(** "Homotopy type theory and Voevodsky's univalent foundations" Companion File *)

(** By Alvaro Pelayo and Michael A. Warren. August, 2012.*)

(** This file, which is based very closely on Voevodsky's own files, is meant to accompany the paper "Homotopy type theory and Voevodsky's univalent foundations" and is intended to be a more accessible introduction to Voevodsky's Coq library. As such, we attempt, wherever possible, to follow Voevodsky in his choice of notation and terminology except where we have decided (rightly or wrongly) that changes are in the interest of exposition. Whenever possible we try to indicate any serious divergence from Voevodsky's library.*)

(** Section numbers and titles in comments correspond to those in the paper.*)

(** * Section 3: BASIC COQ CONSTRUCTIONS *)

(** ** Section 3.2: Types and terms in Coq. *)

(** We let UU denote the universe of small types. We think of small types as small (in the sense of Grothendieck universes or large cardinals) spaces.*)

```
Definition UU := Type.
```

(** Coq secretly assigns indices to each Type which occurs in an internally consistent way. To play around with this a bit, you might uncomment the following code and run it through Coq to see what you get back in the *response* buffer: *)

```
(**
Check Type.
Set Printing Universes.
Check Type.
Unset Printing Universes.
*)
```

(** ** Section 3.3: A direct definition involving function spaces. *)

(** Here is an example of a direct definition:*)

```
Definition idfun ( A : UU ) : A -> A := fun x => x.
```

(** We can now play around a little bit with the type checking mechanism of Coq:*)

```
Section idfun_test.

Variable A : UU.
Variable a : A.

Check idfun A.
Check idfun _ a.

End idfun_test.

(** Uncomment the following line to see what the response is.  Be sure to re-
comment it (or delete it) once you're done, otherwise Coq won't go any further.*)

(** Check A. *)



(** ** Section 3.4: An indirect definition involving function spaces. *)

(** Let's now look at two alternative versions of the same definition.  In the first, we
will use Coq tactics to construct the desired term.  In the second, we give a direct
definition.*)

Definition funcomp_indirect { A B C : UU } ( f : A -> B ) ( g : B -> C ) : A -> C.
Proof.
  intro x. apply g. apply f. assumption.
Defined.

Definition funcomp { A B C : UU } ( f : A -> B ) ( g : B -> C ) := fun x : A => g ( f x ).

Print funcomp.

Print funcomp_indirect.



(** * Section 4: SOME BASIC INDUCTIVE TYPES *)



(** ** Section 4.1: The inductive type of natural numbers. *)

Print nat.

(** Note that nat has the type Set here!  Using the patches employed by Voevodsky it
is possible to define nat in such a way that it ends up in UU, but this is not really
necessary for us.  I.e., with the patches, you should try:*)
```

```
(*
Inductive nat' : UU := 0' : nat' | S' : nat' -> nat'.
Print nat'.
*)

(** We will now define the predecessor function as the canonical function nat -> nat
which sends 0 to 0 and (n+1) to n:*)

Definition predecessor ( n : nat ) : nat :=
  match n with
    | 0 => 0
   | S m => m
  end.

(** We can test this out by asking Coq to compute the predecessor of 26:*)

Eval compute in predecessor 26.

(** It is worth considering an indirect definition of the predecessor function, as it
exhibits some new tactics:*)

Definition indirect_predecessor ( n : nat ) : nat.
Proof.
  destruct n. exact 0. exact n.
Defined.

Print predecessor.
Print indirect_predecessor.



(** ** Section 4.2: Fibrations and the total space of a fibration. *)

(** We view dependent types as maps B -> UU and then the following inductive type
"total" corresponds to the dependent sum.  From the homotopical perspective, this is
the total space of the fibration.  From the category theoretic perspective, this is the
infinity groupoid obtained by performing the Grothendieck construction on the map B
-> UU.*)

Inductive total { B : UU } ( E : B -> UU ) : UU := pair ( x : B ) ( y : E x ).
Implicit Arguments pair [ B E ].

Definition dirprod ( A B : UU ) : UU := total ( fun x : A => B ).
Definition dirprodpair { A B : UU } : A -> B -> dirprod A B  := fun x y => pair x y.

(** In Voevodsky's files, total is called total2 and is defined as what is called a "record
type".  This is convenient for practical reasons (and is mathematically equivalent),
but in order to keep the ammount of Coq machinery to a minimum we instead define
this type as an inductive type.*)
```

```
Definition pr1 { B : UU } { E : B -> UU } : total E -> B := fun z => match z with pair x y
=> x end.
Definition pr2 { B : UU } { E : B -> UU } ( s : total E ) : E ( pr1 s ) := match s with pair x
y => y end.




(** * Section 5: THE PATH SPACE *)

Notation paths := identity.

Print identity.

Notation idpath := identity_refl.



(** ** Section 5.1: Groupoid structure of the path space. *)

(** Given f : paths a b, pathsinv(f) : paths b a is the inverse of f.  This is called
pathsinv0 in Voevodsky's files.*)

Definition pathsinv { A : UU } { a b : A } ( f : paths a b ) : paths b a.
Proof.
  destruct f. apply idpath.
Defined.

(** pathscomp f g gives the composite g.f of two paths f and g. This is called
pathscomp0 in Voevodsky's files.*)

Definition pathscomp { A : UU } { a b c : A } ( f : paths a b ) ( g : paths b c ) : paths a c.
Proof.
  destruct f. assumption.
Defined.

(** The left-unit law holds on the nose.  The right-unit law for composition only holds
up to the existence of a higher-dimensional path.  This is the following lemma.  In
Voevodsky's files this is the term pathscomp0rid.*)

Lemma isrunitalpathscomp { A : UU } { a b : A } ( f : paths a b ) : paths ( pathscomp f
( idpath b ) ) f.
Proof.
  destruct f. apply idpath.
Defined.
```

```
Lemma isassocpathscomp { A : UU } { a b c d : A } ( f : paths a b ) ( g : paths b c )
( h : paths c d ) : paths ( pathscomp ( pathscomp f g ) h ) ( pathscomp f ( pathscomp
g h ) ).
Proof.
  destruct f. apply idpath.
Defined.

(** The following is the same as pathsinv0l in Voevodsky's files:*)

Lemma islinvpathsinv { A : UU } { a b : A } ( f : paths a b ) : paths ( pathscomp
( pathsinv f ) f ) ( idpath _ ).
Proof.
  destruct f. apply idpath.
Defined.

(** The following is the same as pathsinv0r in Voevodsky's files:*)

Lemma isrinvpathsinv { A : UU } { a b : A } ( f : paths a b ) : paths ( pathscomp f
( pathsinv f ) ) ( idpath _ ).
