Chapter 1

Library Functions

Definition idmap $A := \texttt{fun}\ x : A \Rightarrow x$.

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

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

Chapter 2

Library Paths

Basic homotopy-theoretic approach to paths.

For compatibility with Coq 8.2.

`Inductive` **paths** $\{A\} : A \to A \to$ `Type` := idpath : $\forall\ x$, **paths** $x\ x$.

We introduce notation $x \rightsquigarrow y$ for the space **paths** $x\ y$ of paths from x to y. We can then write $p : x \rightsquigarrow y$ 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 $x \rightsquigarrow x$, `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).

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

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 $\{A\}\ \{x\ y\ z : A\} : (x \rightsquigarrow y) \to (y \rightsquigarrow z) \to (x \rightsquigarrow z)$.

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 $x \rightsquigarrow z$ as a concatenation of paths $x \rightsquigarrow y \rightsquigarrow z$. 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 $\{A\}$ $\{x\ y : A\}$: $(x \rightsquigarrow y) \to (y \rightsquigarrow x)$.

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 $(x\ y : A)$ $(p : x \rightsquigarrow y)$: idpath x @ $p \rightsquigarrow p$.

Lemma idpath_right_unit A $(x\ y : A)$ $(p : x \rightsquigarrow y)$: (p @ idpath y) $\rightsquigarrow p$.

Lemma opposite_right_inverse A $(x\ y : A)$ $(p : x \rightsquigarrow y)$: (p @ !p) $\rightsquigarrow$ idpath x.

Lemma opposite_left_inverse A $(x\ y : A)$ $(p : x \rightsquigarrow y)$: (!p @ p) $\rightsquigarrow$ idpath y.

Lemma opposite_concat A $(x\ y\ z : A)$ $(p : x \rightsquigarrow y)$ $(q : y \rightsquigarrow z)$: !(p @ q) $\rightsquigarrow$!q @ !p.

Lemma opposite_idpath A $(x : A)$: !(idpath x) $\rightsquigarrow$ idpath x.

Lemma opposite_opposite A $(x\ y : A)$ $(p : x \rightsquigarrow y)$: !(! p) $\rightsquigarrow p$.

Lemma concat_associativity A $(w\ x\ y\ z : A)$ $(p : w \rightsquigarrow x)$ $(q : x \rightsquigarrow y)$ $(r : y \rightsquigarrow z)$:
(p @ q) @ $r \rightsquigarrow p$ @ (q @ r).

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 $\{A\}$ $\{x\ y\ z : A\}$ $\{p\ p' : x \rightsquigarrow y\}$ $\{q\ q' : y \rightsquigarrow z\}$:
$(p \rightsquigarrow p') \to (q \rightsquigarrow q') \to (p$ @ $q \rightsquigarrow p'$ @ $q')$.

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

We also have whiskering operations.

Definition whisker_right $\{A\}$ $\{x\ y\ z : A\}$ $\{p\ p' : x \rightsquigarrow y\}$ $(q : y \rightsquigarrow z)$:
$(p \rightsquigarrow p') \to (p$ @ $q \rightsquigarrow p'$ @ $q)$.

Definition whisker_left $\{A\}$ $\{x\ y\ z : A\}$ $\{q\ q' : y \rightsquigarrow z\}$ $(p : x \rightsquigarrow y)$:
$(q \rightsquigarrow q') \to (p$ @ $q \rightsquigarrow p$ @ $q')$.

Definition whisker_right_toid $\{A\}$ $\{x\ y : A\}$ $\{p : x \rightsquigarrow x\}$ $(q : x \rightsquigarrow y)$:

$(p \rightsquigarrow \mathsf{idpath}\ x) \to (p\ \mathtt{@}\ q \rightsquigarrow q)$.

`Definition` whisker_right_fromid $\{A\}\ \{x\ y : A\}\ \{p : x \rightsquigarrow x\}\ (q : x \rightsquigarrow y) :$
$(\mathsf{idpath}\ x \rightsquigarrow p) \to (q \rightsquigarrow p\ \mathtt{@}\ q)$.

