```
(** *Formal Power Series *)

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

(** January 2011 *)

(** Settings *)

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

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

(** Imports *)

Require Export lemmas.

(** ** I. Summation in a commutative ring *)

Open Scope rng_scope.

Definition natsummation0 { R : commrng } ( upper : nat ) ( f : nat
->
R ) : R.  Proof.  intro R. intro upper. induction upper. intros.
exact
( f 0%nat ). intros. exact ( ( IHupper f + ( f ( S upper ) ) ) ).
Defined.

Lemma natsummationpaths { R : commrng } { upper upper' : nat } ( u :
upper ~> upper' ) ( f : nat -> R ) : natsummation0 upper f ~>
natsummation0 upper' f.  Proof.  intros. destruct u. auto.  Defined.

Lemma natsummationpathsupperfixed { R : commrng } { upper : nat }
( f
f' : nat -> R ) ( p : forall x : nat, natleh x upper -> f x ~> f'
x )
: natsummation0 upper f ~> natsummation0 upper f'.  Proof.  intros R
upper. induction upper. intros f f' p. simpl. apply p. apply
isreflnatleh.  intros. simpl. rewrite ( IHupper f f' ). rewrite ( p
(
S upper ) ). apply idpath. apply isreflnatleh. intros x p'. apply
p. apply ( istransnatleh _ upper ). assumption. apply
natlthtoleh. apply natlthnsn.  Defined.

(* Here we consider summation of functions which are, in a fixed
interval, 0 for all but either the first or last value. *)

Lemma natsummationae0bottom { R : commrng } { f : nat -> R }
( upper :
nat ) ( p : forall x : nat, natlth 0 x -> f x ~> 0 ) : natsummation0
upper f ~> ( f 0%nat ).  Proof.  intros R f upper. induction
```

```
upper. auto. intro p.  simpl. rewrite ( IHupper ). rewrite ( p ( S
upper ) ). rewrite ( rngrunax1 R ). apply idpath. apply (
natlehlthtrans _ upper _ ). apply natleh0n. apply
natlthnsn. assumption.  Defined.

Lemma natsummationae0top { R : commrng } { f : nat -> R } ( upper :
nat ) ( p : forall x : nat, natlth x upper -> f x ~> 0 ) :
natsummation0 upper f ~> ( f upper ).  Proof.  intros R f
upper. induction upper. auto. intro p.  assert ( natsummation0 upper
f
~> ( natsummation0 ( R := R ) upper ( fun x : nat => 0 ) ) ) as g.
apply natsummationpathsupperfixed. intros m q. apply p. exact (
natlehlthtrans m upper ( S upper ) q ( natlthnsn upper ) ).
simpl. rewrite g. assert ( natsummation0 ( R := R ) upper ( fun _ :
nat => 0 ) ~> 0 ) as g'. set ( g'' := fun x : nat => rngunel1 ( X :=
R
) ).  assert ( forall x : nat, natlth 0 x -> g'' x ~> 0 ) as q0.
intro
k. intro pp. auto. exact ( natsummationae0bottom upper q0 ). rewrite
g'. rewrite ( rnglunax1 R ). apply idpath.  Defined.

Lemma natsummationshift0 { R : commrng } ( upper : nat ) ( f : nat
->
R ) : natsummation0 ( S upper ) f ~> ( natsummation0 upper ( fun x :
nat => f ( S x ) ) + f 0%nat ).  Proof.  intros R upper. induction
upper. intros f. simpl. apply R.  intros. change ( natsummation0 ( S
upper ) f + f ( S ( S upper ) ) ~> ( natsummation0 upper ( fun x :
nat
=> f ( S x ) ) + f ( S ( S upper ) ) + f 0%nat ) ).  rewrite
IHupper. rewrite 2! ( rngassoc1 R ). rewrite ( rngcomm1 R ( f
0%nat )
_ ). apply idpath.  Defined.

Lemma natsummationshift { R : commrng } ( upper : nat ) ( f : nat ->
R
) { i : nat } ( p : natleh i upper ) : natsummation0 ( S upper ) f
~>
( natsummation0 upper ( funcomp ( natcoface i ) f ) + f i ).  Proof.
intros R upper. induction upper. intros f i p. destruct i.  unfold
funcomp. apply R. assert empty. exact ( negnatlehsn0 i p
). contradiction.  intros f i p. destruct i. apply
natsummationshift0.
destruct ( natlehchoice ( S i ) ( S upper ) p ) as [ h | k ].
change
( natsummation0 ( S upper ) f + f ( S ( S upper ) ) ~>
( natsummation0
( S upper ) ( funcomp ( natcoface ( S i ) ) f ) + f ( S i ) )
). rewrite ( IHupper f ( S i ) ).  simpl. unfold funcomp at 3.
unfold
natcoface at 3. rewrite 2! ( rngassoc1 R ).  rewrite ( rngcomm1 R _
(
f ( S i ) ) ). simpl. rewrite ( natgehimplnatgtbfalse i upper ).
apply
idpath. apply p. apply natlthsntoleh. assumption.  simpl. assert (
natsummation0 upper ( funcomp ( natcoface ( S i ) ) f ) ~>
natsummation0 upper f ) as h. apply
natsummationpathsupperfixed. intros m q. unfold funcomp.  unfold
natcoface. assert ( natlth m ( S i ) ) as q'. apply ( natlehlthtrans
_
upper ).  assumption. rewrite k. apply natlthnsn. unfold natlth in
q'.
rewrite q'. apply idpath.  rewrite <- h. unfold funcomp, natcoface
at
3. simpl. rewrite ( natgehimplnatgtbfalse i upper ).  rewrite 2! (
rngassoc1 R ). rewrite ( rngcomm1 R ( f ( S ( S upper ) ) ) ).
rewrite
k.  apply idpath. apply p.  Defined.

Lemma natsummationplusdistr { R : commrng } ( upper : nat ) ( f g :
nat -> R ) : natsummation0 upper ( fun x : nat => f x + g x ) ~> ( (
natsummation0 upper f ) + ( natsummation0 upper g ) ).  Proof.
intros
R upper. induction upper. auto. intros f g. simpl.  rewrite <- (
rngassoc1 R _ ( natsummation0 upper g ) _ ).  rewrite ( rngassoc1 R
(
natsummation0 upper f ) ).  rewrite ( rngcomm1 R _ ( natsummation0
upper g ) ). rewrite <- ( rngassoc1 R ( natsummation0 upper f ) ).
rewrite <- ( IHupper f g ). rewrite ( rngassoc1 R ).  apply idpath.
Defined.

Lemma natsummationtimesdistr { R : commrng } ( upper : nat ) ( f :
nat
-> R ) ( k : R ) : k * ( natsummation0 upper f ) ~> ( natsummation0
upper ( fun x : nat => k * f x ) ).  Proof.  intros R upper.
induction
upper. auto. intros f k. simpl.  rewrite <- ( IHupper ). rewrite <-
(
rngldistr R ).  apply idpath.  Defined.

Lemma natsummationtimesdistl { R : commrng } ( upper : nat ) ( f :
nat
-> R ) ( k : R ) : ( natsummation0 upper f ) * k ~> ( natsummation0
upper ( fun x : nat => f x * k ) ).  Proof.  intros R upper.
induction
upper. auto. intros f k. simpl.  rewrite <- IHupper. rewrite (
rngrdistr R ). apply idpath.  Defined.

Lemma natsummationsswapminus { R : commrng } { upper n : nat } ( f :
nat -> R ) ( q : natleh n upper ) : natsummation0 ( S ( minus upper
n
) ) f ~> natsummation0 ( minus ( S upper ) n ) f.  Proof.  intros R
upper. induction upper. intros n f q. destruct n. auto. assert
empty. exact ( negnatlehsn0 n q ). contradiction.  intros n f
q. destruct n. auto. change ( natsummation0 ( S ( minus upper n ) )
f
~> natsummation0 ( minus ( S upper ) n ) f ). apply IHupper. apply
q.
Defined.
```

```
(** The following lemma asserts that
$\sum^{n}_{k=0}\sum^{k}_{l=0}f(l,k-l)=\sum^{n}_{k=0}\sum^{n-k}_{l=0}
f(k,l)$
*)

Lemma natsummationswap { R : commrng } ( upper : nat ) ( f : nat ->
nat -> R ) : natsummation0 upper ( fun i : nat => natsummation0 i (
fun j : nat => f j ( minus i j ) ) ) ~> ( natsummation0 upper ( fun
k
: nat => natsummation0 ( minus upper k ) ( fun l : nat => f k
l ) ) ).
Proof.  intros R upper. induction upper. auto.

