```
(** *Fixing notation, terminology and basic lemmas *)

(** By Alvaro Pelayo, Vladimir Voevodsky and Michael A. Warren *)

(** Settings *)

Add Rec LoadPath "../Generalities".  Add Rec LoadPath "../hlevel1".
Add Rec LoadPath "../hlevel2".

Unset Automatic Introduction. (** This line has to be removed for
the
file to compile with Coq8.2 *)

(** Imports *)

Require Export hSet.

Require Export hnat.

Require Export hz.

(*Require Export finitesets.

Require Export stnfsets.*)

(** Fixing some notation *)

(** * Notation, terminology and very basic facts *)

Notation "x ~> y" := ( paths x y ) ( at level 50 ) : type_scope.

Implicit Arguments tpair [ T P ].

Lemma pathintotalfiber ( B : UU ) ( E : B -> UU ) ( b0 b1 : B )
( e0 :
E b0 ) ( e1 : E b1 ) ( p0 : b0 ~> b1 ) ( p1 : transportf E p0 e0 ~>
e1
) : ( tpair b0 e0 ) ~> ( tpair b1 e1 ).  Proof.  intros. destruct
p0,
p1. apply idpath.  Defined.

Definition neq ( X : UU0 ) : X -> X -> hProp := fun x y : X =>
hProppair (neg (x ~> y)) (isapropneg (x ~> y)).

Definition pathintotalpr1 { B : UU } { E : B -> UU } { v w : total2
E
} ( p : v ~> w ) : ( pr1 v ) ~> ( pr1 w ) := maponpaths ( fun x =>
pr1
x ) p.

Lemma isinclisinj { A B : UU } { f : A -> B } ( p : isincl f ) { a
b :
A } ( p : f a ~> f b ) : a ~> b.  Proof.  intros. set ( q := p ( f
a )
```

```
). set ( a' := hfiberpair f a ( idpath ( f a ) ) ).  set ( b' :=
hfiberpair f b ( pathsinv0 p0 ) ). assert ( a' ~> b' ) as p1. apply
(
p ( f a ) ).  apply ( pathintotalpr1 p1 ).  Defined.

(** * I. Lemmas on natural numbers *)

Lemma minus0r ( n : nat ) : minus n 0 ~> n.  Proof.  intros
n. destruct n. apply idpath. apply idpath.  Defined.

Lemma minusnn0 ( n : nat ) : minus n n ~> 0%nat.  Proof.
intro. induction n. apply idpath. assumption.  Defined.

Lemma minussn1 ( n : nat ) : minus ( S n ) 1 ~> n.  Proof.
intro. destruct n. apply idpath. apply idpath.  Defined.

Lemma minussn1non0 ( n : nat ) ( p : natlth 0 n ) : S ( minus n 1 )
~>
n.  Proof.  intro. destruct n. intro p. assert empty. exact (
isirreflnatlth 0%nat p ). contradiction. intro. apply
maponpaths. apply minus0r.  Defined.

Lemma minusleh ( n m : nat ) : natleh ( minus n m ) n.  Proof.
intros
n. induction n. intros m. apply isreflnatleh.  intros m. destruct
m. apply isreflnatleh. exact ( istransnatleh ( minus n m ) n ( S n )
(
IHn m ) ( natlthtoleh n ( S n ) ( natlthnsn n ) ) ).  Defined.

Lemma minus1leh { n m : nat } ( p : natlth 0 n ) ( q : natlth 0 m )
(
r : natleh n m ) : natleh ( minus n 1 ) ( minus m 1 ).  Proof.
intro
n. destruct n. auto. intros m p q r. destruct m. assert empty. exact
(
isirreflnatlth 0%nat q ). contradiction. assert ( natleh n m ) as
a. apply r. assert ( natleh ( minus n 0%nat ) m ) as a0. exact (
transportf ( fun x : nat => natleh x m ) ( pathsinv0 ( minus0r n ) )
a
). exact ( transportf ( fun x : nat => natleh ( minus n 0 ) x ) (
pathsinv0 ( minus0r m ) ) a0 ).  Defined.

Lemma minuslth ( n m : nat ) ( p : natlth 0 n ) ( q : natlth 0 m ) :
natlth ( minus n m ) n.  Proof.  intro n. destruct n. auto. intros m
p
q. destruct m. assert empty. exact ( isirreflnatlth 0%nat q
). contradiction. apply ( natlehlthtrans _ n _ ). apply ( minusleh n
m
). apply natlthnsn.  Defined.

Lemma natlthsntoleh ( n m : nat ) : natlth m ( S n ) -> natleh m n.
Proof.  intro. induction n. intros m p. destruct m. apply
isreflnatleh. assert ( natlth m 0 ) as q. apply p. intro. unfold
natlth in q. exact ( negnatgth0n m q ).  intros m p. destruct m.
```

```
apply
natleh0n. apply ( IHn m ). assumption.  Defined.

Lemma natlthminus0 { n m : nat } ( p : natlth m n ) : natlth 0
( minus
n m ).  Proof.  intro n. induction n. intros m p. assert empty.
exact
( negnatlthn0 m p ). contradiction. intros m p. destruct
m. auto. apply IHn. apply p.  Defined.

Lemma natlthsnminussmsn ( n m : nat ) ( p : natlth m n ) : natlth (
minus ( S n ) ( S m ) ) ( S n ).  Proof.  intro. induction n. intros
m
p. assert empty. apply
nopathsfalsetotrue. assumption. contradiction. intros m p. destruct
m. assert ( minus ( S ( S n ) ) 1 ~> S n ) as f. destruct
n. auto. auto. rewrite f. apply natlthnsn. apply ( istransnatlth _
( S
n ) _ ). apply IHn. assumption. apply natlthnsn.  Defined.