`Definition` whisker_left_toid $\{A\}\ \{x\ y : A\}\ \{p : y \rightsquigarrow y\}\ (q : x \rightsquigarrow y) :$
$(p \rightsquigarrow \mathsf{idpath}\ y) \to (q\ \mathtt{@}\ p \rightsquigarrow q)$.

`Definition` whisker_left_fromid $\{A\}\ \{x\ y : A\}\ \{p : y \rightsquigarrow y\}\ (q : x \rightsquigarrow y) :$
$(\mathsf{idpath}\ y \rightsquigarrow p) \to (q \rightsquigarrow q\ \mathtt{@}\ p)$.

The interchange law for whiskering.

`Definition` whisker_interchange $A\ (x\ y\ z : A)\ (p\ p' : x \rightsquigarrow y)\ (q\ q' : y \rightsquigarrow z)$
$(a : p \rightsquigarrow p')\ (b : q \rightsquigarrow q') :$
$(\mathsf{whisker\_right}\ q\ a)\ \mathtt{@}\ (\mathsf{whisker\_left}\ p'\ b) \rightsquigarrow (\mathsf{whisker\_left}\ p\ b)\ \mathtt{@}\ (\mathsf{whisker\_right}\ q'\ a)$.

The interchange law for concatenation.

`Definition` concat2_interchange $A\ (x\ y\ z : A)\ (p\ p'\ p'' : x \rightsquigarrow y)\ (q\ q'\ q'' : y \rightsquigarrow z)$
$(a : p \rightsquigarrow p')\ (b : p' \rightsquigarrow p'')\ (c : q \rightsquigarrow q')\ (d : q' \rightsquigarrow q'') :$
$(a\ \mathtt{@@}\ c)\ \mathtt{@}\ (b\ \mathtt{@@}\ d) \rightsquigarrow (a\ \mathtt{@}\ b)\ \mathtt{@@}\ (c\ \mathtt{@}\ d)$.

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

`Definition` opposite2 $A\ \{x\ y : A\}\ (p\ q : x \rightsquigarrow y)\ (a : p \rightsquigarrow q) : (\mathtt{!}p \rightsquigarrow \mathtt{!}q)$.

Now we consider the application of functions to paths.

A path $p : x \rightsquigarrow y$ in a space A is mapped by $f : A \to B$ to a path $\mathsf{map}\ f\ p : f\ x \rightsquigarrow f\ y$ in B.

`Lemma` map $\{A\ B\}\ \{x\ y : A\}\ (f : A \to B) : (x \rightsquigarrow y) \to (f\ x \rightsquigarrow f\ y)$.

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

`Lemma` idpath_map $A\ B\ (x : A)\ (f : A \to B) : \mathsf{map}\ f\ (\mathsf{idpath}\ x) \rightsquigarrow \mathsf{idpath}\ (f\ x)$.

`Lemma` concat_map $A\ B\ (x\ y\ z : A)\ (f : A \to B)\ (p : x \rightsquigarrow y)\ (q : y \rightsquigarrow z) :$
$\mathsf{map}\ f\ (p\ \mathtt{@}\ q) \rightsquigarrow (\mathsf{map}\ f\ p)\ \mathtt{@}\ (\mathsf{map}\ f\ q)$.

`Lemma` opposite_map $A\ B\ (f : A \to B)\ (x\ y : A)\ (p : x \rightsquigarrow y) :$
$\mathsf{map}\ f\ (\mathtt{!}\ p) \rightsquigarrow \mathtt{!}\ \mathsf{map}\ f\ p$.

It is also the case that $\mathsf{map}\ f\ p$ is functorial in f.

`Lemma` idmap_map $A\ (x\ y : A)\ (p : x \rightsquigarrow y) : \mathsf{map}\ (\mathsf{idmap}\ A)\ p \rightsquigarrow p$.