  intros f. change ( natsummation0 upper (fun i : nat =>
natsummation0
  i (fun j : nat => f j ( minus i j))) + natsummation0 ( S upper ) (
  fun j : nat => f j ( minus ( S upper ) j ) ) ~> ( natsummation0
  upper (fun k : nat => natsummation0 (S upper - k) (fun l : nat =>
f
  k l)) + natsummation0 ( minus ( S upper ) ( S upper ) ) ( fun l :
  nat => f ( S upper ) l ) ) ).  change ( natsummation0 upper (fun
i :
  nat => natsummation0 i (fun j : nat => f j ( minus i j))) + (
  natsummation0 upper ( fun j : nat => f j ( minus ( S upper ) j ) )
+
  f ( S upper ) ( minus ( S upper ) ( S upper ) ) ) ~>
( natsummation0
  upper (fun k : nat => natsummation0 (S upper - k) (fun l : nat =>
f
  k l)) + natsummation0 ( minus ( S upper ) ( S upper ) ) ( fun l :
  nat => f ( S upper ) l ) ) ).

  assert ( (natsummation0 upper (fun k : nat => natsummation0 ( S (
  minus upper k ) ) (fun l : nat => f k l)) ) ~> (natsummation0
upper
  (fun k : nat => natsummation0 (minus ( S upper ) k) (fun l : nat
=>
  f k l)) ) ) as A.  apply natsummationpathsupperfixed. intros n
  q. apply natsummationsswapminus. exact q. rewrite <- A.  change (
  fun k : nat => natsummation0 (S ( minus upper k)) (fun l : nat =>
f
  k l) ) with ( fun k : nat => natsummation0 ( minus upper k ) ( fun
l
  : nat => f k l ) + f k ( S ( minus upper k ) ) ).  rewrite (
  natsummationplusdistr upper _ ( fun k : nat => f k ( S ( minus
upper
  k ) ) ) ). rewrite IHupper. rewrite minusnn0. rewrite ( rngassoc1
  R).  assert ( natsummation0 upper ( fun j : nat => f j ( minus ( S
  upper ) j ) ) ~> natsummation0 upper ( fun k : nat => f k ( S (
  minus upper k ) ) ) ) as g. apply
  natsummationpathsupperfixed. intros m q. rewrite pathssminus.
apply
  idpath. apply ( natlehlthtrans _ upper ). assumption. apply
  natlthnsn.  rewrite g. apply idpath.  Defined.

(** * II. Reindexing along functions i : nat -> nat which are
automorphisms of the interval of summation.*)

Definition isnattruncauto ( upper : nat ) ( i : nat -> nat ) :=
dirprod ( forall x : nat, natleh x upper -> total2 ( fun y : nat =>
dirprod ( natleh y upper ) ( dirprod ( i y ~> x ) ( forall z : nat,
natleh z upper -> i z ~> x -> y ~> z ) ) ) ) ( forall x : nat,
natleh
x upper -> natleh ( i x ) upper ).

Lemma nattruncautoisinj { upper : nat } { i : nat -> nat } ( p :
isnattruncauto upper i ) { n m : nat } ( n' : natleh n upper )
( m' :
natleh m upper ) : i n ~> i m -> n ~> m.  Proof.  intros upper i p n
m
n' m' h. assert ( natleh ( i m ) upper ) as q. apply
p. assumption. set ( x := pr1 p ( i m ) q ). set ( v := pr1 x ). set
(
w := pr1 ( pr2 x ) ).  set ( y := pr1 ( pr2 ( pr2 x ) ) ). change (
pr1 x ) with v in w, y. assert ( v ~> n ) as a. apply ( pr2 x
). assumption. assumption. rewrite <- a. apply ( pr2 x
). assumption. apply idpath.  Defined.

Definition nattruncautopreimage { upper : nat } { i : nat -> nat }
( p
: isnattruncauto upper i ) { n : nat } ( n' : natleh n upper ) : nat
:= pr1 ( pr1 p n n' ).

Definition nattruncautopreimagepath { upper : nat } { i : nat ->
nat }
( p : isnattruncauto upper i ) { n : nat } ( n' : natleh n upper ) :
i
( nattruncautopreimage p n' ) ~> n := ( pr1 ( pr2 ( pr2 ( pr1 p n
n' )
) ) ).

Definition nattruncautopreimageineq { upper : nat } { i : nat ->
nat }
( p : isnattruncauto upper i ) { n : nat } ( n' : natleh n upper ) :
natleh ( nattruncautopreimage p n' ) upper := ( ( pr1 ( pr2 ( pr1 p
n
n' ) ) ) ).

Definition nattruncautopreimagecanon { upper : nat } { i : nat ->
nat
} ( p : isnattruncauto upper i ) { n : nat } ( n' : natleh n upper )
{
m : nat } ( m' : natleh m upper ) ( q : i m ~> n ) :
nattruncautopreimage p n' ~> m := ( pr2 ( pr2 ( pr2 ( pr1 p n
n' ) ) )
) m m' q.
```

```
Definition nattruncautoinv { upper : nat } { i : nat -> nat } ( p :
isnattruncauto upper i ) : nat -> nat.  Proof.  intros upper i p
n. destruct ( natgthorleh n upper ) as [ l | r ].  exact n. exact (
nattruncautopreimage p r ).  Defined.

Lemma nattruncautoinvisnattruncauto { upper : nat } { i : nat ->
nat }
( p : isnattruncauto upper i ) : isnattruncauto upper (
nattruncautoinv p ).  Proof.  intros. split. intros n n'. split with
(
i n ). split. apply p. assumption. split. unfold
nattruncautoinv. destruct ( natgthorleh ( i n ) upper ) as [ l | r
]. assert empty. apply ( isirreflnatlth ( i n ) ).  apply (
natlehlthtrans _ upper ). apply
p. assumption. assumption. contradiction. apply ( nattruncautoisinj
p
). apply ( nattruncautopreimageineq ). assumption. apply (
nattruncautopreimagepath p r ).  intros m x v.  unfold
nattruncautoinv
in v. destruct ( natgthorleh m upper ) as [ l | r ].  assert
empty. apply ( isirreflnatlth upper ).  apply ( natlthlehtrans _ m
). assumption. assumption. contradiction.  rewrite <- v. apply (
nattruncautopreimagepath p r ).  intros x X. unfold
nattruncautoinv. destruct ( natgthorleh x upper ) as [ l | r
]. assumption. apply ( nattruncautopreimageineq p r ).  Defined.

Definition truncnattruncauto { upper : nat } { i : nat -> nat }
( p :
isnattruncauto ( S upper ) i ) : nat -> nat.  Proof.  intros upper i
p
n.  destruct ( natlthorgeh ( i n ) ( S upper ) ) as [ l | r ].
exact
( i n ).  destruct ( natgehchoice _ _ r ) as [ a | b ]. exact ( i
n ).
destruct ( isdeceqnat n ( S upper ) ) as [ h | k ]. exact ( i n ).
exact ( i ( S upper ) ).  Defined.

Lemma truncnattruncautobound { upper : nat } ( i : nat -> nat )
( p :
isnattruncauto ( S upper ) i ) ( n : nat ) ( q : natleh n upper ) :
natleh ( truncnattruncauto p n ) upper.  Proof.  intros. unfold
truncnattruncauto. destruct ( natlthorgeh ( i n ) ( S upper) ) as
[ l
| r ]. apply natlthsntoleh. assumption. destruct ( natgehchoice ( i
n
) ( S upper ) ) as [ l' | r' ]. assert empty. apply ( isirreflnatlth
(
i n ) ). apply ( natlehlthtrans _ ( S upper ) ). apply p. apply
natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
natlthnsn. assumption. contradiction. destruct ( isdeceqnat n ( S
upper ) ) as [ l'' | r'' ]. assert empty. apply ( isirreflnatlth
upper
). apply ( natlthlehtrans _ ( S upper ) ). apply natlthnsn. rewrite
<-
```

```
l''. assumption. contradiction. assert ( natleh ( i ( S upper ) )
( S
upper ) ) as aux. apply p. apply isreflnatleh. destruct
( natlehchoice
_ _ aux ) as [ l''' | r''' ]. apply natlthsntoleh. assumption.
assert
empty. apply r''. apply ( nattruncautoisinj p ). apply
natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
natlthnsn. apply isreflnatleh. rewrite r'. rewrite r'''. apply
idpath. contradiction.  Defined.