Lemma natlehsnminussmsn ( n m : nat ) ( p : natleh m n ) : natleh (
minus ( S n ) ( S m ) ) ( S n ).  Proof.  intro n. induction n.
intros
m p X. assert empty. apply nopathsfalsetotrue. assumption.
assumption. intros m p.  destruct m. apply natlthtoleh. apply
natlthnsn.  apply ( istransnatleh _ ( S n ) _ ). apply
IHn. assumption. apply natlthtoleh. apply natlthnsn.  Defined.

Lemma pathssminus ( n m : nat ) ( p : natlth m ( S n ) ) : S ( minus
n
m ) ~> minus ( S n ) m.  Proof.  intro n. induction n. intros m
p. destruct m. auto. assert empty. apply nopathstruetofalse. apply
pathsinv0. assumption. contradiction. intros m p. destruct
m. auto. apply IHn. apply p.  Defined.

Lemma natlehsminus ( n m : nat ) : natleh ( minus ( S n ) m ) ( S (
minus n m ) ).  Proof.  intro n. induction n. intros m X. apply
nopathstruetofalse. apply pathsinv0. destruct
m. assumption. assumption. intros m. destruct m. apply
isreflnatleh. apply IHn.  Defined.

Lemma natlthssminus { n m l : nat } ( p : natlth m ( S n ) ) ( q :
natlth l ( S ( minus ( S n ) m ) ) ) : natlth l ( S ( S n ) ).
Proof.
intro n. intros m l p q. apply ( natlthlehtrans _ ( S ( minus ( S
n )
m ) ) ).  assumption. destruct m. apply isreflnatleh. apply
natlthtoleh.  apply natlthsnminussmsn. assumption.  Defined.

Lemma natdoubleminus { n k l : nat } ( p : natleh k n ) ( q : natleh
l
k ) : ( minus n k) ~> ( minus ( minus n l ) ( minus k l ) ).  Proof.
intro n. induction n. auto. intros k l p q. destruct k.  destruct l.
auto. assert empty. exact ( negnatlehsn0 l q ). contradiction.
```

```
destruct l. auto. apply ( IHn k l ). assumption. assumption.
Defined.

Lemma minusnleh1 ( n m : nat ) ( p : natlth m n ) : natleh m ( minus
n
1 ).  Proof.  intro n. destruct n.  intros m p. assert empty. exact
(
negnatlthn0 m p ). contradiction.  intros m p. destruct m. apply
natleh0n. apply natlthsntoleh. change ( minus ( S n ) 1 ) with
( minus
n 0 ). rewrite minus0r. assumption.  Defined.

Lemma doubleminuslehpaths ( n m : nat ) ( p : natleh m n ) : minus n
(
minus n m ) ~> m.  Proof.  intro n. induction n. intros m p.
destruct
( natlehchoice m 0 p ) as [ h | k ]. assert empty. apply negnatlthn0
with ( n := m ). assumption. contradiction.  simpl. apply
pathsinv0. assumption.

  intros. destruct m. simpl. apply minusnn0. change ( minus ( S n )
(
minus n m ) ~> S m ). rewrite <- pathssminus. rewrite IHn. apply
idpath. assumption. apply ( minuslth ( S n ) ( S m ) ). apply (
natlehlthtrans _ n ). apply natleh0n. apply natlthnsn. apply (
natlehlthtrans _ m ). apply natleh0n. apply natlthnsn.  Defined.

Lemma boolnegtrueimplfalse ( v : bool ) ( p : neg ( v ~> true ) ) :
v
~> false.  Proof.  intros. destruct v. assert empty. apply
p. auto. contradiction. auto.  Defined.

Definition natcoface ( i : nat ) : nat -> nat.  Proof.  intros i
n. destruct ( natgtb i n ). exact n. exact ( S n ).  Defined.

Lemma natcofaceleh ( i n upper : nat ) ( p : natleh n upper ) :
natleh
( natcoface i n ) ( S upper ).  Proof.  intros. unfold
natcoface. destruct ( natgtb i n ). apply natlthtoleh.  apply (
natlehlthtrans _ upper ). assumption. apply natlthnsn. apply p.
Defined.

Lemma natgehimplnatgtbfalse ( m n : nat ) ( p : natgeh n m ) :
natgtb
m n ~> false.  Proof.  intros. unfold natgeh in p. unfold natgth in
p. apply boolnegtrueimplfalse. intro q.  apply p. auto.  Defined.

Definition natcofaceretract ( i : nat ) : nat -> nat.  Proof.
intros
i n. destruct ( natgtb i n ). exact n. exact ( minus n 1 ).
Defined.

Lemma natcofaceretractisretract ( i : nat ) : funcomp ( natcoface
i )
```

```
( natcofaceretract i ) ~> idfun nat.  Proof.  intro i. apply
funextfun. intro n. unfold funcomp.  set ( c := natlthorgeh n i
). destruct c as [ h | k ]. unfold natcoface. rewrite h. unfold
natcofaceretract. rewrite h.  apply idpath.  assert ( natgtb i n ~>
false ) as f. apply natgehimplnatgtbfalse. assumption.  unfold
natcoface. rewrite f. unfold natcofaceretract.  assert ( natgtb i
( S
n ) ~> false ) as ff.  apply natgehimplnatgtbfalse. apply (
istransnatgeh _ n ).  apply natgthtogeh. apply natgthsnn.
assumption.
rewrite ff.  rewrite minussn1.  apply idpath.  Defined.

