A Coq proof that Univalence Axioms implies Functional Extensionality

Andrej Bauer Peter LeFanu Lumsdaine

March 6, 2011

1 Introduction

This is a self-contained presentation of the proof that the Univalence Axiom implies Functional Extensionality. It was developed by Peter LeFanu Lumsdaine and Andrej Bauer, after a suggestion by Steve Awodey. Peter has his own Coq file with essentially the same proof.

Our proof contains a number of ideas from Voevodsky's proof. Perhaps the most important difference is our use of the induction principle for weak equivalences, see *weq_induction* below. We outline the proof in human language at the end of the file, just before the proof of `extensionality`.

This file is an adaptation of a *small* part of Vladimir Voevodsky's Coq files on homotopy theory and univalent foundations, see https://github.com/vladimirias/Foundations/.

The main difference with Voevodsky's file is rather liberal use of standard Coq tricks, such as notation, implicit arguments and tactics. Also, we are lucky enough to avoid universe inconsistencies. Coq is touchy-feely about universes and one unfortunate definition seems to be enough to cause it to encounter a universe inconsistency. In fact, an early version of this file encountered a universe inconsistency in the last line of the main proof. By removing some auxiliary defnitions, we managed to make it go away.

This file also contains extensive comments about Coq. This is meant to increase its instructional value.

2 Basic definitions

Suppose A is a space and $P : A \to$ `Type` is a map from A to spaces. We can think of P as a family of spaces indexed by A. Actually, P should be thought of as a fibration, because in the intended interpretation dependent types correspond to fibrations.

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 of course is just Coq's dependent sum construction, which is 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*.

Coq is very picky about what point belong to what space. For example, if $x : A$ is a point in the base space and $y : P\ x$ is a point in the fiber $P\ x$ then, mathematically speaking, y is also a point in the total space *sigT P*. But Coq wants explicit notation for y as a point of *sigT P*. In Coq this is written as *existT P x y*.

Given a point p : *sigT P* of the total space, we can reconstruct the corresponding base point as *projT1 p* and the point in the fiber as *projT2 p*.

We proceed to the definition of the identity map. The notation `fun` $x \Rightarrow e$ is Coq's way of writing a map which takes x to e.

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

The definition below defines composition of functions. The curly braces around $A\ B\ C$ means that A, B and C are *implicit* arguments. This means that we do not have to write them because Coq will compute them (from the types of g and f). If we insist, we can specify the implicit arguments. For example *compose* $(A{:=}X)$ $(B{:=}Y)$ f g means that A is set to X and B to Y. By writing @*compose* we get a version of *compose* without explicit arguments. So instead of writing *compose* $(A{:=}X)$ $(B{:=}Y)$ $(C{:=}Z)$ f g it is better to write @*compose* X Y Z f g. But for the most part the implicit arguments notation is extremely convenient. In the rare cases when Coq cannot figure out what the implicit arguments are, it tells us so.

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

In Coq it is possible to define all sorts of new notations. One should not exaggerate with strange notations, it can be quite convenient to define special notation for commonly used notions.

We define special notation $g\ o\ f$ for composition of functions. The *printing* comment (which is not visible after coqdoc has processed it) tells how to display the notation in LaTeX.

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

3 Paths

Next we define the space of paths between two given points. This is a central concept in homotopy theory.

`Inductive` *paths* $\{A\}$: $A \rightarrow A \rightarrow$ `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).

`Hint Resolve` @*idpath*.

The definition of *paths* requires an explanation. The *idpath* clause tells Coq that for every $x : A$ there is a path *idpath* x, which we think of as the identity path from x to x.

Furthermore, because *paths* is defined as an inductive type, Coq automatically generates an associated induction principle *paths_rect* (as well as some other variant induction principles which we ignore here). The induction principle is a bit complicated (type "`Print` *paths_rect*." to see it), so let us explain what it says. Suppose P is a fibration over *paths* in A, i.e., for any two points $x\ y : A$ and a path $p : x \rightsquigarrow y$ we have a space $P\ x\ y\ p$. Now suppose u is an element of $P\ x\ x$ (*idpath* x). The induction principle *paths_rect* allows us to conclude that u can be transported to an element of $P\ x\ y\ p$.