`Lemma` compose_map $A\ B\ C\ (f : A \to B)\ (g : B \to C)\ (x\ y : A)\ (p : x \rightsquigarrow y) :$
$\mathsf{map}\ (g \circ f)\ p \rightsquigarrow \mathsf{map}\ g\ (\mathsf{map}\ f\ p)$.

We can also map paths between paths.

`Definition` map2 $\{A\ B\}\ \{x\ y : A\}\ \{p\ q : x \rightsquigarrow y\}\ (f : A \to B) :$
$p \rightsquigarrow q \to (\mathsf{map}\ f\ p \rightsquigarrow \mathsf{map}\ f\ q)$
$:= \mathsf{map}\ (\mathsf{map}\ f)$.

The type of "homotopies" between two functions f and g is $\forall\ x,\ f\ x \rightsquigarrow g\ x$. These can be derived from "paths" between functions $f \rightsquigarrow g$; 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*.

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.

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.

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).

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

This variant does not call path_simplify.

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*.

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

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.

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.

This tactic simply calls the previous one for the source and the target, repeatedly, until it can no longer make progress.

Now the same thing on the right

This tactic tries to cancel $!p$ on both the left and the right.

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.

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 $!p \mathbin{@} !q \mathbin{@} q$, 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!).

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

Tactics for opposite_map.

Tactics for opposite_concat.

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

Tactics for concat_map.

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

Lemma homotopy_naturality $A\ B\ (f\ g : A \to B)\ (p : \forall\ x, f\ x \rightsquigarrow g\ x)\ (x\ y : A)\ (q : x \rightsquigarrow y) :$
map $f\ q$ @ $p\ y \rightsquigarrow p\ x$ @ map $g\ q$.

Lemma homotopy_naturality_toid $A\ (f : A \to A)\ (p : \forall\ x, f\ x \rightsquigarrow x)\ (x\ y : A)\ (q : x \rightsquigarrow y) :$
map $f\ q$ @ $p\ y \rightsquigarrow p\ x$ @ q.

Lemma homotopy_naturality_fromid $A\ (f : A \to A)\ (p : \forall\ x, x \rightsquigarrow f\ x)\ (x\ y : A)\ (q : x \rightsquigarrow y) :$
q @ $p\ y \rightsquigarrow p\ x$ @ map $f\ q$.

Cancellability of concatenation on both sides.

Lemma concat_cancel_right $A\ (x\ y\ z : A)\ (p\ q : x \rightsquigarrow y)\ (r : y \rightsquigarrow z) : (p$ @ $r \rightsquigarrow q$ @ $r) \to (p \rightsquigarrow q)$.

Lemma concat_cancel_left $A\ (x\ y\ z : A)\ (p : x \rightsquigarrow y)\ (q\ r : y \rightsquigarrow z) : (p$ @ $q \rightsquigarrow p$ @ $r) \to (q \rightsquigarrow r)$.

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 $A\ (f : A \to A)\ (p : \forall\ x, f\ x \rightsquigarrow x)\ (x : A) :$
map $f\ (p\ x) \rightsquigarrow p\ (f\ x)$.

Mates

Lemma concat_moveright_onright $A\ (x\ y\ z : A)\ (p : x \rightsquigarrow z)\ (q : x \rightsquigarrow y)\ (r : z \rightsquigarrow y) :$
$(p \rightsquigarrow q$ @ $!r) \to (p$ @ $r \rightsquigarrow q)$.

Lemma concat_moveleft_onright $A\ (x\ y\ z : A)\ (p : x \rightsquigarrow y)\ (q : x \rightsquigarrow z)\ (r : z \rightsquigarrow y) :$
$(p$ @ $!r \rightsquigarrow q) \to (p \rightsquigarrow q$ @ $r)$.