Lemma isinjnatcoface ( i x y : nat ) : natcoface i x ~> natcoface i
y
-> x ~> y.  Proof.  intros i x y p. change x with ( ( idfun _ ) x
). rewrite <- ( natcofaceretractisretract i ). change y with
( ( idfun
_ ) y ). rewrite <- ( natcofaceretractisretract i ). unfold funcomp.
rewrite p. apply idpath.  Defined.

Lemma natlehdecomp ( b a : nat ) : hexists ( fun c : nat => ( a + c
)%nat ~> b ) -> natleh a b.  Proof.  intro b. induction b. intros a
p. apply p. intro t. destruct t as [ c f ]. destruct a. apply
isreflnatleh.  assert empty. simpl in f . exact ( negpathssx0 ( a +
c
) f ). contradiction.  intros a p. apply p. intro t. destruct t as
[ c
f ]. destruct a.  apply natleh0n. assert ( natleh a b ) as q. simpl
in
f.  apply IHb.  intro P. intro s. apply s. split with c. apply
invmaponpathsS. assumption. apply q.  Defined.

Lemma natdivleh ( a b k : nat ) ( f : ( a ∗ k )%nat ~> b ) : coprod
(
natleh a b ) ( b ~> 0%nat ).  Proof.  intros. destruct k. rewrite
natmultcomm in f. simpl in f. apply ii2. apply
pathsinv0. assumption. rewrite natmultcomm in f. simpl in f. apply
ii1. apply natlehdecomp. intro P. intro g. apply g. split with ( k ∗
a
)%nat. assumption.  Defined.

(** ∗ II. Lemmas on rings ∗)

Open Scope rng_scope.

Lemma rngminusdistr { X : commrng } ( a b c : X ) : a ∗ (b − c) ~>
(a
∗ b − a ∗ c).  Proof.  intros. rewrite rngldistr. rewrite
rngrmultminus. apply idpath.  Defined.

Lemma rngminusdistl { X : commrng } ( a b c : X ) : (b − c) ∗ a ~>
(b
∗ a − c ∗ a).  Proof.  intros. rewrite rngrdistr. rewrite
rnglmultminus. apply idpath.  Defined.
```

```
Lemma multinvmultstable ( A : commrng ) ( a b : A ) ( p :
multinvpair
A a ) ( q : multinvpair A b ) : multinvpair A ( a ∗ b ).  Proof.
intros. destruct p as [ a' p ]. destruct q as [ b' q ]. split with (
b' ∗ a' ).  split. change ( ( ( b' ∗ a' ) ∗ ( a ∗ b ) )%rng ~> (
@rngunel2 A )). rewrite ( rngassoc2 A b').  rewrite <- ( rngassoc2 A
a' ). change ( dirprod ( ( a' ∗ a )%rng ~> ( @rngunel2 A ) ) ( ( a ∗
a' )%rng ~> ( @rngunel2 A ) ) ) in p. change ( dirprod ( ( b' ∗ b
)%rng ~> ( @rngunel2 A ) ) ( ( b ∗ b' )%rng ~> ( @rngunel2 A ) ) )
in
q. rewrite <- ( pr1 q ).  apply maponpaths. assert ( a' ∗ a ∗ b ~> 1
∗
b ) as f. apply ( maponpaths ( fun x => x ∗ b ) ( pr1 p ) ). rewrite
rnglunax2 in f. assumption. change ( ( ( a ∗ b ) ∗ ( b' ∗ a' ) )%rng
~> ( @rngunel2 A )). rewrite ( rngassoc2 A a). rewrite <-
( rngassoc2
A b ). change ( dirprod ( ( a' ∗ a )%rng ~> ( @rngunel2 A ) ) ( ( a
∗
a' )%rng ~> ( @rngunel2 A ) ) ) in p. change ( dirprod ( ( b' ∗ b
)%rng ~> ( @rngunel2 A ) ) ( ( b ∗ b' )%rng ~> ( @rngunel2 A ) ) )
in
q. rewrite <- ( pr2 q ). rewrite ( pr2 q ). rewrite rnglunax2. apply
p.  Defined.

Lemma commrngaddinvunique ( X : commrng ) ( a b c : X ) ( p : @op1 X
a
b ~> @rngunel1 X ) ( q : @op1 X a c ~> @rngunel1 X ) : b ~> c.
Proof.
intros. rewrite ( pathsinv0 ( rngrunax1 X b ) ).  rewrite
( pathsinv0
q ). rewrite ( pathsinv0 ( rngassoc1 X _ _ _ ) ).  rewrite
( rngcomm1
X b _ ). rewrite p.  rewrite rnglunax1. apply idpath.  Defined.

Lemma isapropmultinvpair ( X : commrng ) ( a : X ) : isaprop (
multinvpair X a ).  Proof.  intros. unfold isaprop.  intros b c.