Another way of reading the induction principle is as follows. Suppose P is a property of maps. Then in order to show that P holds of all paths $p : x \rightsquigarrow y$ it is sufficient to check that P holds of the identity paths *idpath* $x : x \rightsquigarrow x$.

A special case of the induction principle *paths_rect* happens when the fibration P depends just on the points of A rather than paths in A: a point $u : P\ x$ in the fiber over $x : A$ can be transported along a path $p : x \rightsquigarrow y$ in the base to the fiber $P\ y$. See the *transport* theorem below.

If we read $x \rightsquigarrow y$ as "x and y are equal" then the transport along paths in the base becomes the logical principle of *substitution of equals for equals*: if $P\ x$ holds (as witnessed by u) and $x \rightsquigarrow y$ then $P\ y$ holds. We shall not comment further on this double reading of *paths* which reaveals a fascinating connection between homotopy theory and logic.

A typical use of *paths_rect* is as follows. Suppose we want to construct a point v in some space $V\ x\ y\ p$ which depends on a path $p : x \rightsquigarrow y$. We employ the Coq `induction` tactic (which applies *paths_rect*) to a hopefully easier problem of constructing a point u of the space $V\ x\ x$ (*idpath* x).

We now prove some basic fact about paths.

Paths can be concatenated.

```
Definition concat {A} {x y z : A} : (x ⇝ y) → (y ⇝ z) → (x ⇝ z).
Proof.
  intros 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 $x \leadsto z$ as a concatenation of paths $x \leadsto y \leadsto z$. 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 p.
  induction p.
  apply idpath.
Defined.
```

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

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

In the previous two proofs we always used the same proof strategy: apply induction on paths and then apply idpath. Such tactics can be automated in Coq, as shown below where a new tactic *path_induction* is defined. 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 [... ⊢ ...] is a pattern for contexts. To the left of the symbol ⊢ 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`.

Next we show basic properties of paths and composition of paths. Note that all statements are "up to homotopy", e.g., the composition of p and the identity path is not equal to p but only conntected to it with a path. We call paths between paths *homotopies.* The following lemmas should be self-explanatory.

```
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_opposite A (x y : A) (p : x ⇝ y) : !(! p) ⇝ p.
Proof.
  path_induction.
Defined.
```

We place the lemmas just proved into the `Hint Resolve` database so that `auto` will know about them.

```
Hint Resolve
  idpath_left_unit idpath_right_unit
  opposite_right_inverse opposite_left_inverse.
```

```
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.
```

Homotopies of concatenable maps can be concatenated.

```
Definition homotopy_concat 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.
```

A path $p : x \leadsto y$ in a space A is mapped by $f : A \to B$ to a map *map* $f\ p : f\ x \leadsto f\ y$ in B. Note that we cannot transfer p by just composing it with f because p is *not* a function. Instead, we use the induction principle.

```
Lemma map {A B} {x y : A} (f : A → B) (p : 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.
```

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 composition_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.
```

Other facts about map.

```
Lemma opposite_map A B (f : A → B) (x y : A) (p : x ↝ y) : ! (map f p) ↝ map f (! p).
Proof.
  path_induction.
Defined.

Lemma map_cancel A B (f : A → B) (x y : A) (p q : x ↝ y) : p ↝ q → (map f p ↝ map f q).
Proof.
  intro h.
  path_induction.
Defined.
```

So far *path_induction* has worked beautifully, but we are soon going to prove more complicated theorems which require smarter tactics, so we define some.

This time we first declare some `Hint Resolve` hints, but notice that we put them 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
  concat_map
  opposite_map map_cancel
  opposite_concat opposite_opposite
  homotopy_concat : path_hints.
```

By the way, we can add more hints to the database later.

Next we define a simple strategy which tries a number of more complicated lemmas and uses the first one that makes progress (that is the meaning of the *first* keyword), after which it performs an `auto` using the *path_hints* database.

```
Ltac path_tricks :=
  first
  [ apply homotopy_concat
  | apply opposite_map
  | apply opposite_opposite
  | apply opposite_concat
  | apply map_cancel
  | idtac] ; auto with path_hints.
```

The *path_via* x tactic is used to construct a path $a \rightsquigarrow b$ as a composition of paths $a \rightsquigarrow x$ and $x \rightsquigarrow b$. It also applies the *path_tricks* to help user get rid of the easy cases.

```
Ltac path_via x := apply @concat with (y := x); path_tricks.
```