Lemma concat_moveleft_onleft $A\ (x\ y\ z : A)\ (p : y \rightsquigarrow z)\ (q : x \rightsquigarrow z)\ (r : y \rightsquigarrow x) :$
$(!r$ @ $p \rightsquigarrow q) \to (p \rightsquigarrow r$ @ $q)$.

Lemma concat_moveright_onleft $A\ (x\ y\ z : A)\ (p : x \rightsquigarrow z)\ (q : y \rightsquigarrow z)\ (r : y \rightsquigarrow x) :$
$(p \rightsquigarrow !r$ @ $q) \to (r$ @ $p \rightsquigarrow q)$.

Chapter 3

Library Fibrations

For compatibility with Coq 8.2.

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 total := **sigT**.
Notation tpair := (@existT _ _).
Notation pr1 := (@projT1 _ _).
Notation pr2 := (@projT2 _ _).

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

`Definition` section $\{A\}\ (P : A \to$ `Type`$) := \forall\ x : A,\ P\ x$.

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 $\{A\}\ \{P : A \to$ `Type`$\}\ \{x\ y : A\}\ (p : x \rightsquigarrow y) : P\ x \to P\ y$.

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

`Definition` hfiber $\{A\ B\}\ (f : A \to B)\ (y : B) :=$ `{`x `:` A `&` $f\ x \rightsquigarrow y$`}`.

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.
```

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).
```

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).
```

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.
```

```
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.
```

```
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.
```

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).
```

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.
```

```
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.
```

Transporting along a concatenation is transporting twice.

Lemma trans_concat $\{A\}$ $\{P : A \to \mathsf{Type}\}$ $\{x\ y\ z : A\}$ $(p : x \rightsquigarrow y)$ $(q : y \rightsquigarrow z)$ $(z : P\ x)$:
transport $(p \mathbin{@} q)\ z \rightsquigarrow$ transport q (transport $p\ z$).

Transporting commutes with pulling back along a map.

Lemma map_trans $\{A\ B\}$ $\{x\ y : A\}$ $(P : B \to \mathsf{Type})$ $(f : A \to B)$ $(p : x \rightsquigarrow y)$ $(z : P\ (f\ x))$:
(transport $(P := (\mathtt{fun}\ x \Rightarrow P\ (f\ x)))\ p\ z$) $\rightsquigarrow$ (transport (map $f\ p$) z).

And also with applying fiberwise functions.

Lemma trans_map $\{A\}$ $\{P\ Q : A \to \mathsf{Type}\}$ $\{x\ y : A\}$ $(p : x \rightsquigarrow y)$ $(f : \forall\ x, P\ x \to Q\ x)$ $(z : P\ x)$:
$f\ y$ (transport $p\ z$) $\rightsquigarrow$ (transport p $(f\ x\ z)$).

A version of map for dependent functions.

Lemma map_dep $\{A\}$ $\{P : A \to \mathsf{Type}\}$ $\{x\ y : A\}$ $(f : \forall\ x, P\ x)$ $(p\colon x \rightsquigarrow y)$:
transport p $(f\ x) \rightsquigarrow f\ y$.

Lemma trans_trivial $\{A\ B : \mathsf{Type}\}$ $\{x\ y : A\}$ $(p : x \rightsquigarrow y)$ $(z : B)$:
transport $(P := \mathtt{fun}\ x \Rightarrow B)\ p\ z \rightsquigarrow z$.

Lemma map_dep_trivial $\{A\ B\}$ $\{x\ y : A\}$ $(f : A \to B)$ $(p\colon x \rightsquigarrow y)$:
map_dep $f\ p \rightsquigarrow$ trans_trivial p $(f\ x)$ @ map $f\ p$.

Lemma map_twovar $\{A : \mathsf{Type}\}$ $\{P : A \to \mathsf{Type}\}$ $\{B : \mathsf{Type}\}$ $\{x\ y : A\}$ $\{a : P\ x\}$ $\{b : P\ y\}$
$(f : \forall\ x : A, P\ x \to B)$ $(p : x \rightsquigarrow y)$ $(q :$ transport $p\ a \rightsquigarrow b)$:
$f\ x\ a \rightsquigarrow f\ y\ b$.