  assert ( b ~> c ) as f. destruct b as [ b b' ]. destruct c as [ c
c'
  ].  assert ( b ~> c ) as f0. rewrite <- ( @rngrunax2 X b ). change
(
  b ∗ ( @rngunel2 X ) ) with ( b ∗ 1 )%multmonoid. rewrite <- ( pr2
c'
  ). change ( ( b ∗ ( a ∗ c ) )%rng ~> c ).  rewrite <- ( rngassoc2
X
  ). change ( b ∗ a )%rng with ( b ∗ a )%multmonoid. rewrite ( pr1
b'
  ). change ( ( @rngunel2 X ) ∗ c ~> c )%rng. apply rnglunax2.
apply
  pathintotalfiber with ( p0 := f0 ).  assert ( isaprop ( dirprod (c
∗
  a ~> ( @rngunel2 X ) ) (a ∗ c ~> ( @rngunel2 X )) ) ) as is.
apply
```

```
  isofhleveldirprod. apply ( setproperty X ). apply ( setproperty X
  ). apply is.  split with f. intros g. assert ( isaset
( multinvpair
  X a ) ) as is. unfold multinvpair. unfold invpair. change isaset
  with ( isofhlevel 2 ). apply isofhleveltotal2. apply ( pr1 ( pr1 (
  rigmultmonoid X ) ) ). intros. apply isofhleveldirprod. apply
  hlevelntosn. apply ( setproperty X ). apply hlevelntosn. apply (
  setproperty X ). apply is.  Defined.

Close Scope rng_scope.

(** * III. Lemmas on hz *)

Open Scope hz_scope.

Lemma hzaddinvplus ( n m : hz ) : - ( n + m ) ~> ( ( - n ) + ( - m )
).  Proof.  intros. apply commrngaddinvunique with ( a := n + m ).
apply rngrinvax1.  assert ( ( n + m ) + ( - n + - m ) ~> ( n + - n +
m
+ - m ) ) as i.  assert ( n + m + ( - n + - m ) ~> ( n + ( m + ( - n
+
- m ) ) ) ) as i0.  apply rngassoc1. assert ( n + ( m + ( - n + -
m )
) ~> ( n + ( m + - n + - m ) ) ) as i1.  apply maponpaths. apply
pathsinv0. apply rngassoc1.  assert ( n + ( m + - n + - m ) ~> ( n +
(
- n + m + - m ) ) ) as i2.  apply maponpaths. apply ( maponpaths
( fun
x : _ => x + - m ) ).  apply rngcomm1. assert ( n + ( - n + m + -
m )
~> ( n + ( - n + m ) + - m ) ) as i3.  apply pathsinv0. apply
rngassoc1.  assert ( n + ( - n + m ) + - m ~> ( n + - n + m + -
m ) )
as i4.  apply pathsinv0. apply ( maponpaths ( fun x : _ => x + - m )
). apply rngassoc1.  exact ( pathscomp0 i0 ( pathscomp0 i1 (
pathscomp0 i2 ( pathscomp0 i3 i4 ) ) ) ).  assert ( n + - n + m + -m
~> 0 ) as j.  assert ( n + - n + m + - m ~> ( 0 + m + - m ) ) as j0.
apply ( maponpaths ( fun x : _ => x + m + - m ) ). apply rngrinvax1.
assert ( 0 + m + - m ~> ( m + - m ) ) as j1. apply ( maponpaths
( fun
x : _ => x + - m ) ). apply rnglunax1.  assert ( m + - m ~> 0 ) as
j2. apply rngrinvax1.  exact ( pathscomp0 j0 ( pathscomp0 j1 j2 ) ).
exact ( pathscomp0 i j ).  Defined.

Lemma hzgthsntogeh ( n m : hz ) ( p : hzgth ( n + 1 ) m ) : hzgeh n
m.
Proof.  intros. set ( c := hzgthchoice2 ( n + 1 ) m ).  destruct c
as
[ h | k ].  exact p. assert ( hzgth n m ) as a. exact (
hzgthandplusrinv n m 1 h ).  apply hzgthtogeh. exact a. rewrite (
hzplusrcan n m 1 k ).  apply isreflhzgeh.  Defined.

Lemma hzlthsntoleh ( n m : hz ) ( p : hzlth m ( n + 1 ) ) : hzleh m
n.
```

```
Proof.  intros. unfold hzlth in p. assert ( hzgeh n m ) as a. apply
hzgthsntogeh. exact p. exact a.  Defined.

Lemma hzabsvalchoice ( n : hz ) : coprod ( 0%nat ~> ( hzabsval n ) )
(
total2 ( fun x : nat => S x ~> ( hzabsval n ) ) ).  Proof.
intros. destruct ( natlehchoice _ _ ( natleh0n ( hzabsval n ) ) ) as
[
l | r ].  apply ii2. split with ( minus ( hzabsval n ) 1 ). rewrite
pathssminus.  change ( minus ( hzabsval n ) 0 ~> hzabsval n ).
rewrite
minus0r. apply idpath. assumption. apply ii1. assumption.  Defined.

Lemma hzlthminusswap ( n m : hz ) ( p : hzlth n m ) : hzlth ( - m )
(
- n ).  Proof.  intros. rewrite <- ( hzplusl0 ( - m ) ). rewrite <-
(
hzrminus n ).  change ( hzlth ( n + - n + - m ) ( - n ) ). rewrite
hzplusassoc. rewrite ( hzpluscomm ( -n ) ). rewrite <-
hzplusassoc. assert ( - n ~> ( 0 + - n ) ) as f. apply
pathsinv0. apply hzplusl0. assert ( hzlth ( n + - m + - n ) ( 0 + -
n
) ) as q. apply hzlthandplusr.  rewrite <- ( hzrminus m ). change
( m
- m ) with ( m + - m ). apply hzlthandplusr. assumption. rewrite <-
f
in q. assumption.  Defined.


Lemma hzlthminusequiv ( n m : hz ) : dirprod ( ( hzlth n m ) -> (
hzlth 0 ( m - n ) ) ) ( ( hzlth 0 ( m - n ) ) -> ( hzlth n m ) ).