Here are several more facts about *map* which have slightly more involved proofs. We use the just defined tactics. The proofs a little too manual, obviously we need even better tactics which will allow us to argue about paths as if they were equalities.

```
Lemma map_naturality 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.
  path_via (p x).
  apply idpath_left_unit.
  apply opposite; apply idpath_right_unit.
Defined.
```

```
Hint Resolve map_naturality : path_hints.
```

```
Lemma homotopy_natural 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.
  path_via (p x).
  path_via (idpath (f x) @ p x).
  path_via (p x @ idpath (g x)).
 apply opposite; auto.
Defined.
```

```
Lemma concat_cancel_right A (x y z : A) (p q : x ⇝ y) (r : y ⇝ z) : p @ r ⇝ q @ r → p ⇝ q.
Proof.
  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 → q ⇝ r.
Proof.
  intro a.
  induction p.
  induction r.
  path_via (idpath x @ q).
  apply opposite; auto.
Defined.
```

```
Lemma concat_move_over_left A (x y z : A) (p : x ⇝ z) (q : x ⇝ y) (r : y ⇝ z) :
  p ⇝ q @ r → p @ !r ⇝ q.
Proof.
  intro a.
  apply concat_cancel_right with (r := r).
  path_via (p @ (!r @ r)).
  apply concat_associativity.
  path_via (p @ idpath z).
  path_via p.
Defined.
```

```
Lemma endomap_homotopy_commute A (f : A → A) (p : ∀ x, f x ⇝ x) (x : A) : map f (p x) ⇝ p (f x).
Proof.
  path_via (map f (p x) @ (p x @ !p x)).
  path_via (map f (p x) @ idpath (f x)); apply opposite; auto.
  path_via ((map f (p x) @ p x) @ !p x).
  apply opposite; apply concat_associativity.
  apply concat_move_over_left.
  apply @concat with (y := (p (f x) @ map (idmap A) (p x))).
  apply homotopy_natural with (g := idmap A).
  apply homotopy_concat; auto.
  apply idmap_map.
Defined.
```

```
Lemma map_action A (f : A → A) (p : ∀ x, f x ⇝ x) (y z : A) (q : f z ⇝ y) :
  map f (p z) @ q ⇝ map f q @ p y.
Proof.
  path_via (p (f z) @ q).
  apply endomap_homotopy_commute.
  apply opposite; apply map_naturality.
Defined.
```

4 Homotopy between maps

There are two senses in which maps f and g can be "homotopic":

- homotopic: there is a path $f \rightsquigarrow g$, or
- pointwise homotopic: for each x in the domain of f there is a path $f\ x \rightsquigarrow g\ x$.

Let us verify that "homotopic" implies "pointwise homotopic".

```
Lemma happly {A B} {f g : A → B}: (f ⇝ g) → (∀ x, f x ⇝ g x).
Proof.
  path_induction.
Defined.
```

The converse of *happly* is known as *extensionality of maps* in type theory and cannot be proved without further assumptions.

At first sight it seems clear from a topological point of view that extensionality should fail: for maps f, g : $A \to B$ to be homotopic it is *not* sufficient to know that there is a path $p\ x : f\ x \rightsquigarrow g\ x$ for every $x : A$ (consider the unit circle in the complex plane with f and g the identity and conjugation, respectively). In fact, the notion "pointwise homotopic" does not seem to be a very good one, topologically speaking.

However, since we are in a setting where *all* maps are continuous, the assignment p of paths $p\ x : f\ x \rightsquigarrow g\ x$ itself is continuous in x, which *is* sufficient to conclude that f and g are homotopic!

The main point of this file is the proof that the Univalence axioms implies extensionality for maps.

We move on to study how paths interact with fibrations. Let us first verify 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.
```

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

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).
Proof.
  path_induction.
Defined.
```

5 Basic homotopy notions

Just like in the case of homotopy of maps, there are two possible definitions of a contractible space:

- a space A is contractible if there is a path from the identity map on A to a constant map on A, or
- a space A is contractible if there is a point $x : A$ and for every $y : A$ a path $y \rightsquigarrow x$.

The pointwise version is the more useful one. An element of *contractible* A is a pair whose first component is a point x and the second component is a pointwise retraction of A to x.