Proof.
  destruct f. apply idpath.
Defined.


(** ** Section 5.2: The functorial action of a continuous map on a path. *)

(** A continuous function f : A -> B has a (weak infinity) functorial action on paths in
A.  This action is coded by "maponpaths": *)

Definition maponpaths { A B : UU } ( f : A -> B ) { a a' : A } ( p : paths a a' ) : paths ( f
a ) ( f a' ).
Proof.
  destruct p. apply idpath.
Defined.

Notation "f ` p" := ( maponpaths f p ) (at level 30 ).

(** That it is so easy to define this functorial action is in contrast to the situation in
many approaches to higher-dimensional category theory where one must first
describe the appropriate (and usually highly non-trivial) combinatorial conditions that
an infinity-functor must satisfy before being able to every start working with them!*)




(** * Section 6: TRANSPORT *)
```

```
Definition transportf { B : UU } ( E : B -> UU ) { b b' : B } ( f : paths b b' ) : E b -> E b'.
Proof.
  intros e. destruct f. assumption.
Defined.



(** ** Section 6.1: Homotopy and homotopy equivalence. *)


(** We define the type of homotopies between two continuous functions f,g : A -> B.
*)

Definition homot { A B : UU } ( f g : A -> B ) := forall x :A, paths ( f x ) ( g x ) .

Definition isheq { A B : UU } ( f : A -> B ) := total ( fun f' : B -> A => dirprod ( homot
( funcomp f' f ) ( idfun _ ) ) ( homot ( funcomp f f' ) ( idfun _ ) ) ).



(** ** Section 6.2: Forward and backward transport. *)


Definition transportb { B : UU } ( E : B -> UU ) { b b' : B } ( f : paths b b' ) : E b' -> E b.
Proof.
  intros e. destruct f. assumption.
Defined.

Lemma backandforth { B : UU }{ E : B -> UU }{ b b' : B } ( f : paths b b' ) : homot
( funcomp ( transportb E f ) ( transportf E f ) ) ( idfun _ ).
Proof.
  intros x. destruct f. apply idpath.
Defined.

Lemma forthandback { B : UU }{ E : B -> UU }{ b b' : B } ( f : paths b b' ) : homot
( funcomp ( transportf E f ) ( transportb E f ) ) ( idfun _ ).
Proof.
  intros x. destruct f. apply idpath.
Defined.

Lemma isheqtransportf { B : UU } ( E : B -> UU ) { b b' : B } ( f : paths b b' ) : isheq
( transportf E f ).
Proof.
  split with ( transportb E f ). split.
  apply backandforth. apply forthandback.
Defined.



(** ** Section 6.3: Paths in the total space.*)
```

```
Lemma pathintotalfiber { B : UU } { E : B -> UU } { x y : total E } ( f : paths ( pr1 x )
( pr1 y ) ) ( g : paths ( transportf E f ( pr2 x ) ) ( pr2 y ) ) : paths x y.
Proof.
  intros. destruct x as [ x0 x1 ]. destruct y as [ y0 y1 ].  simpl in *. destruct f. destruct
g. apply idpath.
Defined.

Definition pathintotalfiberpr1 { B : UU } { E : B -> UU } { x y : total E } ( f : paths x y ) :
paths ( pr1 x ) ( pr1 y ) := pr1 ` f.

Definition pathintotalfiberpr2 { B : UU } { E : B -> UU } { x y : total E } ( f : paths x y ) :
paths ( transportf E ( pathintotalfiberpr1 f ) ( pr2 x ) ) ( pr2 y ).
Proof.
  intros. destruct f. apply idpath.
Defined.

Lemma pathintotalfibercharacterization { B : UU } { E : B -> UU } { x y : total E } ( f :
paths x y ) : paths f  ( pathintotalfiber ( pathintotalfiberpr1 f ) ( pathintotalfiberpr2 f ) ).
Proof.
  intros. destruct f. destruct x as [ x0 x1 ]. apply idpath.
Defined.




(** * Section 7: WEAK EQUIVALENCES AND HOMOTOPY EQUIVALENCES *)



(** ** Section 7.1: Contractibility. *)

Definition iscontr ( A : UU ) := total ( fun center : A => forall b : A, paths center b ).

Lemma iscontrretract { A B : UU } { r : A -> B } { s : B -> A } ( p : homot ( funcomp s r )
( idfun _ ) ) ( is : iscontr A ) : iscontr B.
Proof.
  split with ( r ( pr1 is ) ).  intros b. change b with ( idfun B b ).
  rewrite <- ( p b ). unfold funcomp. apply maponpaths. apply ( pr2 is ).
Defined.

Definition iscontrandheq { A B : UU } { f : A -> B } ( p : isheq f ) ( is : iscontr A ) :
iscontr B := iscontrretract ( pr1 ( pr2 p ) ) is.

Definition iscontrandheqinv { A B : UU } { f : A -> B } ( p : isheq f ) ( is : iscontr B ) :
iscontr A := iscontrretract (pr2 ( pr2 p ) ) is.
```

```
(** ** Section 7.2: Homotopy fibers. *)

Definition hfiber { A B : UU } ( f : A -> B ) ( b : B ) : UU := total ( fun x : A => paths ( f
x ) b ).

Definition hfiberact { A B C : UU }{ f : A -> B }{ g : C -> B } ( h : A -> C ) ( p : homot
( funcomp h g ) f ) ( b : B ) : hfiber f b -> hfiber g b := fun a => pair ( h ( pr1 a ) )
(  pathscomp ( p ( pr1 a ) ) ( pr2 a ) ).

Definition maponhfiber { A B C : UU } ( f : A -> B ) ( g : B -> C ) ( c : C ) : hfiber
( funcomp f g ) c -> hfiber g c := fun a => pair ( f ( pr1 a ) ) ( pr2 a ).

Definition secmaponhfiber { A B : UU }{ r : A -> B }{ s : B -> A } ( p : homot ( funcomp
s r ) ( idfun _ ) ) ( a : A ) : hfiber s a -> hfiber ( funcomp r s ) a := fun b => pair ( s ( pr1
b) ) ( pathscomp ( s ` ( p ( pr1 b ) ) ) ( pr2 b ) ).

Lemma transportfandhfiber { A B : UU }{ f : A -> B }{ a a' : A }{ b : B } ( p : paths a
a' ) ( i : paths ( f a ) b ) : paths ( transportf ( fun x : A => paths ( f x ) b ) p i )
( pathscomp ( pathsinv ( f ` p ) ) i ).
Proof.
  destruct p. apply idpath.
Defined.

Lemma secmaponhfiberissec { A B : UU }{ r : A -> B }{ s : B -> A } ( p : homot
( funcomp s r ) ( idfun _ ) ) ( a : A ) : homot ( funcomp  ( secmaponhfiber p a )
( maponhfiber r s a ) ) ( idfun _ ).