Lemma truncnattruncautoisinj { upper : nat } { i : nat -> nat }
( p :
isnattruncauto ( S upper ) i ) { n m : nat } ( n' : natleh n upper )
(
m' : natleh m upper ) : truncnattruncauto p n ~> truncnattruncauto p
m
-> n ~> m.  Proof.  intros upper i p n m q r s.  apply (
nattruncautoisinj p ). apply natlthtoleh. apply ( natlehlthtrans _
upper ). assumption. apply natlthnsn. apply natlthtoleh. apply (
natlehlthtrans _ upper ). assumption. apply natlthnsn. unfold
truncnattruncauto in s. destruct ( natlthorgeh ( i n ) ( S upper ) )
as [ a0 | a1 ]. destruct ( natlthorgeh ( i m ) ( S upper ) ) as [ b0
|
b1 ]. assumption. assert empty. assert ( i m ~> S upper ) as f0.
destruct ( natgehchoice ( i m ) ( S upper ) b1 ) as [ l | l' ].
assert
empty.  apply ( isirreflnatlth ( S upper ) ). apply ( natlehlthtrans
_
( i n ) ). rewrite
s. assumption. assumption. contradiction. assumption.  destruct (
natgehchoice ( i m ) ( S upper ) b1 ) as [ a00 | a10 ]. apply (
isirreflnatgth ( S upper ) ). rewrite f0 in a00. assumption.
destruct
( isdeceqnat m ( S upper ) ) as [ a000 | a100 ]. rewrite s in
a0. rewrite f0 in a0. apply ( isirreflnatlth ( S upper )
). assumption.  assert ( i m ~> n ) as f1. apply ( nattruncautoisinj
p
).  rewrite f0. apply isreflnatleh. apply natlthtoleh. apply (
natlehlthtrans _ upper ). assumption. apply natlthnsn. rewrite f0.
rewrite s. apply idpath.  apply ( isirreflnatlth upper ). apply (
natlthlehtrans _ n ). rewrite <- f1, f0. apply
natlthnsn. assumption. contradiction.  destruct ( natgehchoice ( i
n )
( S upper ) a1 ) as [ a00 | a01 ].  assert empty. apply (
isirreflnatlth ( S upper ) ). apply ( natlthlehtrans _ ( i n )
). assumption. apply ( p ). apply natlthtoleh. apply
( natlehlthtrans
_ upper ). assumption. apply natlthnsn. contradiction.  destruct (
natlthorgeh ( i m ) ( S upper ) ) as [ b0 | b1 ]. destruct (
isdeceqnat n ( S upper ) ) as [ a000 | a001 ]. assumption. assert
( S
upper ~> m ) as f0. apply ( nattruncautoisinj p ). apply
isreflnatleh. apply natlthtoleh. apply ( natlehlthtrans _ upper
). assumption. apply natlthnsn. assumption. assert empty. apply
a001. rewrite f0. assert empty. apply ( isirreflnatlth ( S upper )
). apply ( natlehlthtrans _ upper ). rewrite f0. assumption. apply
natlthnsn. contradiction. contradiction.  destruct ( natgehchoice
( i
m ) ( S upper ) b1 ) as [ b00 | b01 ].  assert empty. apply (
isirreflnatlth ( i m ) ). apply ( natlehlthtrans _ ( S upper )
). apply p. apply ( natlthtoleh ). apply ( natlehlthtrans _ upper
). assumption. apply natlthnsn. assumption. contradiction. rewrite
b01.  rewrite a01. apply idpath.  Defined.

Lemma truncnattruncautoisauto { upper : nat } { i : nat -> nat }
( p :
isnattruncauto ( S upper ) i ) : isnattruncauto upper (
truncnattruncauto p ).  Proof.  intros. split. intros n q. assert (
natleh n ( S upper ) ) as q'. apply natlthtoleh. apply (
natlehlthtrans _ upper ). assumption. apply natlthnsn. destruct (
isdeceqnat ( nattruncautopreimage p q' ) ( S upper ) ) as [ i0 | i1
]. split with ( nattruncautopreimage p ( isreflnatleh ( S upper ) )
). split. assert ( natleh ( nattruncautopreimage p ( isreflnatleh
( S
upper ) ) ) ( S upper ) ) as aux. apply
nattruncautopreimageineq. destruct ( natlehchoice _ _ aux ) as [ l |
r
]. apply natlthsntoleh. assumption. assert ( n ~> S upper ) as
f0. rewrite <- ( nattruncautopreimagepath p q' ).  rewrite i0.
rewrite
<- r. rewrite ( nattruncautopreimagepath p ( isreflnatleh ( S
upper) )
). rewrite r. apply idpath. assert empty.  apply ( isirreflnatlth
( S
upper ) ). apply ( natlehlthtrans _ upper ).  rewrite <-
f0. assumption. apply natlthnsn. contradiction.

  split. apply ( nattruncautoisinj p ). apply natlthtoleh. apply (
  natlehlthtrans _ upper ). apply truncnattruncautobound.  destruct
(
  natlehchoice _ _ ( nattruncautopreimageineq p ( isreflnatleh ( S
  upper ) ) ) ) as [ l | r]. apply natlthsntoleh. assumption.
assert
  empty. assert ( S upper ~> n ) as f0. rewrite <- (
  nattruncautopreimagepath p ( isreflnatleh ( S upper ) ) ). rewrite
  r.  rewrite <- i0. rewrite ( nattruncautopreimagepath p q' ).
apply
  idpath. apply ( isirreflnatlth ( S upper ) ). apply
( natlehlthtrans
  _ upper ). rewrite f0. assumption. apply
  natlthnsn. contradiction. apply natlthnsn. assumption. unfold
  truncnattruncauto. destruct ( isdeceqnat ( nattruncautopreimage p
(
  isreflnatleh ( S upper ) ) ) ) as [ l | r ].  assert empty. assert
(
  S upper ~> n ) as f0. rewrite <- ( nattruncautopreimagepath p (
  isreflnatleh ( S upper ) ) ). rewrite l.  rewrite <- i0. rewrite (
  nattruncautopreimagepath p q' ). apply idpath. apply (
  isirreflnatlth ( S upper ) ). apply ( natlehlthtrans _ upper
  ). rewrite f0. assumption. apply natlthnsn. contradiction.
destruct
  ( natlthorgeh ( i ( nattruncautopreimage p ( isreflnatleh ( S
upper
  ) ) ) ) ( S upper ) ) as [ l' | r' ]. assert empty. apply (
  isirreflnatlth ( S upper ) ) . rewrite ( nattruncautopreimagepath
p
  ) in l'. assumption. contradiction. destruct ( natgehchoice _ _
r' )
  as [ l'' | r'' ]. assert empty. apply ( isirreflnatlth ( S
upper ) )
  . rewrite ( nattruncautopreimagepath p ) in
  l''. assumption. contradiction.  rewrite <- i0. rewrite (
  nattruncautopreimagepath p q' ). apply idpath.  intros x X y.
apply
  ( nattruncautoisinj p ). apply nattruncautopreimageineq. apply
  natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
  natlthnsn.  unfold truncnattruncauto in y.  destruct ( natlthorgeh
(
  i x ) ( S upper ) ) as [ l | r ].  assert ( S upper ~> x ) as
  f0. apply ( nattruncautoisinj p ). apply isreflnatleh. apply
  natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
  natlthnsn. rewrite <- i0.  rewrite y. rewrite (
  nattruncautopreimagepath p q' ). apply idpath. assert empty. apply
(
  isirreflnatlth ( S upper ) ). apply ( natlehlthtrans _ upper
  ). rewrite f0. assumption. apply natlthnsn. contradiction.
destruct
  ( isdeceqnat x ( S upper ) ) as [ l' | r' ]. assert empty. apply (
  isirreflnatlth ( S upper ) ). apply ( natlehlthtrans _ upper
  ). rewrite <- l'. assumption. apply
  natlthnsn. contradiction. destruct ( natgehchoice _ _ r ) as [ l''
|
  r'' ]. assert empty. apply ( isirreflnatlth n ). apply (
  natlehlthtrans _ ( S upper ) ). assumption. rewrite <-
  y. assumption. contradiction. rewrite ( nattruncautopreimagepath p
_
  ). rewrite r''.  apply idpath.  split with ( nattruncautopreimage
p
  q' ). split. destruct ( natlehchoice _ _
( nattruncautopreimageineq
  p q' ) ) as [ l | r ]. apply natlthsntoleh. assumption.  assert
  empty. apply i1. assumption. contradiction.  split. unfold
  truncnattruncauto.  destruct ( natlthorgeh ( i (
  nattruncautopreimage p q' ) ) ( S upper ) ) as [ l | r ]. apply
  nattruncautopreimagepath. destruct ( natgehchoice _ _ r ) as [ l'
|
  r' ]. apply nattruncautopreimagepath. assert empty. apply (
  isirreflnatlth ( S upper ) ). apply ( natlehlthtrans _ upper
  ). rewrite <- r'. rewrite ( nattruncautopreimagepath p q'
  ). assumption. apply natlthnsn. contradiction.
  intros x X y. apply ( nattruncautoisinj p ). apply ( pr1 p ).
apply
  natlthtoleh. apply ( natlehlthtrans _ upper ). assumption. apply
  natlthnsn.  rewrite ( nattruncautopreimagepath p q' ). unfold
  truncnattruncauto in y. destruct ( natlthorgeh ( i x ) ( S
upper ) )
  as [ l | r ]. rewrite y. apply idpath.  destruct ( isdeceqnat x
( S
  upper ) ) as [ l' | r' ]. assert empty. apply ( isirreflnatlth ( S
  upper ) ). apply ( natlehlthtrans _ upper ).  rewrite <-
  l'. assumption. apply natlthnsn. contradiction. destruct (
  natgehchoice _ _ r ). rewrite y. apply idpath. assert empty. apply
  i1.  apply ( nattruncautoisinj p ). apply
( nattruncautopreimageineq
  p ). apply isreflnatleh. rewrite ( nattruncautopreimagepath p q'
  ). rewrite y. apply idpath. contradiction.  apply
  truncnattruncautobound.  Defined.