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

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

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

```
Ltac contract_hfiber y p :=
  match goal with
    | [ ⊢ contractible (@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 *contractible* (*hfiber* f x), where the question marks in

?*f* and ?*x* are pattern variables that Coq should match against the actualy 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.

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.
  induction p.
  path_via q.
  path_via (!(idpath (f x)) @ q).
  path_via (idpath (f x) @ q).
  apply opposite; auto.
Defined.
```

6 Weak equivalences

A weak equivalence is a map whose homotopy fibers are contractible.

```
Definition is_wequiv {A B} (f : A → B) := ∀ y : B, contractible (hfiber f y).
```

wequiv A B is the space of weak equivalences from *A* to *B*.

```
Definition wequiv A B := { w : A → B & is_wequiv w }.
```

Strictly speaking, an element *w* of *wequiv A B* is a *pair* consisting of a map *projT1 w* and the proof *projT2 w* that it is a weak 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 *wequiv A B* to *A* → *B*.

```
Definition wequiv_coerce_to_function : ∀ A B, wequiv A B → (A → B).
Proof.
  intros A B w.
  exact (projT1 w).
Defined.
```

```
Coercion wequiv_coerce_to_function : wequiv ¿-¿ Funclass.
```

The identity map is a weak equivalence.

```
Definition idweq A : wequiv A A.
Proof.
  ∃ (idmap A).
  intros x.
  contract_hfiber x (idpath x).
  apply total_paths with (p := q).
  simpl.
  compute in q.
  path_induction.
Defined.
```

Every path between spaces gives a weak equivalence.

```
Definition path_to_weq {U V} : U ⇝ V → wequiv U V.
Proof.
  intro p.
  induction p as [S].
  exact (idweq S).
```

```
Defined.
```

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

```
Definition weq_inv {U V} : wequiv U V → (V → U).
Proof.
  intros [w H] y.
  destruct (H y) as [[x p] _].
  exact x.
Defined.
```

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

```
Lemma weq_inv_is_section U V (w : wequiv U V) : ∀ y : V, w (weq_inv w y) ⇝ y.
Proof.
  intro y.
  destruct w as [w G].
  simpl.
  destruct (G y) as [[x p] c].
  exact p.
Defined.
```

```
Lemma weq_inv_is_retraction U V (w : wequiv U V) : ∀ x : U, (weq_inv w (w x)) ⇝ x.
Proof.
  intro x.
  destruct w as [w H].
  simpl.
  destruct (H (w x)) as [[y p] c].
  assert (q := c (existT _ x (idpath (w x)))).
  assert (r := base_path q).
  exact (!r).
Defined.
```

The last general fact about weak equivalences that we need is that they are injective on paths, which is not too surprising, given that they have sections.

```
Lemma weq_injective U V : ∀ (w : wequiv U V) x y, w x ⇝ w y → x ⇝ y.
Proof.
  intros w x y.
  simpl.
  intro p.
  assert (q := map (weq_inv w) p).
  path_via (weq_inv w (w x)).
  apply opposite; apply weq_inv_is_retraction.
  path_via (weq_inv w (w y)).
  apply weq_inv_is_retraction.
Defined.
```

7 Univalence implies Extensionality

At this point we start using the Univalence Axiom. It states that the *path_to_weq* map which turns paths into weak equivalences is itself a weak equivalence.

```
Axiom univalence : ∀ U V, is_wequiv (@path_to_weq U V).
```

The axioms allows us to go in the other direction: every weak equivalence yields a path.

```
Definition weq_to_path {U V} : wequiv U V → U ⇝ V.
```

```
Proof.
  apply weq_inv.
  ∃ (@path_to_weq U V).
  apply univalence.
Defined.
```

The map *weq_to_path* is a section of *path_to_weq*.

```
Lemma weq_to_path_section U V : ∀ (w : wequiv U V), path_to_weq (weq_to_path w) ⇝ w.
Proof.
  intro w.
  exact (weq_inv_is_section _ _ (existT _ (@path_to_weq U V) (univalence U V)) w).
Defined.
```

We can do better than *weq_to_path*, we can turn a fibration fibered by weak equivalences to one fiberered over paths.

```
Definition pred_weq_to_path U V : (wequiv U V → Type) → (U ⇝ V → Type).
Proof.
  intros Q p.
  apply Q.
  apply path_to_weq.
  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 weak equivalences. In the proof we use *pred_weq_to_path* to transport the predicate *P* of weak equivalences to a predicate *P'* on paths. Then we use path induction and transport back to *P*.