Proof.
  intros b. destruct b as [ b i ]. unfold funcomp, idfun in *. simpl.
  apply ( @pathintotalfiber B ( fun x : B => paths ( s x ) a ) ( pair ( r ( s b ) )
( pathscomp ( s ` ( p b ) ) i ) ) ( pair b i ) ( p b ) ).
  rewrite transportfandhfiber. unfold secmaponhfiber. simpl.
  rewrite <- isassocpathscomp. rewrite islinvpathsinv. apply idpath.
Defined.

Definition hfiberandhomot { A B : UU }{ f g : A -> B } ( b : B ) ( p : homot f g ) : hfiber f
b -> hfiber g b := fun a => pair ( pr1 a ) ( pathscomp ( pathsinv ( p ( pr1 a ) ) ) ( pr2
a ) ).

Definition hfiberandhomotinv { A B : UU }{ f g : A -> B } ( b : B ) ( p : homot f g ) :
hfiber g b -> hfiber f b := fun a => pair ( pr1 a ) ( pathscomp ( ( p ( pr1 a ) ) ) ( pr2
a ) ).

Lemma isheqhfiberandhomot { A B : UU }{ f g : A -> B } ( b : B ) ( p : homot f g ) :
isheq ( hfiberandhomot b p ).
Proof.
  split with ( hfiberandhomotinv b p ). split. intros a. unfold funcomp, idfun,
hfiberandhomot, hfiberandhomotinv. simpl. destruct a as [ a i ]. simpl.
  apply ( @pathintotalfiber A ( fun x : A => paths ( g x ) b ) ( pair a  _ ) ( pair a i )
( idpath a ) ).
  simpl. rewrite <- isassocpathscomp. rewrite islinvpathsinv. apply idpath.
  intros a. unfold funcomp, idfun, hfiberandhomot, hfiberandhomotinv. destruct a as
[ a i ]. simpl.
  apply ( @pathintotalfiber A ( fun x : A => paths ( f x ) b ) ( pair a _ ) ( pair a i )
( idpath a ) ). simpl.
  rewrite <- isassocpathscomp. rewrite isrinvpathsinv. apply idpath.
Defined.



(** ** Section 7.3: Weak equivalences. *)

Definition isweq { A B : UU } ( f : A -> B ) := ( forall b : B, iscontr ( hfiber f b ) ).

Definition weq ( A B : UU ) := total ( fun f : A -> B => isweq f ).

Definition isweqidfun ( A : UU ) : isweq ( idfun A ).
Proof.
  intros x. split with ( @pair _ ( fun z => paths ( ( idfun A ) z ) x ) x ( idpath x ) ). intros
y.
  destruct y as [ y y' ]. destruct y'. apply idpath.
Defined.

Definition idweq ( A : UU ) := pair ( idfun A ) ( isweqidfun A ).

Definition weqpreimage { A B : UU }{ f : A -> B } ( is : isweq f ) ( b : B ) : A := pr1 ( pr1
( is b ) ).

Definition weqpreimageeq { A B : UU }{ f : A -> B } ( is : isweq f ) ( b : B ) : paths ( f
( weqpreimage is b ) ) b := pr2 ( pr1 ( is b ) ).

Definition weqpreimageump1 { A B : UU }{ f : A -> B } ( is : isweq f ) ( b : B ) { a : A }
( g : paths ( f a ) b ) : paths ( weqpreimage is b ) a.
Proof.
  intros. change ( paths ( pr1 ( @pair _ ( fun z => paths ( f z ) b ) ( weqpreimage is b )
( weqpreimageeq is b )  ) ) ( pr1 ( @pair _ ( fun z => paths ( f z ) b ) a g ) ) ). apply
pathintotalfiberpr1.
  unfold weqpreimage. unfold weqpreimageeq. destruct ( is b ) as [ l r ]. simpl.
destruct l as [ l l' ]. apply r.
Defined.

Lemma weqpreimageump2 { A B : UU }{ f : A -> B } ( is : isweq f ) { b b' : B } ( g :
paths ( weqpreimage is b ) ( weqpreimage is b' ) ) : paths b b'.
Proof.
  rewrite <- ( weqpreimageeq is b ). rewrite <- ( weqpreimageeq is b' ). rewrite g.
apply idpath.
Defined.
```

```
(** ** Section 7.4: Weak equivalences and homotopy equivalences. *)

Definition weqinv { A B : UU } ( f : weq A B ) : B -> A := fun x => weqpreimage ( pr2 f ) 
x.

Lemma weqinvandidweq ( A : UU ) : homot ( weqinv ( idweq A ) ) ( idfun A ).
Proof.
  intros x. unfold weqinv.
  change ( weqpreimage ( pr2 ( idweq A ) ) x ) with ( idfun A ( weqpreimage ( pr2 
( idweq A ) ) x ) ).
  apply weqpreimageeq.
Defined.

Lemma weqinvislinv { A B : UU } ( f : weq A B ) : homot ( funcomp ( weqinv f ) ( pr1 
f ) ) ( idfun _ ).
Proof.
  intros x. apply weqpreimageeq.
Defined.

Lemma weqinvisrinv { A B : UU } ( f : weq A B ) : homot ( funcomp ( pr1 f ) ( weqinv 
f ) ) ( idfun _ ).
Proof.
  intros x. apply weqpreimageump1. apply idpath.
Defined.

Lemma weqtoheq { A B : UU }{ f : A -> B } ( is : isweq f ) : isheq f.
Proof.
  split with ( weqinv ( pair f is ) ). split. apply weqinvislinv. apply weqinvisrinv.
Defined.

Lemma iscontrhfiberandhretract { A B : UU }{ r : A -> B }{ s : B -> A } ( p : homot 
( funcomp s r ) ( idfun _ ) ) ( a : A ) : iscontr ( hfiber ( funcomp r s ) a ) -> iscontr 
( hfiber s a ).
Proof.
  intros is. apply ( @iscontrretract ( hfiber ( funcomp r s ) a ) ( hfiber s a ) 
( maponhfiber _ _ _ ) ( secmaponhfiber p a ) ). apply secmaponhfiberissec. 
assumption.
Defined.

Lemma iscontrhfiberandhomot { A B : UU }{ f f' : A -> B } ( h : homot f f' ) ( b : B ) : 
iscontr ( hfiber f' b ) -> iscontr ( hfiber f b ).
Proof.
  intros is. apply ( iscontrandheqinv ( isheqhfiberandhomot b h ) ). assumption.
Defined.

Theorem gradth { A B : UU }{ f : A -> B } ( is : isheq f ) : isweq f.