Definition truncnattruncautoinv { upper : nat } { i : nat -> nat }
( p
: isnattruncauto ( S upper ) i ) : nat -> nat := nattruncautoinv (
truncnattruncautoisauto p ).

Lemma precompwithnatcofaceisauto { upper : nat } ( i : nat -> nat )
(
p : isnattruncauto ( S upper ) i ) ( bound : natlth 0 (
nattruncautopreimage p ( isreflnatleh ( S upper ) ) ) ) :
isnattruncauto upper (funcomp ( natcoface ( nattruncautopreimage p (
isreflnatleh ( S upper ) ) ) ) i ).  Proof.  intros. set ( v :=
nattruncautopreimage p ( isreflnatleh ( S upper ) ) ).  change (
nattruncautopreimage p ( isreflnatleh ( S upper ) ) ) with v in
bound. unfold isnattruncauto. split. intros m q.  unfold
funcomp. assert ( natleh m ( S upper ) ) as aaa. apply
natlthtoleh. apply natlehlthtrans with ( m := upper
). assumption. exact ( natlthnsn upper ).  set ( m' :=
nattruncautopreimage p aaa ).  destruct ( natlthorgeh m' v ) as [ l
|
r ].  (* CASE m' < v *) split with m'. split. apply natlthsntoleh.
apply ( natlthlehtrans _ v ). assumption. apply (
nattruncautopreimageineq p _ ).  split. unfold natcoface. rewrite
l. apply ( nattruncautopreimagepath p aaa ).  intros n j w. assert (
natcoface v n ~> m' ) as f0. apply pathsinv0.  apply (
nattruncautopreimagecanon p aaa ). apply
natcofaceleh. assumption. assumption. rewrite <- f0. destruct (
natgthorleh v n ) as [ l' | r' ]. unfold natcoface. rewrite l'.
apply
idpath.  assert empty. apply ( isirreflnatlth v ). apply (
natlehlthtrans _ n ). assumption. apply ( istransnatlth _ ( S n )
). apply natlthnsn. unfold natcoface in f0. rewrite (
natgehimplnatgtbfalse v n r' ) in f0. rewrite
f0. assumption. contradiction.  (* CASE v <= m' *) set ( j :=
nattruncautopreimagepath p aaa ). change ( nattruncautopreimage p
aaa
) with m' in j.  set ( m'' := minus m' 1 ). assert ( natleh m''
```

```
upper
) as a0. destruct ( natlthorgeh 0 m' ) as [ h | h' ]. rewrite <- (
minussn1 upper ). apply minus1leh. assumption.  apply
( natlehlthtrans
_ upper ). apply natleh0n. apply natlthnsn. apply (
nattruncautopreimageineq ). destruct ( natgehchoice 0 m' h' ) as [ k
|
k' ]. assert empty. apply ( negnatgth0n m' k ). contradiction.
unfold
m''. rewrite <- k'. apply natleh0n.  destruct ( natgehchoice m' v
r )
as [ l' | r' ]. assert ( natleh v m'' ) as a2. apply
natlthsntoleh. unfold m''. rewrite pathssminus. rewrite
minussn1. assumption. destruct ( natlehchoice 0 m' ( natleh0n m' ) )
as [ k | k' ]. assumption. assert empty. apply ( negnatgth0n v
). rewrite k'. assumption. contradiction.  assert ( i ( natcoface v
m'' ) ~> m ) as a1. unfold natcoface.  rewrite
( natgehimplnatgtbfalse
v m'' a2 ). unfold m''. rewrite pathssminus. rewrite
minussn1. assumption. destruct ( natlehchoice 0 m' ( natleh0n m' ) )
as [ k | k' ]. assumption. assert empty. apply ( negnatgth0n v
). rewrite k'. assumption. contradiction.  split with
m''. split. assumption. split. assumption. intros n s t.  assert (
natcoface v n ~> natcoface v m'' ) as g. assert ( natcoface v n ~>
m'
) as g0. apply pathsinv0. apply ( nattruncautopreimagecanon p aaa
). apply natcofaceleh. assumption. assumption.  assert ( natcoface v
m'' ~> m' ) as g1. unfold m'. unfold nattruncautopreimage. apply
pathsinv0. apply ( nattruncautopreimagecanon p aaa ). apply
natcofaceleh. assumption.  assumption. rewrite g0, g1. apply idpath.
change ( idfun _ m'' ~> idfun _ n ). rewrite <- (
natcofaceretractisretract v ). unfold funcomp. rewrite g. apply
idpath.  assert empty. apply ( isirreflnatlth ( S upper ) ). apply (
natlehlthtrans _ upper ). assert ( S upper ~> m ) as g. rewrite <- (
nattruncautopreimagepath p ( isreflnatleh ( S upper ) ) ). change
( i
v ~> m ).  rewrite <- j. rewrite r'. apply idpath. rewrite
g. assumption. apply natlthnsn. contradiction.

      intros x X. unfold funcomp. assert ( natleh ( i ( natcoface v
x
      ) ) ( S upper ) ) as a0. apply p. apply
      natcofaceleh. assumption.  destruct ( natlehchoice _ _ a0 ) as
[
      l | r ]. apply natlthsntoleh. assumption. assert ( v ~>
      natcoface v x ) as g. unfold v. apply (
      nattruncautopreimagecanon p ( isreflnatleh ( S upper ) )
      ). unfold natcoface. destruct ( natgthorleh v x ) as [ a | b
      ]. unfold v in a. rewrite a. apply natlthtoleh. apply (
      natlehlthtrans _ upper ). assumption. apply natlthnsn. unfold
v
      in b. rewrite ( natgehimplnatgtbfalse _ x b ). assumption.
      assumption. assert empty. destruct ( natgthorleh v x ) as [ a
|
      b ].  unfold natcoface in g. rewrite a in g. apply (
      isirreflnatlth x ). rewrite g in a. assumption. unfold
natcoface
      in g. rewrite ( natgehimplnatgtbfalse v x b ) in g. apply (
      isirreflnatlth x ). apply ( natlthlehtrans _ ( S x ) ). apply
      natlthnsn. rewrite <- g. assumption. contradiction.  Defined.

Lemma nattruncautocompstable { R : commrng } { upper : nat } ( i j :
nat -> nat ) ( p : isnattruncauto upper i ) ( p' : isnattruncauto
upper j ) : isnattruncauto upper ( funcomp j i ).  Proof.
intros. split. intros n n'. split with ( nattruncautopreimage p' (
nattruncautopreimageineq p n' ) ).  split. apply (
nattruncautopreimageineq p' ). split. unfold funcomp.  rewrite (
nattruncautopreimagepath p' _ ). rewrite ( nattruncautopreimagepath
p
_ ). apply idpath. intros x X y. unfold funcomp in y.  apply (
nattruncautoisinj p' ). apply
nattruncautopreimageineq. assumption. apply ( nattruncautoisinj p
). apply p'. apply nattruncautopreimageineq. apply
p'. assumption. rewrite ( nattruncautopreimagepath p' ). rewrite (
nattruncautopreimagepath p ). rewrite y. apply idpath.  intros x
X. unfold funcomp. apply p. apply p'. assumption.  Defined.

Definition nattruncreverse ( upper : nat ) : nat -> nat.  Proof.
intros upper n. destruct ( natgthorleh n upper ) as [ h | k ]. exact
n. exact ( minus upper n ).  Defined.

Definition nattruncbottomtopswap ( upper : nat ) : nat -> nat.
Proof.
intros upper n. destruct ( isdeceqnat 0%nat n ) as [ h | k ]. exact
(
upper ).  destruct ( isdeceqnat upper n ) as [ l | r ]. exact
( 0%nat
). exact n.  Defined.

Lemma nattruncreverseisnattruncauto ( upper : nat ) : isnattruncauto
upper ( nattruncreverse upper ).  Proof.  intros. unfold
isnattruncauto. split. intros m q.  set ( m' := minus upper m
). assert ( natleh m' upper ) as a0. apply minusleh.  assert (
nattruncreverse upper m' ~> m ) as a1.  unfold
nattruncreverse. destruct ( natgthorleh m' upper ). assert
empty. apply isirreflnatlth with ( n := m' ).  apply natlehlthtrans
with ( m := upper ).  assumption. assumption. contradiction. unfold
m'. rewrite doubleminuslehpaths. apply idpath. assumption. split
with
m'. split. assumption. split. assumption.  intros n qq u. unfold
m'. rewrite <- u. unfold nattruncreverse.  destruct ( natgthorleh n
upper ) as [ l | r ].  assert empty. apply ( isirreflnatlth n ).
apply
( natlehlthtrans _ upper
). assumption. assumption. contradiction. rewrite
doubleminuslehpaths. apply idpath. assumption.  intros x X. unfold
nattruncreverse. destruct ( natgthorleh x upper ) as [ l | r
]. assumption. apply minusleh.  Defined.
