevel n A).
Proof.
  intros n A.
  apply inhabited_contr_isprop.
  apply hlevel_inhabited_contr.
Defined.

Definition isprop_isprop {A} : is_prop (is_prop A) := hlevel_isprop.

Theorem prop_equiv_inhabited_contr {A} : is_prop A ≃ (A → is_contr A).
Proof.
  intros A.
  ∃ prop_inhabited_contr.
  apply hequiv_is_equiv with (g := inhabited_contr_isprop).
  intro H.
  unfold prop_inhabited_contr, inhabited_contr_isprop.
  simpl.
  apply funext.
  intro x.
  apply contr_path.
  apply contr_contr.
  exact (H x).
  intro H.
  unfold prop_inhabited_contr, inhabited_contr_isprop.
  apply funext_dep.
  intro x.
  apply funext_dep.
  intro y.
  apply contr_path.
```

```
  apply contr_contr.
  exact (H x y).
Defined.
```

And another one.

```
Theorem prop_path {A} : is_prop A → ∀ (x y : A), x = y.
Proof.
  intro A.
  unfold is_prop. simpl.
  intros H x y.
  exact (pr1 (H x y)).
Defined.

Theorem allpath_prop {A} : (∀ (x y : A), x = y) → is_prop A.
  intro A.
  intros H x y.
  assert (K : is_contr A).
  ∃ x. intro y'. apply H.
  apply contr_pathcontr. assumption.
Defined.

Theorem prop_equiv_allpath {A} : is_prop A ≃ (∀ (x y : A), x = y).
Proof.
  intro A.
  ∃ prop_path.
  apply @hequiv_is_equiv with (g := allpath_prop).
  intro H.
  apply funext_dep.
  intro x.
  apply funext_dep.
  intro y.
  apply contr_path.
  apply (allpath_prop H).
  intro H.
  apply funext_dep.
  intro x.
  apply funext_dep.
  intro y.
  apply contr_path.
  apply contr_contr.
  apply H.
Defined.
```

Sets are of h-level 2.

```
Definition is_set := is_hlevel 2.
```

A type is a set if and only if it satisfies Axiom K.

```
Definition axiomK A := ∀ (x : A) (p : x = x), p = idpath x.

Definition isset_implies_axiomK {A} : is_set A → axiomK A.
Proof.
  intros A H x p.
  apply H.
Defined.

Definition axiomK_implies_isset {A} : axiomK A → is_set A.
Proof.
  intros A H x y.
  apply allpath_prop.
  intros p q.
  induction q.
  apply H.
Defined.

Theorem isset_equiv_axiomK {A} :
  is_set A ≃ (∀ (x : A) (p : x = x), p = idpath x).
Proof.
  intro A.
  ∃ isset_implies_axiomK.
  apply @hequiv_is_equiv with (g := axiomK_implies_isset).
  intro H.
  apply funext_dep.
  intro x.
  apply funext_dep.
  intro p.
  apply contr_path.
  apply (axiomK_implies_isset H).
  intro H.
  apply funext_dep.
  intro x.
  apply funext_dep.
  intro y.
  apply prop_path.
  apply isprop_isprop.
Defined.

Definition isset_isprop {A} : is_prop (is_set A) := hlevel_isprop.

Theorem axiomK_isprop {A} : is_prop (axiomK A).
Proof.
  intro A.
  apply @hlevel_equiv with (A := is_set A).
```

```
  apply isset_equiv_axiomK.
  apply hlevel_isprop.
Defined.
```

```
Theorem set_path2 (A : Type) (x y : A) (p q : x = y) :
  is_set A → (p = q).
Proof.
  intros A x y p q.
  intro H.
  apply contr_path.
  apply prop_inhabited_contr.
  cbv. cbv in H.
  apply H.
  assumption.
Defined.
```

Recall that axiom K says that any self-path is homotopic to the identity path. In particular, the identity path is homotopic to itself. The following lemma says that the endo-homotopy of the identity path thus specified is in fact (homotopic to) its identity homotopy (whew!).

```
Lemma axiomK_idpath (A : Type) (x : A) (K : axiomK A) :
  K x (idpath x) = idpath (idpath x).
Proof.
  intros.
  set (qq := map_dep (K x) (K x (idpath x))).
  set (q2 := !trans_is_concat_opp (K x (idpath x)) (K x (idpath x)) @ qq).
  path_via (!! K x (idpath x)).
  path_via (! idpath (idpath x)).
  apply concat_cancel_right with (r := K x (idpath x)).
  cancel_units.
Defined.
```

Any type with "decidable equality" is a set.

```
Definition decidable_paths (A : Type) :=
  ∀ (x y : A), (x = y) + ((x = y) → Empty_set).
```

```
Definition inl_injective (A B : Type) (x y : A) (p : inl B x = inl B y) : (x = y) :=
  transport (P := fun (s:A+B) ⇒ x = match s with inl a ⇒ a | inr b ⇒ x end) p (idpath
x).
```

```
Theorem decidable_isset (A : Type) :
  decidable_paths A → is_set A.
Proof.
  intros A d.
  apply axiomK_implies_isset.
  intros x p.
```

```
  set (q := d x x).
  set (qp := map_dep (d x) p).
  fold q in qp.
  generalize qp.
  clear qp.
  destruct q as [q | q'].
  intro qp0.
  apply concat_cancel_left with (p := q).
  path_via (transport p q).
  apply opposite, trans_is_concat.
  path_via q.
  set (qp1 := trans_map p (fun (x0:A) ⇒ inl (x = x0 → Empty_set)) q).
  apply inl_injective with (B := (x = x → Empty_set)).
  exact (qp1 @ qp0).
  induction (q' p).
Defined.
```

Chapter 12

Library Homotopy

```
Require Export Paths Fibrations Contractible Equivalences FiberEquivalences.
Require Export Funext Univalence UnivalenceAxiom.
Require Export HLevel.
```