Proof.
  intro b. destruct is as [ f' is ].
  apply ( @iscontrhfiberandhretract B A f' f  ( pr2 is ) ).
  apply ( @iscontrhfiberandhomot B _ ( funcomp f' f ) ( idfun B ) ( pr1 is ) ).
  apply isweqidfun.
Defined.

Corollary weqinvisweq { A B : UU } ( f : weq A B ) : isweq ( weqinv f ).
Proof.
  apply gradth. split with ( pr1 f ). split. apply weqinvisrinv. apply weqinvislinv.
Defined.





(** * Section 8: THE UNIVALENCE AXIOM AND SOME CONSEQUENCES *)


(** ** We now turn to the univalence axiom and some of its consequences. This
section requires a patched version of Coq in order to compile.*)



(** ** Section 8.1: An alternative characterization of the Univalence Axiom. *)

Definition eqweqmap { A B : UU } ( p : paths A B ) : weq A B := match p with idpath
=> ( idweq _ ) end.


(** The univalence axiom states that eqweqmap is itself a weak equivalence.  We will
describe an equivalent principle. *)

Definition isweqeqweqmap := forall A B : UU, isweq ( @eqweqmap A B ).

Definition weqeqmap ( univ : isweqeqweqmap ) { A B : UU } ( f : weq A B ) :=
( weqinv ( pair eqweqmap ( univ A B ) ) ) f.

Lemma weqeqmaprinv ( univ : isweqeqweqmap ) { A B : UU } ( f : paths A B ) : paths
( weqeqmap univ ( eqweqmap f ) ) f.
Proof.
  unfold weqeqmap.
  change ( paths ( ( funcomp ( pr1 ( pair eqweqmap ( univ A B ) ) ) ( weqinv ( pair _
( univ A B ) ) )  ) f ) ( ( idfun ( paths A B ) ) f ) ).
  apply weqinvisrinv.
Defined.

Lemma weqeqmaplinv ( univ : isweqeqweqmap ) { A B : UU } ( f : weq A B ) : paths
( eqweqmap ( weqeqmap univ f ) ) f.
Proof.
  unfold weqeqmap.
  change ( paths ( ( funcomp ( weqinv ( pair _ ( univ A B ) ) ) ( pr1 ( pair eqweqmap
( univ A B ) ) ) ) f ) ( ( idfun ( weq A B ) ) f ) ).
  apply weqinvislinv.
Defined.

Lemma weqeqmapcomp ( univ : isweqeqweqmap ) ( A : UU ) : paths ( weqeqmap
univ ( idweq A ) ) ( idpath _ ).
Proof.
  change ( idweq A ) with ( eqweqmap ( idpath A ) ). apply weqeqmaprinv.
Defined.

Definition weqindelim := forall E : total ( fun x : UU => total ( fun y : UU => weq x y ) )
-> UU, forall p : ( forall x : UU, E ( pair x ( pair x ( idweq x ) ) ) ), forall x y : UU, forall
z : weq x y, E ( pair x ( pair y z ) ).

Definition weqindcomp ( rec : weqindelim ) := forall  E : total ( fun x : UU => total ( fun
y : UU => weq x y ) ) -> UU, forall p : ( forall x : UU, E ( pair x ( pair x ( idweq x ) ) ) ),
forall x : UU, paths ( rec E p x x ( idweq x ) ) ( p x ).

Lemma weqind0 ( E : total ( fun x : UU => total ( fun y : UU => weq x y ) ) -> UU )
( p : ( forall x : UU, E ( pair x ( pair x ( idweq x ) ) ) ) ) : ( forall x y : UU, forall z : paths
x y, E ( pair x ( pair y ( eqweqmap z ) ) ) ).
Proof.
  intros A. destruct z. apply p.
Defined.

Definition weqind ( univ : isweqeqweqmap ) : weqindelim := fun E p A B f =>
transportf ( fun z => E ( pair A ( pair B z ) ) ) ( weqeqmaplinv univ f ) ( weqind0 E p A
B (weqeqmap univ f) ).

Lemma weqind0lemma ( E : total ( fun x : UU => total ( fun y : UU => weq x y ) ) ->
UU ) ( p : ( forall x : UU, E ( pair x ( pair x ( idweq x ) ) ) ) ) ( A B : UU ) ( f g : paths A
B ) ( q : paths f g ) : paths ( weqind0 E p A B f ) ( transportb ( fun z : paths A B => E
( pair A ( pair B ( eqweqmap z ) ) ) ) q ( weqind0 E p A B g ) ).
Proof.
  destruct q. apply idpath.
Defined.

(** It can be shown that homotopy inverses of weak equivalences can be chosen to
satisfy the triangle equalities without using the univalnce axiom.  However, as this
lemma plays only a minor role here we will give the shorter proof using univalence
for the one trinagle equation we require. *)

Lemma triangleidweq { A : UU } ( x : A ) : paths ( weqinvislinv ( idweq A ) x ) ( ( idfun
A ) ` ( weqinvisrinv  (idweq A ) ) x ).
Proof.
  apply idpath.
Defined.

Lemma triangle ( univ : isweqeqweqmap ) { A B : UU } ( f : weq A B ) ( a : A ) : paths
( weqinvislinv f ( ( pr1 f ) a ) ) ( ( pr1 f ) ` ( weqinvisrinv f a ) ).
```

```
Proof.
  set ( E := fun z : total ( fun x : UU => total ( fun y : UU => weq x y ) ) => forall v :
( pr1 z ), paths ( weqinvislinv ( pr2 ( pr2 z ) ) ( ( pr1 ( pr2 ( pr2 z ) ) ) v ) ) ( ( pr1 ( pr2
( pr2 z ) ) ) ` ( weqinvisrinv ( pr2 ( pr2 z ) ) v ) ) ).
  apply ( weqind univ E ( fun A x => idpath _ ) A B f ).
Defined.

Lemma eqweqmaptriangle ( univ : isweqeqweqmap ) { A B : UU } ( f : paths A B ) :
paths ( weqeqmaplinv univ ( eqweqmap f ) ) ( eqweqmap ` weqeqmaprinv univ f ).
Proof.
  destruct f. change eqweqmap with ( pr1 ( pair eqweqmap ( univ A A ) ) ).
  exact ( ( triangle univ ( pair eqweqmap ( univ A A ) ) ( ( idpath A ) ) ) ).
Defined.

Lemma transportbandmop ( A B : UU ) ( E : B -> UU ) ( f : A -> B ) ( a a' : A ) ( p :
paths a a' ) : paths ( transportb E ( f ` p ) ) ( transportb ( funcomp f E ) p ).
Proof.
  destruct p. apply idpath.
Defined.

Lemma weqeqmapandtransportb ( univ : isweqeqweqmap ) ( A : UU ) ( E : weq A A
-> UU )  : paths (transportb (fun z : paths A A => E (eqweqmap z)) (weqeqmaprinv
univ (idpath A))) (transportb (fun z : weq A A => E z ) (weqeqmaplinv univ (idweq A))).