```

```
Lemma nattruncbottomtopswapselfinv ( upper n : nat ) :
nattruncbottomtopswap upper ( nattruncbottomtopswap upper n ) ~> n.
Proof.  intros. unfold nattruncbottomtopswap. destruct ( isdeceqnat
upper n ).  destruct ( isdeceqnat 0%nat n ). destruct ( isdeceqnat
0%nat upper ). rewrite <- i0. rewrite <- i1. apply idpath.  assert
empty. apply e. rewrite i0. rewrite i. apply idpath. contradiction.
destruct ( isdeceqnat 0%nat 0%nat ). assumption. assert empty. apply
e0. auto. contradiction.  destruct ( isdeceqnat 0%nat n ). destruct
(
isdeceqnat 0%nat upper ). rewrite <- i. rewrite i0. apply idpath.
destruct ( isdeceqnat upper upper ). assumption. assert empty. apply
e1. auto. contradiction.  destruct ( isdeceqnat 0%nat n ). assert
empty. apply e0. assumption. contradiction.  destruct ( isdeceqnat
upper n ). assert empty. apply e. assumption. contradiction. auto.
Defined.

Lemma nattruncbottomtopswapbound ( upper n : nat ) ( p : natleh n
upper ) : natleh (nattruncbottomtopswap upper n ) upper.  Proof.
intros. unfold nattruncbottomtopswap. destruct (isdeceqnat 0%nat n
). auto.  destruct ( isdeceqnat upper n ). apply isreflnatleh. apply
isreflnatleh. destruct ( isdeceqnat upper n ). apply
natleh0n. assumption.  Defined.

Lemma nattruncbottomtopswapisnattruncauto ( upper : nat ) :
isnattruncauto upper ( nattruncbottomtopswap upper ).  Proof.
intros. unfold isnattruncauto. split.  intros m p. set ( m' :=
nattruncbottomtopswap upper m ).  assert ( natleh m' upper ) as
a0. apply nattruncbottomtopswapbound. assumption.  assert
(nattruncbottomtopswap upper m' ~> m) as a1.  apply
nattruncbottomtopswapselfinv.  split with
m'. split. assumption. split. assumption.  intros k q u. unfold
m'. rewrite <- u. rewrite nattruncbottomtopswapselfinv. apply
idpath. intros n p. apply nattruncbottomtopswapbound. assumption.
Defined.

Lemma isnattruncauto0S { upper : nat } { i : nat -> nat } ( p :
isnattruncauto (S upper) i ) ( j : i 0%nat ~> S upper ) :
isnattruncauto upper ( funcomp S i ).  Proof.  intros. unfold
isnattruncauto. split. intros m q.  set ( v := nattruncautopreimage
p
(natlthtoleh m (S upper) (natlehlthtrans m upper (S upper) q
(natlthnsn upper)))).  destruct ( isdeceqnat 0%nat v ) as [ i0 | i1
]. assert empty. apply ( isirreflnatlth ( i 0%nat ) ). apply (
natlehlthtrans _ upper ). rewrite i0. unfold v. rewrite (
nattruncautopreimagepath ). assumption. rewrite j. apply
natlthnsn. contradiction.  assert ( natlth 0 v ) as aux. destruct (
natlehchoice _ _ ( natleh0n v ) ). assumption. assert empty. apply
i1. assumption. contradiction.  split with ( minus v 1
). split. rewrite <- ( minussn1 upper ).  apply ( minus1leh aux (
natlehlthtrans _ _ _ ( natleh0n upper ) ( natlthnsn upper ) ) (
nattruncautopreimageineq p ( natlthtoleh m ( S upper ) (
natlehlthtrans m upper ( S upper ) q ( natlthnsn upper ) ) ) )
). split. unfold funcomp.  rewrite pathssminus. rewrite
```

```
minussn1. apply nattruncautopreimagepath. assumption.  intros n uu
k. unfold funcomp in k. rewrite <- ( minussn1 n ).  assert ( v ~> S
n
) as f.  apply ( nattruncautopreimagecanon p _
). assumption. assumption. rewrite f.  apply idpath.  intros x
X. unfold funcomp. assert ( natleh ( i ( S x ) ) ( S upper ) ) as
aux. apply p. assumption. destruct ( natlehchoice _ _ aux ) as [ h |
k
]. apply natlthsntoleh. assumption. assert empty. assert ( 0%nat ~>
S
x ) as ii. apply ( nattruncautoisinj p ). apply natleh0n.
assumption.
rewrite j. rewrite k. apply idpath. apply ( isirreflnatlth ( S
x ) ).
apply ( natlehlthtrans _ x ). rewrite <- ii. apply natleh0n. apply
natlthnsn. contradiction.  Defined.

(* The following lemma says that we may reindex sums along
automorphisms of the interval over which the finite summation is
being
taken. *)

Lemma natsummationreindexing { R : commrng } { upper : nat } ( i :
nat
-> nat ) ( p : isnattruncauto upper i ) ( f : nat -> R ) :
natsummation0 upper f ~> natsummation0 upper (funcomp i f ).  Proof.
intros R upper. induction upper. intros. simpl. unfold funcomp.
assert ( 0%nat ~> i 0%nat ) as f0. destruct ( natlehchoice ( i
0%nat )
0%nat ( pr2 p 0%nat ( isreflnatleh 0%nat ) ) ) as [ h | k ]. assert
empty. exact ( negnatlthn0 ( i 0%nat ) h ). contradiction. rewrite
k. apply idpath. rewrite <- f0. apply idpath.  intros. simpl (
natsummation0 ( S upper ) f ).  set ( j := nattruncautopreimagepath
p
( isreflnatleh ( S upper ) ) ).  set ( v := nattruncautopreimage p (
isreflnatleh ( S upper ) ) ).  change ( nattruncautopreimage p (
isreflnatleh ( S upper ) ) ) with v in j.  destruct ( natlehchoice
0%nat v ( natleh0n v ) ).  set ( aaa := nattruncautopreimageineq p (
isreflnatleh ( S upper ) ) ). change ( nattruncautopreimage p (
isreflnatleh ( S upper ) ) ) with v in aaa.  destruct ( natlehchoice
v
( S upper ) aaa ) as [ l | r ].  rewrite ( IHupper ( funcomp (
natcoface v ) i ) ).

  change ( funcomp ( funcomp ( natcoface v ) i ) f ) with ( funcomp
(
  natcoface v ) ( funcomp i f ) ).  assert ( f ( S upper ) ~> (
  funcomp i f ) v ) as f0.  unfold funcomp.  rewrite j. apply
  idpath. rewrite f0.

  assert ( natleh v upper ) as aux. apply natlthsntoleh. assumption.
  rewrite ( natsummationshift upper ( funcomp i f ) aux ). apply
  idpath.  apply precompwithnatcofaceisauto. assumption.
  rewrite ( IHupper ( funcomp ( natcoface v ) i ) ). assert (
  natsummation0 upper ( funcomp ( funcomp ( natcoface v ) i) f ) ~>
  natsummation0 upper ( funcomp i f ) ) as f0.  apply
  natsummationpathsupperfixed. intros x X. unfold funcomp. unfold
  natcoface. assert ( natlth x v ) as a0. apply ( natlehlthtrans _
  upper ). assumption. rewrite r. apply natlthnsn. rewrite a0. apply
  idpath. rewrite f0. assert ( f ( S upper ) ~> ( funcomp i f ) ( S
  upper ) ) as f1. unfold funcomp.  rewrite <- r. rewrite j. rewrite
  <- r. apply idpath. rewrite f1. apply idpath.  apply
  precompwithnatcofaceisauto. assumption.  rewrite
  natsummationshift0. unfold funcomp at 2. rewrite i0. rewrite j.
  assert ( i 0%nat ~> S upper ) as j'. rewrite i0. rewrite j. apply
  idpath.  rewrite ( IHupper ( funcomp S i ) ( isnattruncauto0S p
j' )
  ). apply idpath.  Defined.

(** * III. Formal Power Series *)

Definition seqson ( A : UU ) := nat -> A.

Lemma seqsonisaset ( A : hSet ) : isaset ( seqson A ).  Proof.
intros. unfold seqson. change ( isofhlevel 2 ( nat -> A ) ). apply
impredfun. apply A.  Defined.

Definition isasetfps ( R : commrng ) : isaset ( seqson R ) :=
seqsonisaset R.

Definition fps ( R : commrng ) : hSet := hSetpair _ ( isasetfps R ).

Definition fpsplus ( R : commrng ) : binop ( fps R ) := fun v w n =>
(
( v n ) + ( w n ) ).