Lemma total_path2 $(A : \mathsf{Type})$ $(P : A \to \mathsf{Type})$ $(x\ y :$ sigT $P)$
$(p\ q : x \rightsquigarrow y)$ $(r :$ base_path $p \rightsquigarrow$ base_path $q)$:
(transport $(P := \mathtt{fun}\ s \Rightarrow$ transport s (pr2 x) $\rightsquigarrow$ (pr2 y)) r (fiber_path p) $\rightsquigarrow$ fiber_path q)
$\to (p \rightsquigarrow q)$.

Chapter 4

Library Contractible

For compatibility with Coq 8.2.

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).
```

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).
```

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).
```

The total space of any based path space is contractible.

```
Lemma pathspace_contr {X} (x:X) : is_contr (sigT (paths x)).
Lemma pathspace_contr' {X} (x:X) : is_contr { y:X & x ⇝ y }.
```

The unit type is contractible.

```
Lemma unit_contr : is_contr unit.
```

Chapter 5

Library Equivalences

For compatibility with Coq 8.2.

An equivalence is a map whose homotopy fibers are contractible.

`Definition` is_equiv $\{A\ B\}\ (f : A \to B) := \forall\ y : B$, is_contr (hfiber $f\ y$).

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

`Definition` equiv $A\ B :=$ `{` w `:` $A \to B$ `&` is_equiv w `}`.

Notation "A $\simeq>$ 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 $A\ B\ (w : A \xrightarrow{\sim} B) : (A \to B)$
$:=$ 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.

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 $A : A \xrightarrow{\sim} A$.

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

`Definition` inverse $\{U\ V\}\ (w : U \xrightarrow{\sim} V) : (V \to U) :=$

fun $y \Rightarrow$ 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 $\{U\ V\}$ $(w : U \xrightarrow{\sim} V)$ $y : w\ (w^{-1}\ y) \rightsquigarrow y :=$
pr2 (pr1 ((pr2 w) y)).

Definition inverse_is_retraction $\{U\ V\}$ $(w : U \xrightarrow{\sim} V)$ $x :$ $(w^{-1}\ (w\ x)) \rightsquigarrow x :=$
!base_path (pr2 ((pr2 w) $(w\ x)$) (tpair x (idpath $(w\ x)$))).

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

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

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 $\{A\ B\}$ $(w : A \xrightarrow{\sim} B)$ $x :$
(map w (inverse_is_retraction $w\ x$)) $\rightsquigarrow$ (inverse_is_section w $(w\ x)$).

Equivalences are "injective on paths".

Lemma equiv_injective $U\ V$ $(w : U \xrightarrow{\sim} V)$ $x\ y : (w\ x \rightsquigarrow w\ y) \to (x \rightsquigarrow y)$.

Anything contractible is equivalent to the unit type.

Lemma contr_equiv_unit $(A$: Type) :
is_contr $A \to (A \xrightarrow{\sim}$ **unit**).

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

Lemma contr_equiv_contr $(A\ B$: Type) :
$A \xrightarrow{\sim} B \to$ is_contr $A \to$ is_contr B.

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

Definition free_path_space $A :=$ {xy : $A \times A$ & fst $xy \rightsquigarrow$ snd xy}.

Definition free_path_source A : free_path_space $A \xrightarrow{\sim} A$.

Definition free_path_target A : free_path_space $A \xrightarrow{\sim} A$.

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 $\{A\ B\}$ $(f : A \to B) :=$
{ g : $B \to A$ &

```
  { 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 := (tpair f E) in
    tpair (w^-1) (tpair (inverse_is_section w) (tpair (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.
```

Here is where we use triangle. Now it's just naturality of 'is_section'.

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.
```

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.
```

Now we just play with naturality until things cancel.

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.
```

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.
```

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

```
Definition equiv_compose {A B C} (f : A ⟶~ B) (g : B ⟶~ C) : (A ⟶~ C).
```

```
Definition equiv_cancel_right {A B C} (f : A ⟶~ B) (g : B → C) :
  is_equiv (g ∘ f) → is_equiv g.
```

```
Definition equiv_cancel_left {A B C} (f : A → B) (g : B ⟶~ C) :
```

is_equiv $(g \circ f) \to$ is_equiv f.