Proof.  intros. rewrite <- ( hzrminus n ). change ( n - n ) with ( n
+
- n ). change ( m - n ) with ( m + - n ). split. intro p. apply
hzlthandplusr.  assumption. intro p. rewrite <- ( hzplusr0 n
). rewrite <- ( hzplusr0 m ).  rewrite <- ( hzlminus n ). rewrite <-
2! hzplusassoc. apply hzlthandplusr. assumption.  Defined.

Lemma hzlthminus ( n m k : hz ) ( p : hzlth n k ) ( q : hzlth m k )
(
q' : hzleh 0 m ) : hzlth ( n - m ) k.  Proof.  intros. destruct (
hzlehchoice 0 m q' ) as [ l | r ].  apply ( istranshzlth _ n _ ).
assert ( hzlth ( n - m ) ( n + 0 ) ) as i0.  rewrite <- ( hzrminus m
). change ( m - m ) with ( m + - m ).  rewrite <- ( hzplusassoc
). apply hzlthandplusr.  assert ( hzlth ( n + 0 ) ( n + m ) ) as
i00. apply hzlthandplusl. assumption. rewrite ( hzplusr0 ) in
i00. assumption. rewrite hzplusr0 in
i0. assumption. assumption. rewrite <- r.  change ( n - 0 ) with ( n
+
- 0 ). rewrite hzminuszero. rewrite ( hzplusr0 n ). assumption.
Defined.

Lemma hzabsvalandminuspos ( n m : hz ) ( p : hzleh 0 n ) ( q : hzleh
0
```

```
m ) : nattohz ( hzabsval ( n - m ) ) ~> nattohz ( hzabsval ( m - n )
).  Proof.  intros. destruct ( hzlthorgeh n m ) as [ l | r ].
assert
( hzlth ( n - m ) 0 ) as a. change ( n - m ) with ( n + - m ).
rewrite <- ( hzrminus m ). change ( m - m ) with ( m + - m ).  apply
(
hzlthandplusr ). assumption.  assert ( hzlth 0 ( m - n ) ) as b.
change ( m - n ) with ( m + - n ). rewrite <- ( hzrminus n ). change
(
n - n ) with ( n + - n ).  apply hzlthandplusr. assumption.  rewrite
(
hzabsvallth0 a ). rewrite hzabsvalgth0.  change ( n - m ) with ( n +
-
m ). rewrite hzaddinvplus.  change ( - - m ) with ( - - m
)%rng. rewrite ( rngminusminus ).  rewrite hzpluscomm. apply
idpath. apply b.  destruct ( hzgehchoice n m r ) as [ h | k ].
assert
( hzlth 0 ( n - m ) ) as a. change ( n - m ) with ( n + - m ).
rewrite <- ( hzrminus m ). change ( m - m ) with ( m + - m ).  apply
hzlthandplusr. assumption. assert ( hzlth ( m - n ) 0 ) as b.
change
( m - n ) with ( m + - n ). rewrite <- ( hzrminus n ).  apply
hzlthandplusr. apply h.  rewrite ( hzabsvallth0 b ). rewrite (
hzabsvalgth0 ).  change ( ( n + - m ) ~> - ( m + - n ) ).  rewrite
hzaddinvplus. change ( - - n ) with ( - - n )%rng.  rewrite
rngminusminus. rewrite hzpluscomm. apply idpath. apply a.  rewrite
k. apply idpath.  Defined.

Lemma hzabsvalneq0 ( n : hz ) ( p : hzneq 0 n ) : hzlth 0 ( nattohz
(
hzabsval n ) ).  Proof.  intros. destruct ( hzneqchoice 0 n p ) as [
left | right ].  rewrite hzabsvallth0. apply hzlth0andminus. apply
left. apply left.  rewrite hzabsvalgth0. assumption. apply right.
Defined.

Definition hzrdistr ( a b c : hz ) : ( a + b ) * c ~> ( ( a * c ) +
(
b * c ) ) := rngrdistr hz a b c.

Definition hzldistr ( a b c : hz ) : c * ( a + b ) ~> ( ( c * a ) +
(
c * b ) ) := rngldistr hz a b c.


Lemma hzabsvaland1 : hzabsval 1 ~> 1%nat.  Proof.  apply
( isinclisinj
isinclnattohz ). rewrite hzabsvalgth0. rewrite nattohzand1. apply
idpath. rewrite <- ( hzplusl0 1). apply ( hzlthnsn 0 ).  Defined.

Lemma hzabsvalandplusnonneg ( n m : hz ) ( p : hzleh 0 n ) ( q :
hzleh
0 m ) : hzabsval ( n + m ) ~> ( ( hzabsval n ) + ( hzabsval m ) )
%nat.
Proof.  intros. assert ( hzleh 0 ( n + m ) ) as r. rewrite <- (
```

```
hzrminus n ). change ( n - n ) with ( n + - n ). apply
hzlehandplusl. apply ( istranshzleh _ 0 _ ).  apply
hzgeh0andminus. apply p. assumption.  apply ( isinclisinj
isinclnattohz ).  rewrite nattohzandplus. rewrite 3!
hzabsvalgeh0. apply idpath. apply q. apply p. apply r.  Defined.

Lemma hzabsvalandplusneg ( n m : hz ) ( p : hzlth n 0 ) ( q : hzlth
m
0 ) : hzabsval ( n + m ) ~> ( ( hzabsval n ) + ( hzabsval m ) )%nat.