Definition fpstimes ( R : commrng ) : binop ( fps R ) := fun s t n
=>
natsummation0 n ( fun x : nat => ( s x ) * ( t ( minus n x ) ) ).

(* SOME TESTS OF THE SUMMATION AND FPSTIMES DEFINITIONS:

Definition test0 : seqson hz.  Proof.  intro n. induction n. exact
0. exact ( nattohz ( S n ) ).  Defined.

Eval lazy in ( hzabsval ( natsummation0 1 test0 ) ).

Definition test1 : seqson hz.  Proof.  intro n. induction n. exact
( 1
+ 1 ). exact ( ( 1 + 1 ) * IHn ).  Defined.

Eval lazy in ( hzabsval ( fpstimes hz test0 test1 0%nat ) ).

Eval lazy in ( hzabsval ( fpstimes hz test0 test1 1%nat ) ).

Eval lazy in ( hzabsval ( fpstimes hz test0 test1 2%nat ) ).
Eval lazy in ( hzabsval ( fpstimes hz test0 test1 3%nat ) ).

Eval lazy in ( hzabsval ( fpstimes hz test0 test1 4%nat ) ).  *)

Definition fpszero ( R : commrng ) : fps R := ( fun n : nat => 0 ).

Definition fpsone ( R : commrng ) : fps R.  Proof.  intros. intro
n. destruct n. exact 1. exact 0.  Defined.

Definition fpsminus ( R : commrng ) : fps R -> fps R := ( fun s n =>
-
( s n ) ).

Lemma ismonoidopfpsplus ( R : commrng ) : ismonoidop ( fpsplus R ).
Proof.  intros. unfold ismonoidop. split.  unfold isassoc.  intros s
t
u. unfold fpsplus.  (* This is a hack which should work immediately
without such a workaround! *) change ( (fun n : nat => s n + t n + u
n) ~> (fun n : nat => s n + (t n + u n)) ). apply funextfun.  intro
n. apply R.

      unfold isunital. assert ( isunit ( fpsplus R ) ( fpszero R ) )
      as a.  unfold isunit.  split.  unfold islunit. intro s. unfold
      fpsplus. unfold fpszero.  change ( (fun n : nat => 0 + s n) ~>
s
      ). apply funextfun.  intro n. apply rnglunax1.

        unfold isrunit. intro s. unfold fpsplus. unfold fpszero.
        change ( (fun n : nat => s n + 0) ~> s ). apply funextfun.
        intro n. apply rngrunax1.  exact ( tpair ( fpszero R ) a ).
        Defined.

Lemma isgropfpsplus ( R : commrng ) : isgrop ( fpsplus R ).  Proof.
intros. unfold isgrop.  assert ( invstruct ( fpsplus R ) (
ismonoidopfpsplus R ) ) as a.  unfold invstruct.  assert ( isinv
(fpsplus R) (unel_is (ismonoidopfpsplus R)) ( fpsminus R ) ) as b.
unfold isinv. split.  unfold islinv. intro s. unfold fpsplus. unfold
fpsminus. unfold unel_is.  simpl. unfold fpszero. apply
funextfun. intro n. exact ( rnglinvax1 R ( s n ) ).

    unfold isrinv. intro s. unfold fpsplus. unfold fpsminus. unfold
    unel_is.  simpl. unfold fpszero. apply funextfun. intro n. exact
(
    rngrinvax1 R ( s n ) ).  exact ( tpair ( fpsminus R ) b ).
exact
    ( tpair ( ismonoidopfpsplus R ) a ).  Defined.

Lemma iscommfpsplus ( R : commrng ) : iscomm ( fpsplus R ).  Proof.
intros. unfold iscomm.  intros s t. unfold fpsplus.  change ((fun
n :
nat => s n + t n) ~> (fun n : nat => t n + s n) ).  apply
funextfun. intro n. apply R.  Defined.

Lemma isassocfpstimes ( R : commrng ) : isassoc (fpstimes R).
```

```
Proof.
intros. unfold isassoc. intros s t u.  unfold fpstimes.

  assert ( (fun n : nat => natsummation0 n (fun x : nat =>
    natsummation0 ( minus n x) (fun x0 : nat => s x * ( t x0 * u (
    minus ( minus n x ) x0) )))) ~> (fun n : nat => natsummation0 n
    (fun x : nat => s x * natsummation0 ( minus n x) (fun x0 : nat
=>
    t x0 * u ( minus ( minus n x ) x0)))) ) as A.  apply
    funextfun. intro n. apply
    natsummationpathsupperfixed. intros. rewrite
    natsummationtimesdistr. apply idpath. rewrite <- A.  assert
( (fun
    n : nat => natsummation0 n (fun x : nat => natsummation0 ( minus
n
    x) (fun x0 : nat => s x * t x0 * u ( minus ( minus n x ) x0 ))))
    ~> (fun n : nat => natsummation0 n (fun x : nat => natsummation0
(
    minus n x) (fun x0 : nat => s x * ( t x0 * u ( minus ( minus n
x )
    x0) )))) ) as B.  apply funextfun. intro n. apply
    maponpaths. apply funextfun. intro m. apply maponpaths. apply
    funextfun. intro x. apply R.  assert ( (fun n : nat =>
    natsummation0 n (fun x : nat => natsummation0 x (fun x0 : nat =>
s
    x0 * t ( minus x x0 ) * u ( minus n x )))) ~> (fun n : nat =>
    natsummation0 n (fun x : nat => natsummation0 ( minus n x) (fun
x0
    : nat => s x * t x0 * u ( minus ( minus n x ) x0 )))) ) as C.
    apply funextfun. intro n.  set ( f := fun x : nat => ( fun x0 :
    nat => s x * t x0 * u ( minus ( minus n x ) x0 ) ) ).  assert (
    natsummation0 n ( fun x : nat => natsummation0 x ( fun x0 : nat
=>
    f x0 ( minus x x0 ) ) ) ~> ( natsummation0 n ( fun x : nat =>
    natsummation0 ( minus n x ) ( fun x0 : nat => f x x0 ) ) ) ) as
D.
    apply natsummationswap. unfold f in D.  assert ( natsummation0 n
(
    fun x : nat => natsummation0 x ( fun x0 : nat => s x0 * t
( minus
    x x0 ) * u ( minus n x ) ) ) ~> natsummation0 n ( fun x : nat =>
    natsummation0 x ( fun x0 : nat => s x0 * t ( minus x x0 ) * u (
    minus ( minus n x0 ) ( minus x x0 ) ) ) ) ) as E.  apply
    natsummationpathsupperfixed. intros k p.  apply
    natsummationpathsupperfixed. intros l q. rewrite
( natdoubleminus
    p q ). apply idpath. rewrite E, D. apply idpath. rewrite <-
    B. rewrite <- C.  assert ( (fun n : nat => natsummation0 n (fun
x
    : nat => natsummation0 x (fun x0 : nat => s x0 * t ( minus x
x0))
    * u ( minus n x))) ~> (fun n : nat => natsummation0 n (fun x :
nat
    => natsummation0 x (fun x0 : nat => s x0 * t ( minus x x0) * u (
    minus n x)))) ) as D.  apply funextfun. intro n. apply
    maponpaths. apply funextfun.  intro m. apply
    natsummationtimesdistl. rewrite <- D. apply idpath.  Defined.

Lemma natsummationandfpszero ( R : commrng ) : forall m : nat,
natsummation0 m ( fun x : nat => fpszero R x ) ~> ( @rngunel1 R ).
Proof.  intros R m. induction m. apply idpath. simpl. rewrite
IHm. rewrite ( rnglunax1 R ). apply idpath.  Defined.

Lemma ismonoidopfpstimes ( R : commrng ) : ismonoidop ( fpstimes
R ).
Proof.  intros. unfold ismonoidop.  split. apply
isassocfpstimes. split with ( fpsone R). split. intro s.  unfold
fpstimes. change ( ( fun n : nat => natsummation0 n ( fun x : nat =>
fpsone R x * s ( minus n x ) ) ) ~> s ).  apply funextfun. intro
n. destruct n. simpl. rewrite ( rnglunax2 R ). apply idpath. rewrite
natsummationshift0. rewrite ( rnglunax2 R). rewrite minus0r. assert
(
natsummation0 n ( fun x : nat => fpsone R ( S x ) * s ( minus n
x ) )
~> ( ( natsummation0 n ( fun x : nat => fpszero R x ) ) ) ) as
f. apply natsummationpathsupperfixed. intros m m'. rewrite
( rngmult0x
R ). apply idpath.  change ( natsummation0 n ( fun x : nat => fpsone
R
( S x ) * s ( minus n x ) ) + s ( S n ) ~> ( s ( S n ) ) ). rewrite
f. rewrite natsummationandfpszero. apply ( rnglunax1 R ).