Proof.
  change ( idweq A ) with ( eqweqmap ( idpath A ) ). rewrite eqweqmaptriangle.
  rewrite ( transportbandmop ( paths A A ) ( weq A A ) ( fun z : weq A A => E z )
eqweqmap _ _ ( weqeqmaprinv univ ( idpath A ) ) ).
  apply idpath.
Defined.

Definition pathtohtpysec { B : UU } { E : B -> UU } ( f g : forall x : B, E x ) : paths f g ->
( forall x, paths ( f x ) ( g x ) ) := fun z => match z with idpath => ( fun x => idpath _ )
end.

Definition pathtohtpy { A B : UU } { f g : A -> B } : paths f g -> homot f g :=
@pathtohtpysec A ( fun z => B ) f g.

Lemma weqcomp ( univ : isweqeqweqmap ) : weqindcomp ( weqind univ ).
Proof.
  intros E p A. unfold weqind.
  assert ( paths ( weqind univ E p A A ( idweq A ) ) ( ( funcomp ( transportb ( fun z :
weq A A => E ( pair A ( pair A z ) ) ) ( weqeqmaplinv univ ( idweq A ) ) ) ( transportf
( fun z : weq A A => E ( pair A ( pair A z ) ) ) ( weqeqmaplinv univ ( idweq A ) ) ) ) ( p
A ) ) ) as g.
  unfold weqind.  unfold funcomp. apply maponpaths.
  rewrite ( weqind0lemma E p A A (weqeqmap univ (idweq A)) (idpath A)
(weqeqmapcomp univ A) ).
  unfold weqeqmapcomp.
  apply ( pathtohtpy ( weqeqmapandtransportb univ A ( fun z => E ( pair A ( pair A
z ) ) ) ) ).
```

```
  rewrite ( @backandforth ( weq A A ) ( fun z : weq A A => E ( pair A ( pair A z ) ) ) _ _
(weqeqmaplinv univ (idweq A)) ) in g.
  unfold idfun in g. apply g.
Defined.

Definition rweqeqmap ( rec : weqindelim ) ( A B : UU ) : weq A B -> paths A B.
Proof.
  set ( E := fun z : total ( fun x : UU => total ( fun y : UU => weq x y ) ) => paths ( pr1
z ) ( pr1 ( pr2 z ) ) ).
  assert ( forall x : UU, E ( pair x ( pair x ( idweq x ) ) ) ) as p.
  intros. apply idpath. apply ( rec E p ).
Defined.

Lemma rweqeqmapcomp { rec : weqindelim } ( reccomp : weqindcomp rec ) ( A :
UU ) : paths ( rweqeqmap rec A A ( idweq A ) ) ( idpath A ).
Proof.
  apply ( reccomp (fun z : total (fun x : UU => total (fun y : UU => weq x y)) =>
        paths (pr1 z) (pr1 (pr2 z))) ).
Defined.

Lemma rweqeqislinv { rec : weqindelim } ( reccomp : weqindcomp rec ) { A B : UU }
( f : weq A B ) : paths ( eqweqmap ( rweqeqmap rec A B f ) ) f.
Proof.
  set ( E := fun z : total ( fun x : UU => total ( fun y : UU => weq x y ) ) => paths
( eqweqmap ( rweqeqmap rec ( pr1 z ) ( pr1 ( pr2 z ) ) ( pr2 ( pr2 z ) ) ) ) ( pr2 ( pr2
z ) ) ).
  assert ( forall x : UU, E ( pair x ( pair x ( idweq x ) ) ) ) as p. intros C. unfold E. simpl.
  rewrite ( rweqeqmapcomp reccomp ). apply idpath.
  apply ( rec E p A B f).
Defined.

Lemma rweqeqisrinv { rec : weqindelim } ( reccomp : weqindcomp rec ) { A B : UU }
( f : paths A B ) : paths ( rweqeqmap rec A B ( eqweqmap f ) ) f .
Proof.
  destruct f. simpl. rewrite ( rweqeqmapcomp reccomp ). apply idpath.
Defined.

Lemma isheqeqweqmap { rec : weqindelim } ( reccomp : weqindcomp rec ) ( A B :
UU ) : isheq ( @eqweqmap A B ).
Proof.
  split with ( rweqeqmap rec A B ). split. intros f. unfold funcomp, idfun. apply
( rweqeqislinv reccomp ).
  intros f. unfold funcomp, idfun. apply ( rweqeqisrinv reccomp ).
Defined.

Theorem univfromind { rec : weqindelim } ( reccomp : weqindcomp rec ) :
isweqeqweqmap.
Proof.
  intros A B. apply gradth. apply ( isheqeqweqmap reccomp ).
Defined.
```

```
(** ** Secction 8.2: Function extensionality. *)

Axiom univ : isweqeqweqmap.

Axiom funextsec : forall B : UU, forall E : B -> UU, forall f g : forall x : B, E x, isweq
( pathtohtpysec f g ).

Definition funextfun { A B : UU } ( f g : A -> B ) : homot f g -> paths f g := weqinv ( pair
_ ( funextsec A ( fun z => B ) f g ) ).



(** ** Section 8.3: Impredicativity of h-levels. *)

Fixpoint isofhlevel ( n : nat ) ( A : UU ) :=
  match n with
    | 0 => iscontr A
    | S n => forall a b : A, isofhlevel n ( paths a b )
  end.

Lemma impredbase { B : UU } ( E : B -> UU ) : ( forall x : B, iscontr ( E x ) ) -> iscontr
( forall x : B, E x ).
Proof.
  intros is. split with ( fun x => ( pr1 ( is x ) ) ). intros p.
  apply funextsec. intros x. apply ( pr2 ( is x ) ).
Defined.

Lemma impred ( n : nat ) : forall B : UU, forall E : B -> UU, ( forall x : B, isofhlevel n
( E x ) ) -> isofhlevel n ( forall x : B, E x ).
Proof.
  induction n. intros B E is. apply ( impredbase E is ).
  intros B E is. change ( forall f g : ( forall x : B, E x ), isofhlevel n ( paths f g ) ).
  intros f g. rewrite ( weqeqmap univ ( pair ( pathtohtpysec f g )  ( funextsec B E f
g ) ) ).
  apply ( IHn  ). intros x. apply ( is x ).
Defined.



(** ** Section 8.4: The total space and h-levels. *)

Definition pathintotalfibertototalspace { B : UU }{ E : B -> UU } ( x y : total E ) : paths
x y -> total ( fun v : paths ( pr1 x ) ( pr1 y ) => paths ( transportf E v ( pr2 x ) ) ( pr2
y ) ) := fun f => pair ( pathintotalfiberpr1 f ) ( pathintotalfiberpr2 f ).