```
Definition contr_contr_equiv {A B} (f : A → B) :
  is_contr A → is_contr B → is_equiv f.
```

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).
Theorem equiv_map_is_equiv {A B} {x y : A} (f : A ⟶ B) :
  is_equiv (@map A B x y f).
Definition equiv_map_equiv {A B} {x y : A} (f : A ⟶ B) :
  (x ⇝ y) ⟶ (f x ⇝ f y) :=
  tpair (@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).
Definition concat_equiv_left {A} (x y z : A) (p : x ⇝ y) :
  (y ⇝ z) ⟶ (x ⇝ z) :=
  tpair (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).
Definition concat_equiv_right {A} (x y z : A) (p : y ⇝ z) :
  (x ⇝ y) ⟶ (x ⇝ z) :=
  tpair (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 : total P) :
  (x ⇝ y) ⟶ { p : pr1 x ⇝ pr1 y & transport p (pr2 x) ⇝ pr2 y }.
```

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.
Theorem hiso_to_equiv {A B} (f : A → B) : is_hiso f → is_equiv f.
```

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

Chapter 6

Library FiberEquivalences

For compatibility with Coq 8.2.

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)) :
  total P → total Q.
```

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 : total P) : pr1 z ⇝ pr1 (tg z).
  Defined.

  Let tg_isg_onfibers (z : total P) :
    g _ (transport (tg_is_fiberwise z) (pr2 z)) ⇝ pr2 (tg z).

  Let tg_isfib_onpaths (z w : total P) (p : z ⇝ w) :
    (tg_is_fiberwise z @ base_path (map tg p) @ !tg_is_fiberwise w) ⇝ base_path p.

  Section TotalIsEquiv.

    Hypothesis tot_iseqv : is_equiv tg.

    Let tot_eqv : (total P) ⟶ (total Q) := (tpair tg tot_iseqv).

    Let ginv (x:A) (y: Q x) : P x.

    Theorem fiber_is_equiv (x:A) : is_equiv (g x).
```

```
  Definition fiber_equiv (x:A) : P x ≃→ Q x :=
    tpair (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 := tpair (g x) (fiber_iseqv x).

  Let total_inv : total Q → total P.

  Theorem total_is_equiv : is_equiv tg.

  Definition total_equiv : total P ≃→ total Q :=
    tpair tg (total_is_equiv).

End FiberIsEquiv.

End FiberMap.
```

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 ∘ f.

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

  Let tg := total_map f g.

  Let tginv : total Q → total pbQ.

  Theorem pullback_total_is_equiv : is_equiv tg.

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

End PullbackMap.
```

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).

  Definition fibseq_fiber_equiv :
    is_equiv tg → ∀ x, P x ⟶~ Q (f x) :=
      fun H x ⇒ tpair (g x) (fibseq_fiber_is_equiv H x).

  Let fibseq_a_totalequiv :
    (∀ x, is_equiv (g x)) → (total P ⟶~ total Q).

  Theorem fibseq_total_is_equiv :
    (∀ x, is_equiv (g x)) → is_equiv tg.

  Definition fibseq_total_equiv :
    (∀ x, is_equiv (g x)) → (total P ⟶~ total Q) :=
    fun H ⇒ tpair tg (fibseq_total_is_equiv H).