Proof.  intros. assert ( hzlth ( n + m ) 0 ) as r. rewrite <- (
hzrminus n ). change ( n - n ) with ( n + - n ). apply
hzlthandplusl. apply ( istranshzlth _ 0 _ ).  assumption. apply
hzlth0andminus. assumption.  apply ( isinclisinj isinclnattohz ).
rewrite nattohzandplus. rewrite 3! hzabsvallth0. rewrite
hzaddinvplus. apply idpath. apply q. apply p. apply r.  Defined.

Lemma hzabsvalandnattohz ( n : nat ) : hzabsval ( nattohz n ) ~> n.
Proof.  induction n. rewrite nattohzand0. rewrite hzabsval0. apply
idpath.  rewrite nattohzandS. rewrite hzabsvalandplusnonneg.
rewrite
hzabsvaland1. simpl. apply maponpaths. assumption.  rewrite <- (
hzplusl0 1). apply hzlthtoleh. apply ( hzlthnsn 0 ).  rewrite <-
nattohzand0. apply nattohzandleh. apply natleh0n.  Defined.

Lemma hzabsvalandlth ( n m : hz ) ( p : hzleh 0 n ) ( p' : hzlth n
m )
: natlth ( hzabsval n ) ( hzabsval m ).  Proof.  intros. destruct (
natlthorgeh ( hzabsval n ) ( hzabsval m ) ) as [ h | k ].
assumption. assert empty.  apply ( isirreflhzlth m ). apply (
hzlehlthtrans _ n _ ).  rewrite <- ( hzabsvalgeh0 ). rewrite <- (
hzabsvalgeh0 p ).  apply nattohzandleh. apply k. apply
hzgthtogeh. apply ( hzgthgehtrans _ n _ ).  apply p'. apply
p. assumption. contradiction.  Defined.

Lemma nattohzandlthinv ( n m : nat ) ( p : hzlth ( nattohz n )
(nattohz m ) ) : natlth n m.  Proof.  intros. rewrite <- (
hzabsvalandnattohz n ). rewrite <- ( hzabsvalandnattohz m ).  apply
hzabsvalandlth. change 0 with ( nattohz 0%nat ).  apply
nattohzandleh.
apply natleh0n . assumption.  Defined.

Close Scope hz_scope.

(** * IV. Generalities on apartness relations *)

Definition iscomparel { X : UU0 } ( R : hrel X ) := forall x y z :
X,
R x y -> coprod ( R x z ) ( R z y ).

Definition isapart { X : UU0 } ( R : hrel X ) := dirprod ( isirrefl
R
) ( dirprod ( issymm R ) ( iscotrans R ) ).

Definition istightapart { X : UU0 } ( R : hrel X ) := dirprod (
```

```
isapart R ) ( forall x y : X, neg ( R x y ) -> ( x ~> y ) ).

Definition apart ( X : hSet ) := total2 ( fun R : hrel X => isapart
R
).

Definition isbinopapartl { X : hSet } ( R : apart X ) ( opp : binop
X
) := forall a b c : X, ( ( pr1 R ) ( opp a b ) ( opp a c ) ) ->
( pr1
R ) b c.

Definition isbinopapartr { X : hSet } ( R : apart X ) ( opp : binop
X
) := forall a b c : X, ( pr1 R ) ( opp b a ) ( opp c a ) -> ( pr1
R )
b c.

Definition isbinopapart { X : hSet } ( R : apart X ) ( opp : binop
X )
:= dirprod ( isbinopapartl R opp ) ( isbinopapartr R opp ).

Lemma deceqtoneqapart { X : UU0 } ( is : isdeceq X ) : isapart ( neq
X
).  Proof.  intros. split. intros a. intro p. apply p. apply idpath.
split. intros a b p q. apply p. apply pathsinv0. assumption.  intros
a
c b p P s. apply s. destruct ( is a c ) as [ l | r ].  apply
ii2. rewrite <- l. assumption. apply ii1. assumption.  Defined.

Definition isapartdec { X : hSet } ( R : apart X ) := forall a b :
X,
coprod ( ( pr1 R ) a b ) ( a ~> b ).

Lemma isapartdectodeceq { X : hSet } ( R : apart X ) ( is :
isapartdec
R ) : isdeceq X.  Proof.  intros X R is y z. destruct ( is y z ) as
[
l | r ]. apply ii2. intros f. apply ( ( pr1 ( pr2 R ) ) z). rewrite
f
in l. assumption. apply ii1. assumption.  Defined.

Lemma isdeceqtoisapartdec ( X : hSet ) ( is : isdeceq X ) :
isapartdec
( tpair _ ( deceqtoneqapart is ) ).  Proof.  intros X is a b.
destruct
( is a b ) as [ l | r ]. apply ii2. assumption. apply ii1. intros
f. apply r. assumption.  Defined.

(** * V. Apartness relations on rings *)

Open Scope rng_scope.

Definition acommrng := total2 ( fun X : commrng => total2 ( fun R :
```

```
apart X => dirprod ( isbinopapart R ( @op1 X ) ) ( isbinopapart R (
@op2 X ) ) ) ).

Definition acommrngpair := tpair ( P := fun X : commrng => total2 (
fun R : apart X => dirprod ( isbinopapart R ( @op1 X ) ) (
isbinopapart R ( @op2 X ) ) ) ).  Definition acommrngconstr :=
acommrngpair.

Definition acommrngtocommrng : acommrng -> commrng := @pr1 _ _.
Coercion acommrngtocommrng : acommrng >-> commrng.

Definition acommrngapartrel ( X : acommrng ) := pr1 ( pr1 ( pr2
X ) ).