Lemma pathintotalfiberpr1andpitf { B : UU }{ E : B -> UU } ( x y : total E )  ( a : paths
(pr1 x) (pr1 y) ) ( a' : paths (transportf E a (pr2 x)) (pr2 y) ) : paths (pathintotalfiberpr1
(pathintotalfiber a a')) a.
Proof.
  destruct x as [ x x' ]. destruct y as [ y y' ]. simpl in *. destruct a. destruct a'. apply
idpath.
Defined.

Lemma isweqpitftototalspace { B : UU } { E : B -> UU } ( x y : total E ) : isweq
( pathintotalfibertototalspace x y ).
Proof.
  apply gradth. split with ( fun f : total ( fun v : paths ( pr1 x ) ( pr1 y ) => paths
( transportf E v ( pr2 x ) ) ( pr2 y ) ) => pathintotalfiber ( pr1 f ) ( pr2 f ) ). split. intros a.
unfold funcomp, idfun.
  unfold pathintotalfibertototalspace.  destruct a as [ a a' ].  simpl.
  apply ( @pathintotalfiber _ ( fun v : paths ( pr1 x ) ( pr1 y ) => paths ( transportf E v
( pr2 x ) ) ( pr2 y ) ) ( pair (pathintotalfiberpr1 (pathintotalfiber a a'))
      (pathintotalfiberpr2 (pathintotalfiber a a'))) ( pair a a' ) ( pathintotalfiberpr1andpitf
_ _ a a' ) ). simpl.
  destruct x as [ x x' ]. destruct y as [ y y' ].  simpl in a. destruct a. simpl in a'.  destruct
a'. apply idpath.
  intros a. unfold funcomp, idfun. unfold pathintotalfibertototalspace. destruct a as [ a
a' ]. simpl.
  rewrite pathintotalfibercharacterization. apply idpath.
Defined.

Lemma totalandhlevel ( n : nat ) : forall B : UU, forall E : B -> UU, forall is : isofhlevel
n B, forall is' : forall x : B, isofhlevel n ( E x ) , isofhlevel n ( total E ).
Proof.
  induction n. intros. split with ( pair ( pr1 is ) ( pr1 ( is' ( pr1 is ) ) ) ).
  intros e. destruct e as [ b e ]. apply ( @pathintotalfiber _ E ( pair ( pr1 is ) _ ) ( pair b
e ) ( ( pr2 is ) b ) ).
  simpl. rewrite <- ( ( pr2 ( is' b ) ) ( transportf E ( pr2 is b ) ( pr1 ( is' ( pr1 is ) ) ) ) ).
  rewrite <- ( ( pr2 ( is' b ) ) e ). apply idpath.
  intros. change ( forall x y : total E, isofhlevel n ( paths x y ) ).
  intros x y. rewrite ( weqeqmap univ ( pair _ ( isweqpitftototalspace x y ) ) ).
  apply IHn. apply is. intros f. apply ( is' ( pr1 y ) ).
Defined.



(** ** Section 8.5: The unit type and contractibility. *)

(** The unit type unit is already built in to Coq as the inductive type with a single
generator tt : unit. *)

Definition tounit ( A : UU ) : A -> unit := fun x => tt.

Lemma tounitandiscontr { A : UU } ( is : iscontr A ) : isweq ( tounit A ).
Proof.
  apply gradth. destruct is as [ center is ]. split with ( fun x => center ).
  split. intros x. destruct x. apply idpath. intros x. apply is.
```

```
Defined.

Lemma iscontrunit : iscontr unit.
Proof.
  split with tt. intros b. destruct b. apply idpath.
Defined.

Lemma isweqtounitpaths ( a b : unit ) : isweq ( tounit ( paths a b ) ).
Proof.
  apply gradth. split with ( fun x => pathscomp ( pathsinv ( pr2 ( iscontrunit ) a ) )
( ( pr2 ( iscontrunit ) ) b ) ). destruct a, b. split. intros c. destruct c. apply idpath.
intros c. destruct c. apply idpath.
Defined.

Lemma tounitisweqtoiscontr { A : UU } ( is : isweq ( tounit A ) ) : iscontr A.
Proof.
  split with ( weqinv ( pair _ is ) tt ).
  intros a. change tt with ( tounit A a ).
  exact ( weqinvisrinv ( pair _ is ) a ).
Defined.

Definition isaprop ( A : UU ) := isofhlevel 1 A.

Lemma isapropunit : isaprop unit.
Proof.
  intros x y. apply tounitisweqtoiscontr. apply ( isweqtounitpaths x y ).
Defined.

Definition iscontrunittounitinv : unit -> iscontr unit.
Proof.
  intros x. split with x. intros b. destruct x, b. apply idpath.
Defined.

Lemma unitpathssectionunique ( s t : forall b : unit, paths tt b ) : paths s t.
Proof.
  apply funextsec. intros x. rewrite <- ( ( pr2 ( isapropunit tt x ) ) ( s x ) ).
  rewrite <- ( ( pr2 ( isapropunit tt x ) ) ( t x ) ). apply idpath.
Defined.

Lemma isweqtounitiscontrunit : isweq ( tounit ( iscontr unit ) ).
Proof.
  apply gradth. split with iscontrunittounitinv. split. intros x. destruct x.
  apply idpath. intros x. destruct x as [ x is ]. destruct x.
  unfold funcomp, idfun. change (tounit (iscontr unit) (pair tt is)) with tt.
  apply ( @pathintototalfiber _ _ ( pair tt _ ) ( pair tt is ) ( idpath tt ) ).
  apply unitpathssectionunique.
Defined.

Lemma iscontrcontr { A : UU } ( is : iscontr A ) : iscontr ( iscontr A ).
Proof.
```

```
  rewrite ( weqeqmap univ ( pair _ ( tounitandiscontr is ) ) ).
  rewrite ( weqeqmap univ ( pair _ ( isweqtounitiscontrunit ) ) ).
  apply iscontrunit.
Defined.



(** ** Section 8.6: Some propositions. *)

Lemma iscontrtoisaprop { A : UU } ( is : iscontr A ) : isaprop A.
Proof.
   rewrite ( weqeqmap univ ( pair _ ( tounitandiscontr is ) ) ).
   apply isapropunit.
Defined.

Lemma isapropiscontr ( A : UU ) : isaprop ( iscontr A ).
Proof.
  intros x y. apply iscontrtoisaprop. apply iscontrcontr. assumption.
Defined.

Lemma isapropisweq { A B : UU } ( f : A -> B ) : isaprop ( isweq f ).
Proof.
  unfold isweq.  apply impred. intros b. apply isapropiscontr.
Defined.

Theorem isaxiomunivalence : isaprop ( isweqeqweqmap ).
Proof.
  apply impred. intros A. apply impred. intros B. apply isapropisweq.
Defined.