End FibrationMap.
```

Chapter 7

Library Funext

For compatibility with Coq 8.2.

Definition ext_dep_eq $\{X\}$ $\{P : X \to \mathsf{Type}\}$ $(f\ g : \forall\ x,\ P\ x) := \forall\ x : X,\ f\ x \rightsquigarrow g\ x$.

Notation "f ≡ g" := (ext_dep_eq $f\ g$) (at *level* 50).

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.

Definition funext_statement : Type :=
$\forall\ (X\ Y : \mathsf{Type})\ (f\ g\colon X \to Y),\ f \equiv g \to f \rightsquigarrow g$.

Definition funext_dep_statement : Type :=
$\forall\ (X : \mathsf{Type})\ (P : X \to \mathsf{Type})\ (f\ g : \mathsf{section}\ P),\ f \equiv g \to (f \rightsquigarrow g)$.

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.

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 :=
$\forall\ (X\ Y : \mathsf{Type})\ (f\ g : X \to Y)$, is_equiv (@happly $X\ Y\ f\ g$).

Definition strong_funext_dep_statement : Type :=
$\forall\ (X : \mathsf{Type})\ (P : X \to \mathsf{Type})\ (f\ g : \mathsf{section}\ P)$,
is_equiv (@happly_dep $X\ P\ f\ g$).

Of course, strong functional extensionality implies naive functional extensionality, along with a computation rule.

```
Theorem strong_to_naive_funext :
  strong_funext_statement → funext_statement.

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.

Theorem strong_to_naive_funext_dep :
  strong_funext_dep_statement → funext_dep_statement.

Theorem strong_funext_dep_compute
  (strong_funext_dep : strong_funext_dep_statement)
  (X : Type) (P : X → Type) (f g : section P) (p : f ≡ g) (x : X) :
  happly_dep (strong_to_naive_funext_dep strong_funext_dep X P f g p) x ⇝ p x.
```

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.

Theorem funext_dep_to_nondep :
  funext_dep_statement → funext_statement.
```

Can we go backwards, getting to strong functional extensionality from naive functional extensionality? At first the prospects don't look good; naive functional extensionality gives us a map going backwards, but it doesn't assert anything *about* that map, so it seems unlikely that it would be an inverse to happly.

However, it turns out that we can go backwards; the key is to first forget down to an even weaker axiom, called "weak functional extensionality". This has only one version, which states that the dependent product of a family of (continuously) contractible types is contractible.

```
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.
```

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.
```

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.
```

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

```
Lemma eta_is_equiv : eta_statement → ∀ (A B : Type),
  is_equiv (@eta A B).

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).

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).
```

Less trivial is the fact that weak functional extensionality implies *strong* (dependent) functional extensionality, at least in the presence of the dependent eta rule.

```
Theorem weak_to_strong_funext_dep :
  eta_dep_statement → weak_funext_statement → strong_funext_dep_statement.
```

Of course, naive dependent functional extensionality implies the eta rule.

```
Theorem funext_dep_to_eta_dep : funext_dep_statement → eta_dep_statement.
```

Therefore, strong dependent functional extensionality is equivalent to (weak functional extensionality + dependent eta).

Chapter 8

Library Univalence

For compatibility with Coq 8.2.

Every path between spaces gives an equivalence.

```
Definition path_to_equiv {U V} : (U ⇝ V) → (U ⟶ V).
```

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.
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)).

Lemma opposite_to_inverse {A B} (p : A ⇝ B) :
  (path_to_equiv p)^-1 ⇝ path_to_equiv (!p).
```

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) := (tpair (@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).
```

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).

End Univalence.
```

Chapter 9

Library UnivalenceImpliesFunext

For compatibility with Coq 8.2.

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 : $\forall\ \{A\ B\}\ (w : A \xrightarrow{\sim} B)\ C$,
$(C \to A) \xrightarrow{\sim} (C \to B)$.

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

`Theorem` univalence_implies_funext : funext_statement.

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.

`End` UnivalenceImpliesFunext.

Chapter 10

Library UnivalenceAxiom

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.

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

For compatibility with Coq 8.2.

Some more stuff about contractibility.