Notation " a # b " := ( acommrngapartrel _ a b ) ( at level 50 ) :
rng_scope.

Definition acommrng_aadd ( X : acommrng ) : isbinopapart ( pr1 ( pr2
X
) ) op1 := ( pr1 ( pr2 ( pr2 X ) ) ).  Definition acommrng_amult
( X :
acommrng ) : isbinopapart ( pr1 ( pr2 X ) ) op2 := ( pr2 ( pr2 ( pr2
X
) ) ).  Definition acommrng_airrefl ( X : acommrng ) : isirrefl
( pr1
( pr1 ( pr2 X ) ) ) := pr1 ( pr2 ( pr1 ( pr2 X ) ) ).  Definition
acommrng_asymm ( X : acommrng ) : issymm ( pr1 ( pr1 ( pr2
X ) ) ) :=
pr1 ( pr2 ( pr2 ( pr1 ( pr2 X ) ) ) ).  Definition acommrng_acotrans
(
X : acommrng ) : iscotrans ( pr1 ( pr1 ( pr2 X ) ) ) := pr2 ( pr2 (
pr2 ( pr1 ( pr2 X ) ) ) ).

Definition aintdom := total2 ( fun A : acommrng => dirprod ( (
rngunel2 ( X := A ) ) # 0 ) ( forall a b : A, ( a # 0 ) -> ( b # 0 )
-> ( ( a * b ) # 0 ) ) ).

Definition aintdompair := tpair ( P := fun A : acommrng => dirprod
( (
rngunel2 ( X := A ) ) # 0 ) ( forall a b : A, ( a # 0 ) -> ( b # 0 )
-> ( ( a * b ) # 0 ) ) ).  Definition aintdomconstr := aintdompair.

Definition pr1aintdom : aintdom -> acommrng := @pr1 _ _.  Coercion
pr1aintdom : aintdom >-> acommrng.

Definition aintdomazerosubmonoid ( A : aintdom ) : @subabmonoids (
rngmultabmonoid A ).  Proof. intros. split with ( fun x : A => ( x #
0
) ). split.  intros a b. simpl in *. apply A. apply a. apply b.
apply
A.  Defined.

Definition isaafield ( A : acommrng ) := dirprod ( ( rngunel2 ( X :=
A
) ) # 0 ) ( forall x : A, x # 0 -> multinvpair A x ).
```

```
Definition afld := total2 ( fun A : acommrng => isaafield A ).
Definition afldpair ( A : acommrng ) ( is : isaafield A ) : afld :=
tpair A is .  Definition pr1afld : afld -> acommrng := @pr1 _ _ .
Coercion pr1afld : afld >-> acommrng.

Lemma afldinvertibletoazero ( A : afld ) ( a : A ) ( p : multinvpair
A
a ) : a # 0.  Proof.  intros. destruct p as [ a' p ].  assert ( a' ∗
a
# 0 ) as q.  change ( a' ∗ a # 0 ). assert ( a' ∗ a ~> a ∗ a' ) as
f. apply ( rngcomm2 A ).  assert ( a ∗ a' ~> 1 ) as g. apply
p. rewrite f, g. apply A.  assert ( a' ∗ a # a' ∗ ( rngunel1 ( X :=
A
) ) ) as q'.  assert ( ( rngunel1 ( X := A ) ) ~> ( a' ∗ ( rngunel1
(
X := A ) ) ) ) as f.  apply pathsinv0. apply ( rngmultx0 A ).
rewrite
<- f. assumption.  apply ( ( pr1 ( acommrng_amult A ) ) a'
). assumption.  Defined.

Definition afldtoaintdom ( A : afld ) : aintdom .  Proof.  intro
. split with ( pr1 A ) . split. apply A. intros a b p q.  apply
afldinvertibletoazero. apply multinvmultstable. apply A. assumption.
apply A. assumption.  Defined.

Lemma timesazero { A : acommrng } { a b : A } ( p : a ∗ b # 0 ) :
dirprod ( a # 0 ) ( b # 0 ).  Proof.  intros. split. assert ( a ∗ b
#
0 ∗ b ) as h. rewrite ( rngmult0x A ). assumption. apply ( ( pr2 (
acommrng_amult A ) ) b ). assumption.  apply ( ( pr1
( acommrng_amult
A ) ) a ). rewrite ( rngmultx0 A ). assumption.  Defined.

Lemma aaminuszero { A : acommrng } { a b : A } ( p : a # b ) : ( a -
b
) # 0.  Proof.  intros. rewrite <- ( rngrunax1 A a ) in p. rewrite
<-
( rngrunax1 A b ) in p. assert ( a + 0 ~> ( a + ( b - b ) ) ) as
f. rewrite <- ( rngrinvax1 A b ). apply idpath. rewrite f in
p. rewrite <- ( rngmultwithminus1 A ) in p. rewrite <- ( rngassoc1
A)
in p. rewrite ( rngcomm1 A a ) in p.  rewrite ( rngassoc1 A b ) in
p. rewrite ( rngmultwithminus1 A ) in p.  apply ( ( pr1 (
acommrng_aadd A ) ) b ( a - b ) 0 ). assumption.  Defined.

Lemma aminuszeroa { A : acommrng } { a b : A } ( p : ( a - b ) #
0 ) :
a # b.  Proof.  intros.  change 0 with ( @rngunel1 A ) in p. rewrite
<- ( rngrinvax1 A b ) in p. rewrite <- ( rngmultwithminus1 A ) in
p. apply ( ( pr2 ( acommrng_aadd A ) ) ( -1 ∗ b ) a b ). assumption.