 intros s. unfold fpstimes. change ( ( fun n : nat => natsummation0
n
  ( fun x : nat => s x * fpsone R ( minus n x ) ) ) ~> s ).  apply
  funextfun. intro n. destruct n. simpl. rewrite ( rngrunax2 R
  ). apply idpath. change ( natsummation0 n ( fun x : nat => s x *
  fpsone R ( minus ( S n ) x ) ) + s ( S n ) * fpsone R ( minus n
n )
  ~> s ( S n ) ).  rewrite minusnn0. rewrite ( rngrunax2 R ). assert
(
  natsummation0 n ( fun x : nat => s x * fpsone R ( minus ( S n )
x ))
  ~> ( ( natsummation0 n ( fun x : nat => fpszero R x ) ) ) ) as
  f. apply natsummationpathsupperfixed. intros m m'.  rewrite <-
  pathssminus. rewrite ( rngmultx0 R ). apply idpath. apply (
  natlehlthtrans _ n ). assumption. apply natlthnsn. rewrite
  f. rewrite natsummationandfpszero. apply ( rnglunax1 R ).
Defined.

Lemma iscommfpstimes ( R : commrng ) ( s t : fps R ) : fpstimes R s
t
~> fpstimes R t s.  Proof.  intros. unfold fpstimes.  change ( ( fun
n
: nat => natsummation0 n (fun x : nat => s x * t ( minus n x) ) ) ~>
(fun n : nat => natsummation0 n (fun x : nat => t x * s ( minus n
x)))
). apply funextfun. intro n.
```

```
  assert ( natsummation0 n ( fun x : nat => s x * t ( minus n x ) )
~>
  ( natsummation0 n ( fun x : nat => t ( minus n x ) * s x ) ) ) as
  a0.  apply maponpaths. apply funextfun. intro m. apply R.  assert
(
  ( natsummation0 n ( fun x : nat => t ( minus n x ) * s x ) ) ~> (
  natsummation0 n ( funcomp ( nattruncreverse n ) ( fun x : nat => t
x
  * s ( minus n x ) ) ) ) ) as a1.

  apply natsummationpathsupperfixed. intros m q. unfold
  funcomp. unfold nattruncreverse. destruct (natgthorleh m n ).
  assert empty. apply isirreflnatlth with ( n := n ). apply
  natlthlehtrans with ( m := m ). apply h. assumption.
contradiction.
  apply maponpaths. apply maponpaths. apply pathsinv0. apply
  doubleminuslehpaths.  assumption.  assert ( ( natsummation0 n (
  funcomp ( nattruncreverse n ) ( fun x : nat => t x * s ( minus n
x )
  ) ) ) ~> natsummation0 n ( fun x : nat => t x * s ( minus n
x ) ) )
  as a2. apply pathsinv0. apply natsummationreindexing. apply
  nattruncreverseisnattruncauto. exact ( pathscomp0 a0 ( pathscomp0
a1
  a2 ) ).  Defined.

Lemma isldistrfps ( R : commrng ) ( s t u : fps R ) : fpstimes R s (
fpsplus R t u ) ~> ( fpsplus R ( fpstimes R s t ) ( fpstimes R s u )
).  Proof.  intros. unfold fpstimes. unfold fpsplus. change ((fun
n :
nat => natsummation0 n (fun x : nat => s x * (t (minus n x) + u (
minus n x)))) ~> (fun n : nat => natsummation0 n (fun x : nat => s x
*
t (minus n x)) + natsummation0 n (fun x : nat => s x * u (minus n
x)))
).  apply funextfun. intro upper.  assert ( natsummation0 upper
( fun
x : nat => s x * ( t ( minus upper x ) + u ( minus upper x ) ) ) ~>
(
natsummation0 upper ( fun x : nat => ( ( s x * t ( minus upper x ) )
+
( s x * u ( minus upper x ) ) ) ) ) ) as a0.  apply maponpaths.
apply
funextfun. intro n. apply R.  assert ( ( natsummation0 upper ( fun
x :
nat => ( ( s x * t ( minus upper x ) ) + ( s x * u ( minus upper
x ) )
) ) ) ~> ( ( natsummation0 upper ( fun x : nat => s x * t ( minus
upper x ) ) ) + ( natsummation0 upper ( fun x : nat => s x * u
( minus
upper x ) ) ) ) ) as a1.  apply natsummationplusdistr.  exact (
pathscomp0 a0 a1 ).  Defined.
```

```
Lemma isrdistrfps ( R : commrng ) ( s t u : fps R ) : fpstimes R (
fpsplus R t u ) s ~> ( fpsplus R ( fpstimes R t s ) ( fpstimes R u
s )
).  Proof.  intros. unfold fpstimes. unfold fpsplus. change ((fun
n :
nat => natsummation0 n (fun x : nat => (t x + u x) * s ( minus n x )
)) ~> (fun n : nat => natsummation0 n (fun x : nat => t x * s
( minus
n x ) ) + natsummation0 n (fun x : nat => u x * s (minus n x))) ).
apply funextfun. intro upper.  assert ( natsummation0 upper ( fun
x :
nat => ( t x + u x ) * s ( minus upper x ) ) ~> ( natsummation0
upper
( fun x : nat => ( ( t x * s ( minus upper x ) ) + ( u x * s ( minus
upper x ) ) ) ) ) ) as a0.  apply maponpaths. apply funextfun. intro
n. apply R.  assert ( ( natsummation0 upper ( fun x : nat => ( ( t x
*
s ( minus upper x ) ) + ( u x * s ( minus upper x ) ) ) ) ) ~> ( (
natsummation0 upper ( fun x : nat => t x * s ( minus upper x ) ) ) +
(
natsummation0 upper ( fun x : nat => u x * s ( minus upper
x ) ) ) ) )
as a1.  apply natsummationplusdistr.  exact ( pathscomp0 a0 a1 ).
Defined.

Definition fpsrng ( R : commrng ) := setwith2binoppair ( hSetpair (
seqson R ) ( isasetfps R ) ) ( dirprodpair ( fpsplus R ) ( fpstimes
R
) ).

Theorem fpsiscommrng ( R : commrng ) : iscommrng ( fpsrng R ).
Proof.
intro. unfold iscommrng. unfold iscommrngops. split. unfold
isrngops. split. split.  unfold isabgrop. split. exact
( isgropfpsplus
R ). exact ( iscommfpsplus R ). exact ( ismonoidopfpstimes R ).
unfold isdistr. split. unfold isldistr. intros. apply ( isldistrfps
R
).  unfold isrdistr. intros. apply ( isrdistrfps R ). unfold
iscomm. intros. apply ( iscommfpstimes R ).  Defined.

Definition fpscommrng ( R : commrng ) : commrng := commrngpair (
fpsrng R ) ( fpsiscommrng R ).

Definition fpsshift { R : commrng } ( a : fpscommrng R ) :
fpscommrng
R := fun n : nat => a ( S n ).

Lemma fpsshiftandmult { R : commrng } ( a b : fpscommrng R ) ( p : b
0%nat ~> 0 ) : forall n : nat, ( a * b ) ( S n ) ~> ( ( a *
( fpsshift
b ) ) n ).  Proof.  intros. induction n. change ( a * b ) with (
fpstimes R a b ). change ( a * fpsshift b ) with ( fpstimes R a (
fpsshift b ) ).  unfold fpstimes. unfold fpsshift. simpl. rewrite
p. rewrite ( rngmultx0 R ). rewrite ( rngrunax1 R ). apply idpath.
change ( a * b ) with ( fpstimes R a b ). change ( a * fpsshift b )
with ( fpstimes R a ( fpsshift b ) ). unfold fpsshift.  unfold
fpstimes. change ( natsummation0 (S (S n)) (fun x : nat => a x * b
(minus ( S (S n) ) x)) ) with ( ( natsummation0 ( S n ) ( fun x :
nat
=> a x * b ( minus ( S ( S n ) ) x ) ) ) + a ( S ( S n ) ) * b
( minus
( S ( S n ) ) ( S ( S n ) ) ) ).  rewrite minusnn0. rewrite p.
rewrite
( rngmultx0 R ). rewrite rngrunax1.  apply
natsummationpathsupperfixed. intros x j.  apply maponpaths. apply
maponpaths. rewrite pathssminus.  apply idpath. apply
( natlehlthtrans
_ ( S n ) _ ). assumption. apply natlthnsn.  Defined.

(** * IV. Apartness relation on formal power series *)

Lemma apartbinarysum0 ( R : acommrng ) ( a b : R ) ( p : a + b #
0 ) :
hdisj ( a # 0 ) ( b # 0 ).  Proof.  intros. intros P s. apply (
acommrng_acotrans R ( a + b ) a 0 p ). intro k.  destruct k as [ l |
r
]. apply s. apply ii2. assert ( a + b # a ) as l'. apply l.  assert
(
( a + b ) # ( a + 0 ) ) as l''. rewrite rngrunax1. assumption.
apply
( ( pr1 ( acommrng_aadd R ) ) a b 0 ). assumption. apply s.  apply
ii1. assumption.  Defined.