(** ** Section 8.7: The h-levels of h-universes. *)

Lemma isofhlevelntoSn ( n : nat ) : forall A : UU, isofhlevel n A -> isofhlevel ( S n ) A .
Proof.
  induction n. intros A is. apply iscontrtoisaprop. assumption.
  intros A is. change ( forall x y : A, isofhlevel ( S n ) ( paths x y ) ).
  intros a b. apply IHn. apply is.
Defined.

Lemma iscontrtoisofhleveln ( n : nat ) : forall A : UU, iscontr A -> isofhlevel n A.
Proof.
  induction n. intros A is. assumption. intros A is. change ( forall x y : A, isofhlevel n
( paths x y ) ). intros a b. apply IHn. apply iscontrtoisaprop. assumption.
Defined.

Lemma isaproptoisofhlevelSn ( n : nat ) : forall A : UU, isaprop A -> isofhlevel ( S n )
A.
Proof.
```

```
  induction n. intros A is. assumption. intros A is. change ( forall x y : A, isofhlevel ( S
n ) ( paths x y ) ). intros a b. apply IHn. apply iscontrtoisofhleveln.
  apply is.
Defined.

Lemma hlevelweqcodomain ( n : nat ) : forall A B : UU, isofhlevel ( S n ) B ->
isofhlevel ( S n ) ( weq A B ).
Proof.
  induction n. intros A B is. unfold weq. apply totalandhlevel. apply impred. intros.
apply is. intros. apply isapropisweq.
  intros A B is. apply totalandhlevel. apply impred. intros. apply is. intros.
  apply isaproptoisofhlevelSn. apply isapropisweq.
Defined.

Definition weqsymmap ( A B : UU ) : weq A B -> weq B A := fun f => pair _
( weqinvisweq f ).

Lemma weqinvweqinv { A B : UU } ( f : weq A B ) : paths ( weqinv ( pair _
( weqinvisweq f ) ) ) ( pr1 f ).
Proof.
  apply funextfun. intros x. destruct f as [ f is ]. simpl. unfold weqinv. apply
weqpreimageump1.
  simpl. apply weqpreimageump1. apply idpath.
Defined.

Lemma isweqweqsymmmap ( A B : UU ) : isweq ( weqsymmap A B ).
Proof.
  apply gradth. split with ( weqsymmap B A ). split. intros f. unfold funcomp, idfun.
  destruct f as [ f is ].
  unfold weqsymmap. apply ( @pathintotalfiber _ _ ( pair _ (weqinvisweq (pair
(weqinv (pair f is)) (weqinvisweq (pair f is)))) ) ( pair f is ) ( weqinvweqinv ( pair _
is ) ) ). apply isapropisweq.
  intros f. unfold funcomp, idfun.
  destruct f as [ f is ].
  unfold weqsymmap. apply ( @pathintotalfiber _ _ ( pair _ (weqinvisweq (pair
(weqinv (pair f is)) (weqinvisweq (pair f is)))) ) ( pair f is ) ( weqinvweqinv ( pair _
is ) ) ). apply isapropisweq.
Defined.

Lemma hlevelweqdomain ( n : nat ) : forall A B : UU, isofhlevel ( S n ) A -> isofhlevel
( S n ) ( weq A B ).
Proof.
  intros A B is. rewrite ( weqeqmap univ ( pair _ ( isweqweqsymmmap A B ) ) ). apply
hlevelweqcodomain. assumption.
Defined.

Lemma isapropisofhleveln ( n : nat ) : forall A : UU, isaprop ( isofhlevel n A ).
Proof.
  induction n. intros A. apply isapropiscontr. intros A.
  apply impred. intros x. apply impred. intros y. apply IHn.
```

```
Defined.

Definition hlevelneqweqmap ( n : nat ) ( A B : total ( fun x : UU => isofhlevel n x ) ) :
paths A B -> weq ( pr1 A ) ( pr1 B ) := fun f => eqweqmap ( pathintotalfiberpr1 f ) .

Definition hlevelnweqeqmap ( n : nat ) ( A B : total ( fun x : UU => isofhlevel n x ) ) :
weq ( pr1 A ) ( pr1 B ) -> paths A B.
Proof.
  intros f. apply ( pathintotalfiber ( ( weqeqmap univ ) f ) ). apply isapropisofhleveln.
Defined.

Lemma hlevelnweqeqmapcomp ( n : nat ) ( A : total ( fun x : UU => isofhlevel n x ) ) :
paths ( pathintotalfiberpr1 ( hlevelnweqeqmap n A A ( idweq ( pr1 A ) ) ) )
( pathintotalfiberpr1 ( idpath A ) ).
Proof.
  destruct A as [ A is ].
  unfold hlevelnweqeqmap. rewrite pathintotalfiberpr1andpitf.
  rewrite weqeqmapcomp. apply idpath.
Defined.

Lemma hlevelnweqeqmapatidweq ( n : nat ) ( A : UU ) ( is : isofhlevel n A ) ( is' :
isofhlevel n A ): paths ( hlevelneqweqmap n ( @pair  UU ( fun x : UU => isofhlevel n
x ) A is ) ( @pair  UU ( fun x : UU => isofhlevel n x ) A is' ) ( hlevelnweqeqmap n
( @pair  UU ( fun x : UU => isofhlevel n x ) A is ) ( @pair  UU ( fun x : UU =>
isofhlevel n x ) A is' ) ( idweq A ) ) ) ( idweq A ).
Proof.
  unfold hlevelneqweqmap. unfold hlevelnweqeqmap. rewrite
pathintotalfiberpr1andpitf. apply weqeqmaplinv.
Defined.

Lemma hlevelnweqeqmaprinv ( n : nat ) ( A B : total ( fun x : UU => isofhlevel n x ) )
( f : weq ( pr1 A ) ( pr1 B ) ) : paths ( hlevelneqweqmap n A B ( hlevelnweqeqmap n A
B f ) ) f.
Proof.
  destruct A as [ A is ]. destruct B as [ B is' ].
  set ( E := fun z : ( total ( fun x : UU => total ( fun y : UU => weq x y ) ) ) => forall is :
isofhlevel n ( pr1 z ), forall is' : isofhlevel n ( pr1 ( pr2 z ) ), paths ( hlevelneqweqmap
n ( @pair UU ( fun x : UU => isofhlevel n x ) ( pr1 z ) is ) ( @pair UU ( fun x : UU =>
isofhlevel n x ) ( pr1 ( pr2 z ) ) is' ) ( hlevelnweqeqmap n ( @pair UU ( fun x : UU =>
isofhlevel n x ) ( pr1 z ) is ) ( @pair UU ( fun x : UU => isofhlevel n x ) ( pr1 ( pr2 z ) )
is' ) ( pr2 ( pr2 z ) ) )  ) ( pr2 ( pr2 z ) ) ).
  apply ( ( weqind univ E ( fun x => fun iss => fun iss' => hlevelnweqeqmapatidweq n
x iss iss' ) ) A B f is is' ).