```
Theorem contr_contr {X} : is_contr X → is_contr (is_contr X).
```

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).
```

H-levels are increasing with n.

```
Theorem hlevel_succ {n X} : is_hlevel n X → is_hlevel (S n) X.
```

H-level is preserved under equivalence.

```
Theorem hlevel_equiv {n A B} : (A ⟶ B) → is_hlevel n A → is_hlevel n B.
```

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.
Theorem inhabited_contr_isprop {A} : (A → is_contr A) → is_prop A.
Theorem hlevel_isprop {n A} : is_prop (is_hlevel n A).
Definition isprop_isprop {A} : is_prop (is_prop A) := hlevel_isprop.
Theorem prop_equiv_inhabited_contr {A} : is_prop A ⟶ (A → is_contr A).
```

And another one.

```
Theorem prop_path {A} : is_prop A → ∀ (x y : A), x ⇝ y.
Theorem allpath_prop {A} : (∀ (x y : A), x ⇝ y) → is_prop A.
Theorem prop_equiv_allpath {A} : is_prop A ⟶~ (∀ (x y : A), x ⇝ y).
```

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.
Definition axiomK_implies_isset {A} : axiomK A → is_set A.
Theorem isset_equiv_axiomK {A} :
  is_set A ⟶~ (∀ (x : A) (p : x ⇝ x), p ⇝ idpath x).
Definition isset_isprop {A} : is_prop (is_set A) := hlevel_isprop.
Theorem axiomK_isprop {A} : is_prop (axiomK A).
Theorem set_path2 (A : Type) (x y : A) (p q : x ⇝ y) :
  is_set A → (p ⇝ q).
```

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 endohomotopy 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).
```

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.
```

Chapter 12

Library Homotopy

Chapter 13

Library Cohesive

We give the internal definition of a cohesive (∞,1)-topos in the formal language of homotopy type theory.

There is a full subcategory of objects called discrete `Axiom` *is_discrete* : `Type` → `Type`.
`Axiom` *is_discrete_is_prop*: ∀ X , is_prop (*is_discrete* X).

Every object has a reflection into a discrete object. `Axiom` *pi* : `Type` → `Type`.

`Axiom` *pi_is_discrete* : ∀ X , *is_discrete* (*pi* X).
`Axiom` *map_to_pi* : ∀ X , X → *pi* X.

`Axiom` *pi_is_reflection* : ∀ X Y , *is_discrete* Y →
is_equiv (`fun` f: *pi* X → Y ⇒ f ∘ (*map_to_pi* X)).

The reflector preserves the terminal object `Axiom` *pi_preserve_terminal*: is_equiv (*map_to_pi* **unit**).

Every object has a corefection from a discrete object. `Axiom` *flat* : `Type` → `Type`.
Notation `"♭"` := *flat*.

`Axiom` *flat_is_discrete* : ∀ X , *is_discrete* (♭X).
`Axiom` *map_from_flat* : ∀ X , ♭X → X.

`Axiom` *flat_is_coreflection* : ∀ X Y , *is_discrete* Y →
is_equiv (`fun` f : Y → ♭X ⇒ (*map_from_flat* X) ∘ f).

`Axiom` *is_codiscrete* : `Type` → `Type`.
`Axiom` *is_codiscrete_is_prop*: ∀ X , is_prop (*is_codiscrete* X).

`Axiom` *conc* : `Type` → `Type`.

`Axiom` *conc_is_codiscrete* : ∀ X , *is_codiscrete* (*conc* X).
`Axiom` *map_to_conc* : ∀ X , X → *conc* X.

`Axiom` *conc_is_reflection* : ∀ X Y , *is_codiscrete* Y →
is_equiv (`fun` f: *conc* X → Y ⇒ f ∘ (*map_to_conc* X)).