Defined.

Close Scope rng_scope.
```

```
(** * VI. Lemmas on logic *)

Lemma horelim ( A B : UU0 ) ( P : hProp ) : dirprod ( ishinh_UU A -> P
) ( ishinh_UU B -> P ) -> ( hdisj A B -> P ).  Proof.  intros A B P
p. intro q. apply q. intro u. destruct u as [ u | v ].  apply ( pr1 p
). intro Q. auto. apply ( pr2 p ). intro Q. auto.  Defined.

Lemma stronginduction { E : nat -> UU } ( p : E 0%nat ) ( q : forall n
: nat, natneq n 0%nat -> ( ( forall m : nat, natlth m n -> E m ) -> E
n ) ) : forall n : nat, E n.  Proof.  intros. destruct
n. assumption. apply q.  apply ( negpathssx0 n ).  induction n. intros
m t.  rewrite ( natlth1tois0 m t ). assumption. intros m t. destruct
( natlehchoice _ _ ( natlthsntoleh _ _ t ) ) as [ left | right ].
apply IHn. assumption. apply q. rewrite right. intro f.  apply (
negpathssx0 n ). assumption. intros k s. rewrite right in s. apply (
IHn k ). assumption.  Defined.

Lemma setquotprpathsandR { X : UU0 } ( R : eqrel X ) : forall x y : X,
setquotpr R x ~> setquotpr R y -> R x y.  Proof.  intros. assert ( pr1
( setquotpr R x ) y ) as i. assert ( pr1 ( setquotpr R y ) y ) as
i0. unfold setquotpr. apply R. destruct X0. assumption. apply i.
Defined.

(* Some lemmas on decidable properties of natural numbers. *)

Definition isdecnatprop ( P : nat -> hProp ) := forall m : nat, coprod
( P m ) ( neg ( P m ) ).

Lemma negisdecnatprop ( P : nat -> hProp ) ( is : isdecnatprop P ) :
isdecnatprop ( fun n : nat => hneg ( P n ) ).  Proof.  intros P is
n. destruct ( is n ) as [ l | r ]. apply ii2. intro j. assert hfalse
as x. apply j. assumption. apply x. apply ii1. assumption.  Defined.

Lemma bndexistsisdecnatprop ( P : nat -> hProp ) ( is : isdecnatprop P
) : isdecnatprop ( fun n : nat => hexists ( fun m : nat => dirprod (
natleh m n ) ( P m ) ) ).  Proof.  intros P is n. induction
n. destruct ( is 0%nat ) as [ l | r ].  apply ii1. apply
total2tohexists. split with 0%nat. split. apply
isreflnatleh. assumption. apply ii2. intro j. assert hfalse as
x. apply j. intro m. destruct m as [ m m' ]. apply r. rewrite (
natleh0tois0 m ( pr1 m' ) ) in m'. apply m'. apply x.

  destruct ( is ( S n ) ) as [ l | r ]. apply ii1. apply
```

```
total2tohexists. split with ( S n ). split. apply ( isreflnatleh ( S
n
) ). assumption. destruct IHn as [ l' | r' ]. apply ii1. apply l'.
intro m. destruct m as [ m m' ]. apply total2tohexists. split with
m. split. apply ( istransnatleh _ n _ ). apply m'. apply
natlthtoleh. apply natlthnsn. apply m'. apply ii2. intro j. apply
r'. apply j. intro m. destruct m as [ m m' ]. apply
total2tohexists. split with m. split. destruct ( natlehchoice m ( S
n
) ( pr1 m' ) ). apply natlthsntoleh. assumption. assert empty. apply
r. rewrite <- i. apply m'. contradiction. apply m'.  Defined.

Lemma isdecisbndqdec ( P : nat -> hProp ) ( is : isdecnatprop P )
( n
: nat ) : coprod ( forall m : nat, natleh m n -> P m ) ( hexists
( fun
m : nat => dirprod ( natleh m n ) ( neg ( P m ) ) ) ).  Proof.
intros
P is n. destruct ( bndexistsisdecnatprop _ ( negisdecnatprop P is )
n
) as [ l | r ]. apply ii2. assumption. apply ii1. intros m j.
destruct
( is m ) as [ l' | r' ].  assumption. assert hfalse as x. apply
r. apply total2tohexists. split with
m. split. assumption. assumption. contradiction.  Defined.

Lemma leastelementprinciple ( n : nat ) ( P : nat -> hProp ) ( is :
isdecnatprop P ) : P n -> hexists ( fun k : nat => dirprod ( P k ) (
forall m : nat, natlth m k -> neg ( P m ) ) ).  Proof.  intro
n. induction n. intros P is u. apply total2tohexists. split with
0%nat. split.  assumption. intros m i. assert empty. apply (
negnatgth0n m i ). contradiction. intros P is u.  destruct ( is
0%nat
) as [ l | r ]. apply total2tohexists. split with 0%nat. split.
assumption. intros m i. assert empty. apply ( negnatgth0n m i
). contradiction. set ( P' := fun m : nat => P ( S m ) ). assert (
forall m : nat, coprod ( P' m ) ( neg ( P' m ) ) ) as is'. intros
m. unfold P'. apply ( is ( S m ) ). set ( c := IHn P' is' u ). apply
c. intros k. destruct k as [ k v ]. destruct v as [ v0 v1 ]. apply
total2tohexists. split with ( S k ). split. assumption. intros
m. destruct m. intros i. assumption. intros i. apply v1. apply i.
Defined.

(** END OF FILE *)
```