Lemma apartnatsummation0 ( R : acommrng ) ( upper : nat ) ( f : nat
->
R ) ( p : ( natsummation0 upper f ) # 0 ) : hexists ( fun n : nat =>
dirprod ( natleh n upper ) ( f n # 0 ) ).  Proof.  intros R
upper. induction upper. simpl. intros. intros P s. apply s. split
with
0%nat.  split. intros g. simpl in g. apply
nopathsfalsetotrue. assumption. assumption.  intros. intros P s.
simpl
in p. apply ( apartbinarysum0 R _ _ p ).  intro k. destruct k as [ l
|
r ]. apply ( IHupper f l ). intro k.  destruct k as [ n ab ].
destruct
ab as [ a b ]. apply s. split with n. split.  apply ( istransnatleh
_
upper _ ). assumption. apply natlthtoleh.  apply
natlthnsn. assumption. apply s. split with ( S upper ).  split.
apply
isreflnatleh. assumption.  Defined.

Definition fpsapart0 ( R : acommrng ) : hrel ( fpscommrng R ) := fun
s
t : fpscommrng R => ( hexists ( fun n : nat => ( s n # t n ) ) ).
```

```
Definition fpsapart ( R : acommrng ) : apart ( fpscommrng R ).
Proof.
intros. split with ( fpsapart0 R ).  split. intros s f. assert (
hfalse ) as i. apply f. intro k. destruct k as [ n p ].  apply (
acommrng_airrefl R ( s n ) p). apply i.  split. intros s t p P
j. apply p. intro k. destruct k as [ n q ].  apply j. split with
n. apply ( acommrng_asymm R ( s n ) ( t n ) q ).  intros s t u p P
j. apply p. intro k. destruct k as [ n q ].  apply
( acommrng_acotrans
R ( s n ) ( t n ) ( u n ) q ).  intro l. destruct l as [ l | r ].
apply j. apply ii1. intros v V. apply V. split with n. assumption.
apply j. apply ii2. intros v V. apply V. split with n. assumption.
Defined.

Lemma fpsapartisbinopapartplusl ( R : acommrng ) : isbinopapartl (
fpsapart R ) ( @op1 ( fpscommrng R ) ).  Proof.  intros. intros a b
c
p. intros P s. apply p. intro k.  destruct k as [ n q ]. apply
s. change ( ( a + b ) n ) with ( ( a n ) + ( b n ) ) in q.  change
( (
a + c ) n ) with ( ( a n ) + ( c n ) ) in q.  split with n. apply
( (
pr1 ( acommrng_aadd R ) ) ( a n ) ( b n ) ( c n ) q ).  Defined.

Lemma fpsapartisbinopapartplusr ( R : acommrng ) : isbinopapartr (
fpsapart R ) ( @op1 ( fpscommrng R ) ).  Proof.  intros. intros a b
c
p. rewrite ( rngcomm1 ( fpscommrng R ) ) in p. rewrite ( rngcomm1 (
fpscommrng R ) c ) in p.  apply ( fpsapartisbinopapartplusl _ a b c
). assumption.  Defined.

Lemma fpsapartisbinopapartmultl ( R : acommrng ) : isbinopapartl (
fpsapart R ) ( @op2 ( fpsrng R ) ).  Proof.  intros. intros a b c
p. intros P s. apply p. intro k.  destruct k as [ n q ]. change
( ( a
* b ) n ) with ( natsummation0 n ( fun x : nat => ( a x ) * ( b (
minus n x ) ) ) ) in q.  change ( ( a * c ) n ) with ( natsummation0
n
( fun x : nat => ( a x ) * ( c ( minus n x ) ) ) ) in q.  assert (
natsummation0 n ( fun x : nat => ( a x * b ( minus n x ) - ( a x * c
(
minus n x ) ) ) ) # 0 ) as q'. assert ( natsummation0 n ( fun x :
nat
=> ( a x * b ( minus n x ) ) ) - natsummation0 n ( fun x : nat =>
( a
x * c ( minus n x ) ) ) # 0 ) as q''. apply aaminuszero. assumption.
assert ( (fun x : nat => a x * b (minus n x) - a x * c ( minus n x))
~> (fun x : nat => a x * b ( minus n x) + ( - 1%rng ) * ( a x * c (
minus n x)) ) ) as i.  apply funextfun. intro x. apply
maponpaths. rewrite <- ( rngmultwithminus1 R ). apply idpath.
rewrite
i. rewrite natsummationplusdistr. rewrite <-
( natsummationtimesdistr
n ( fun x : nat => a x * c ( minus n x ) ) ( - 1%rng ) ).  rewrite (
rngmultwithminus1 R ). assumption.  apply ( apartnatsummation0 R n _
q' ). intro k.  destruct k as [ m g ]. destruct g as [ g g' ].
apply
s. split with ( minus n m ). apply ( ( pr1 ( acommrng_amult R ) )
( a
m ) ( b ( minus n m ) ) ( c ( minus n m ) ) ). apply
aminuszeroa. assumption.  Defined.

Lemma fpsapartisbinopapartmultr ( R : acommrng ) : isbinopapartr (
fpsapart R ) ( @op2 ( fpsrng R ) ).  Proof.  intros. intros a b c
p. rewrite ( rngcomm2 ( fpscommrng R ) ) in p. rewrite ( rngcomm2 (
fpscommrng R ) c ) in p.  apply ( fpsapartisbinopapartmultl _ a b c
). assumption.  Defined.

Definition acommrngfps ( R : acommrng ) : acommrng.  Proof.
intros. split with ( fpscommrng R ). split with ( fpsapart R
). split. split. apply ( fpsapartisbinopapartplusl R).  apply (
fpsapartisbinopapartplusr R ). split. apply (
fpsapartisbinopapartmultl R ). apply ( fpsapartisbinopapartmultr
R ).
Defined.

Definition isacommrngapartdec ( R : acommrng ) := isapartdec ( ( pr1
(
pr2 R ) ) ).

Lemma leadingcoefficientapartdec ( R : aintdom ) ( a : fpscommrng
R )
( is : isacommrngapartdec R ) ( p : a 0%nat # 0 ) : forall n : nat,
forall b : fpscommrng R, ( b n # 0 ) -> ( ( acommrngapartrel (
acommrngfps R ) ) ( a * b ) 0 ).  Proof.  intros R a is p n.
induction
n. intros b q. intros P s. apply s.  split with 0%nat. change ( ( a
*
b ) 0%nat ) with ( ( a 0%nat ) * ( b 0%nat ) ). apply
R. assumption. assumption.  intros b q. destruct ( is ( b 0%nat )
0 )
as [ left | right ].  intros P s. apply s. split with 0%nat.  change
(
( a * b ) 0%nat ) with ( ( a 0%nat ) * ( b 0%nat ) ).  apply
R. assumption. assumption.  assert ( ( acommrngapartrel
( acommrngfps
R ) ) ( a * ( fpsshift b ) ) 0 ) as j.  apply IHn. unfold
fpsshift. assumption. apply j. intro k. destruct k as [ k i ].
intros
P s. apply s. rewrite <- ( fpsshiftandmult a b right k ) in i. split
with ( S k ). assumption.  Defined.

Lemma apartdecintdom0 ( R : aintdom ) ( is : isacommrngapartdec
R ) :
forall n : nat, forall a b : fpscommrng R, ( a n # 0 ) -> (
acommrngapartrel ( acommrngfps R ) b 0 ) -> ( acommrngapartrel (
acommrngfps R ) ( a * b ) 0 ).  Proof.  intros R is n. induction
```

```
n. intros a b p q. apply q. intros k. destruct k as [ k k0 ]. apply
(
leadingcoefficientapartdec R a is p k ). assumption. intros a b p
q. destruct ( is ( a 0%nat ) 0 ) as [ left | right ].  apply q.
intros
k. destruct k as [ k k0 ]. apply ( leadingcoefficientapartdec R a is
left k ). assumption.  assert ( acommrngapartrel ( acommrngfps R )
( (
fpsshift a ) * b ) 0 ) as i.  apply IHn. unfold
fpsshift. assumption. assumption. apply i.  intros k. destruct k as
[
k k0 ].  intros P s. apply s. split with ( S k ). rewrite
rngcomm2. rewrite fpsshiftandmult.  rewrite
rngcomm2. assumption. assumption.  Defined.

Theorem apartdectoisaintdomfps ( R : aintdom ) ( is :
isacommrngapartdec R ) : aintdom.  Proof.  intros R. split with (
acommrngfps R ).  split. intros P s. apply s. split with 0%nat.
change
( ( @rngunel1 ( fpscommrng R ) ) 0%nat ) with ( @rngunel1 R ).
change
( @rngunel2 R # ( @rngunel1 R ) ). apply R. intros a b p q.  apply
p. intro n. destruct n as [ n n0 ]. apply ( apartdecintdom0 R is n)
. assumption. assumption.  Defined.

Close Scope rng_scope.

(** END OF FILE *)
```