Defined.

Lemma pathpathintotalsamefiber { B : UU }{ E : B -> UU }{ b0 b1 : B }{ e0 : E b0 }
{ e1 : E b1 } ( f : paths b0 b1 ) ( p q : paths ( transportf E f e0 ) e1 ) ( alpha : paths p
q ) : paths ( @pathintotalfiber B E ( pair b0 e0 ) ( pair b1 e1 ) f p ) ( @pathintotalfiber
B E ( pair b0 e0 ) ( pair b1 e1 ) f q ).
Proof.
```

```
  intros. destruct f.  destruct alpha.  apply idpath.
Defined.

Lemma pathpathintotalfiber0 { B : UU }{ E : B -> UU }{ b0 b1 : B }{ e0 : E b0 }{ e1 :
E b1 } ( f0 f1 : paths b0 b1 ) ( g0 : paths ( transportf E f0 e0 ) e1 ) ( g1 : paths
( transportf E f1 e0) e1 ) ( alpha0 : paths f0 f1 ) ( alpha1 : paths ( transportf ( fun z :
paths b0 b1 => ( paths ( transportf E z e0 ) e1 ) ) alpha0 g0 ) g1 ) : paths
( @pathintotalfiber B E ( pair b0 e0 ) ( pair b1 e1 ) f0 g0 ) ( @pathintotalfiber B E
( pair b0 e0 ) ( pair b1 e1 ) f1 g1 ).
Proof.
 induction alpha0.  apply pathpathintotalsamefiber. assumption.
Defined.

Lemma pathpathintotalfiber { B : UU }{ E : B -> UU }{ x y : total E } ( p q : paths x y )
( alpha0 : paths ( pathintotalfiberpr1 p ) ( pathintotalfiberpr1 q ) )  ( alpha1 : paths
(  transportf ( fun z : paths ( pr1 x ) ( pr1 y ) => paths ( transportf E z ( pr2 x ) ) ( pr2
y ) ) alpha0 ( pathintotalfiberpr2 p ) ) ( pathintotalfiberpr2 q )  ) : paths p q.
Proof.
  destruct x as [ b0 e0 ]. destruct y as [ b1 e1 ]. rewrite
( pathintotalfibercharacterization p ). rewrite ( pathintotalfibercharacterization q ).
  apply ( ( @pathpathintotalfiber0 B E b0 b1 e0 e1 (pathintotalfiberpr1 p) )
( pathintotalfiberpr1 q ) ( pathintotalfiberpr2 p ) ( pathintotalfiberpr2 q ) alpha0
alpha1 ).
Defined.

Lemma hlevelnweqeqmaplinv ( n : nat ) ( A B : total ( fun x : UU => isofhlevel n x ) )
( f : paths A B ) : paths ( hlevelnweqeqmap n A B ( hlevelneqweqmap n A B f ) ) f.
Proof.
  destruct f. destruct A as [ A is ]. unfold hlevelneqweqmap.
  change ( pathintotalfiberpr1 ( idpath ( pair A is ) ) ) with ( idpath A ).
  change ( eqweqmap ( idpath A ) ) with ( idweq A ).
  assert ( paths ( pathintotalfiberpr1 (hlevelnweqeqmap n (@pair UU ( fun x : UU =>
isofhlevel n x ) A is) ( @pair UU ( fun x : UU => isofhlevel n x ) A is) (idweq A)) )
( pathintotalfiberpr1 ( idpath ( pair A is ) ) ) ) as i.
  apply hlevelnweqeqmapcomp.
  apply ( pathpathintotalfiber (hlevelnweqeqmap n (@pair UU ( fun x : UU =>
isofhlevel n x ) A is) (@pair UU ( fun x : UU => isofhlevel n x ) A is) (idweq A)) (idpath
(@pair UU ( fun x : UU => isofhlevel n x ) A is)) i ).
  set ( j := ( ( isapropisofhleveln n A ) (transportf (isofhlevel n) (pathintotalfiberpr1
(idpath (pair A is))) (pr2 (pair A is))) (pr2 (pair A is)) ) ).
  assert ( iscontr ( paths (transportf (fun z : paths (pr1 (pair A is)) (pr1 (pair A is)) =>
paths (transportf (fun x : UU => isofhlevel n x) z (pr2 (pair A is))) (pr2 (pair A is))) i
(pathintotalfiberpr2 (hlevelnweqeqmap n (@pair UU ( fun x : UU => isofhlevel n x ) A
is) (@pair UU ( fun x : UU => isofhlevel n x ) A is) (idweq A)))) (pathintotalfiberpr2
(idpath (pair A is))) ) ) as p.
  assert ( isaprop ( paths (transportf (isofhlevel n) (pathintotalfiberpr1 (idpath (pair A
is))) (pr2 (pair A is))) (pr2 (pair A is)) ) ) as p0.
  apply iscontrtoisaprop. apply isapropisofhleveln. apply p0.
  apply p.
Defined.
```

```
Lemma isweqhlevelneqweqmap ( n : nat ) : forall A B : total ( fun x : UU => isofhlevel
n x ), isweq ( hlevelneqweqmap n A B ).
Proof.
  intros A B. apply gradth. split with ( hlevelnweqeqmap n A B ).
  split. intros f. unfold funcomp,idfun.
  apply hlevelnweqeqmaprinv. destruct A as [ A is ]. destruct B as [ B is' ]. unfold
funcomp, idfun. intros f. apply hlevelnweqeqmaplinv.
Defined.

Lemma iscontrandweq { A B : UU } ( is : iscontr A ) ( is' : iscontr B ) : iscontr ( weq A
B ).
Proof.
  rewrite ( weqeqmap univ ( pair _ ( tounitandiscontr is ) ) ).
   rewrite ( weqeqmap univ ( pair _ ( tounitandiscontr is' ) ) ).
   split with ( idweq _ ). intros f.
   assert ( paths ( idfun unit ) ( pr1 f ) ) as i.
   apply funextfun. intro x. apply isapropunit.
   apply ( @pathintotalfiber _ _ ( pair ( idfun unit ) _ ) f i ).
   apply isapropisweq.
Defined.

Theorem isofhlevelSnhn ( n : nat ) : isofhlevel ( S n ) ( total ( fun x : UU => isofhlevel
n x ) ).
Proof.
  change ( forall A B : total ( fun x : UU => isofhlevel n x ) , isofhlevel n ( paths A B ) ).
intros A B. rewrite ( weqeqmap univ ( pair _ ( isweqhlevelneqweqmap n A B) ) ).
  destruct n. apply iscontrandweq. apply ( pr2 A ). apply ( pr2 B ).
  apply ( hlevelweqcodomain ). apply ( pr2 B ).
Defined.
```