```
Theorem weq_induction (P : ∀ U V, wequiv U V → Type) :
  (∀ T, P T T (idweq T)) → (∀ U V (w : wequiv U V), P U V w).
Proof.
  intro r.
  pose (P' := (fun U V ⇒ pred_weq_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 (weq_to_path_section _ _ w)).
  exact (paths_rect _ P' r' U V (weq_to_path w)).
Defined.
```

We should strive to make the following lemma shorter. The lemma states that a map which is pointwise homotopic to the identity is a weak equivalence.

```
Lemma weq_pointwise_idmap A (f : A → A) : (∀ x, f x ⇝ x) → is_wequiv f.
Proof.
  intros p y.
  contract_hfiber y (p y).
  apply total_paths with (p := ! (p z) @ q).
  simpl.
  eapply concat.
  apply transport_hfiber.
  path_via (map f (!q @ p z) @ q).
  path_via (map f (! (!p z @ q))).
  eapply concat.
  path_tricks.
  path_tricks.
  path_via ((map f (!q) @ map f (p z)) @ q).
  path_via (map f (!q) @ (map f (p z) @ q)).
  apply concat_associativity.
  path_via (map f (!q) @ (map f q @ p y)).
  apply map_action.
  path_via ((map f (!q) @ map f q) @ p y).
  apply opposite; apply concat_associativity.
  path_via (idpath (f y) @ p y).
  path_via (! map f q @ map f q).
  apply opposite; apply opposite_map.
Defined.
```

We need one more axiom, which is about eta-expansion of functions.

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

Axiom eta_axiom : ∀ {A B} (h : A → B), eta h ⇝ h.
```

The eta axiom essentially states that *eta* is a weak equivalence.

```
Theorem etaweq A B : wequiv (A → B) (A → B).
Proof.
  ∃ (@eta A B).
  apply weq_pointwise_idmap.
  apply eta_axiom.
Defined.
```

Another important ingridient in the proof of extensionality is the fact that exponentiation preserves weak equivalences, i.e., if w is a weak equivalence then post-composition by w is again a weak equivalence.

```
Theorem weq_exponential : ∀ {A B} (w : wequiv A B) C, wequiv (C → A) (C → B).
Proof.
  intros A B w C.
  ∃ (fun h ⇒ w ∘ h).
  generalize A B w.
  apply weq_induction.
  intro D.
  apply (projT2 (etaweq C D)).
Defined.
```

We are almost ready to prove extensionality, but first we need to show that the source and target maps from the total space of maps are weak equivalences.

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

Definition src A : wequiv (path_space A) A.
Proof.
  ∃ (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 trg A : wequiv (path_space A) A.
```

```
Proof.
  ∃ (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.
```

And finally, we are ready to prove that extensionality of maps holds, i.e., if two maps are pointwise homotopic then they are homotopic. First we outline the proof.

Suppose maps $f\ g : A \to B$ are extensionally equal via a pointwise homotopy p. We seek a path $f \rightsquigarrow g$. Because *eta* $f \rightsquigarrow f$ and *eta* $g \rightsquigarrow g$ it suffices to find a path *eta* $f \rightsquigarrow$ *eta* g.

Consider the maps $d\ e : S \to$ *path_space* T where $d\ x =$ *existT* _ $(f\ x, f\ x)$ (*idpath* x) and $e\ x =$ *existT* _ $(f\ x, g\ x)$ $(p\ x)$. If we compose d and e with *trg* we get *eta* f and *eta* g, respectively. So, if we had a path from d to e, we would get one from *eta* f to *eta* g. But we can get a path from d to e because *src* $\circ\ d =$ *eta* $f =$ *src* $\circ\ e$ and composition with *src* is an equivalence.

```
Theorem extensionality {A B : Set} (f g : A → B) : (∀ x, f x ⇝ g x) → (f ⇝ g).
Proof.
  intro p.
  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 := weq_exponential (src B) A).
  pose (trg_compose := weq_exponential (trg B) A).
  apply weq_injective with (w := etaweq A B).
  simpl.
  path_via (projT1 trg_compose e).
  path_via (projT1 trg_compose d).
  apply map.
  apply weq_injective with (w := src_compose).
  apply idpath.
Defined.
```

And that is all, thank you.