lib/main/integer-power-series.ath

```
# Power-series over Z. A power-series is represented as a function p
# from N to Z that gives the coefficients of the series; i.e.,

#    sum   (p i) * x**i
#    i>=0

# except that instead of "(p i)" we write (Apply p i), so that we can
# work in first-order logic.  In defining arithmetic we only work with
# the coefficient functions, not with the monomial terms.

# There is no attempt to define arithmetic on this power series
# representation algorithmically; it is pure specification because of
# the universal quantification over all natural numbers.

# Note: For any power series p, p is a polynomial if it is identically
# zero or there is some maximal k such that (p k) =/= 0.  This is
# formally stated at the end of the file but is not further developed.

load "integer-plus"

module ZPS {

domain (Fun N Z)
declare zero: (Fun N Z)
declare apply: [(Fun N Z) N] -> Z
define at := apply

define +' := Z.+
define zero' := Z.zero

define [p q r i] := [?p:(Fun N Z) ?q:(Fun N Z) ?r:(Fun N Z) ?i:N]

assert* equality := (p = q <==> forall i . p at i =  q at i)

assert* zero-definition := (zero at i = zero')

declare +: [(Fun N Z) (Fun N Z)] -> (Fun N Z)

module Plus {
assert* definition := ((p + q) at i = (p at i) +' (q at i))

define right-identity := (forall p . p + zero = p)
define left-identity :=  (forall p . zero + p = p)

conclude right-identity
  pick-any p
    let {lemma :=
          pick-any i
            (!chain
             [((p + zero) at i)
            = ((p at i) +' (zero at i))   [definition]
            = ((p at i) +' zero')         [zero-definition]
            = (p at i)                    [Z.Plus.Right-Identity]])}
      (!chain-> [lemma ==> (p + zero = p) [equality]])

conclude left-identity
  pick-any p
    let {lemma :=
          pick-any i
            (!chain
             [((zero + p) at i)
            = ((zero at i) +' (p at i))    [definition]
            = (zero' +' (p at i))          [zero-definition]
            = (p at i)                     [Z.Plus.Left-Identity]])}
      (!chain-> [lemma ==> (zero + p = p)  [equality]])

define commutative := (forall p q . p + q = q + p)
```

```
define associative := (forall p q r . (p + q) + r = p + (q + r))

conclude commutative
  pick-any p q
    let {lemma :=
          pick-any i
            (!chain [((p + q) at i)
                   = ((p at i) +' (q at i))      [definition]
                   = ((q at i) +' (p at i))      [Z.Plus.commutative]
                   = ((q + p) at i)              [definition]])}
      (!chain-> [lemma ==> (p + q = q + p)       [equality]])

conclude associative
  pick-any p q r
    let {lemma :=
          pick-any i
            (!chain
             [(((p + q) + r) at i)
            = (((p + q) at i) +' (r at i))              [definition]
            = (((p at i) +' (q at i)) +' (r at i))      [definition]
            = ((p at i) +' ((q at i) +' (r at i)))      [Z.Plus.associative]
            = ((p at i) +' ((q + r) at i))              [definition]
            = ((p + (q + r)) at i)                      [definition]])}
    (!chain-> [lemma ==> ((p + q) + r = p + (q + r))    [equality]])
} # close module Plus

declare negate: [(Fun N Z)] -> (Fun N Z)

module Negate {

  assert* definition := ((negate p) at i = Z.negate (p at i))

} # close module Negate

declare -: [(Fun N Z) (Fun N Z)] -> (Fun N Z)

module Minus {
  assert* definition := (p - q = p + negate q)
} # close module Minus

extend-module Plus {
  define Plus-definition := definition
  open Negate
  open Minus

  define right-inverse := (forall p . p + (negate p) = zero)
  define left-inverse := (forall p . (negate p) + p = zero)

  conclude right-inverse
    pick-any p
      let {lemma :=
            pick-any i
              (!chain
                [((p + negate p) at i)
               = ((p at i) +' ((negate p) at i))    [Plus-definition]
               = ((p at i) +' Z.negate (p at i))    [Negate.definition]
               = zero'                              [Z.Plus.Right-Inverse]
               = (zero at i)                        [zero-definition]])}
        (!chain-> [lemma ==> (p + negate p = zero)  [equality]])
  } # close module Plus

declare poly: [(Fun N Z)] -> Boolean

define <= := N.<=

assert poly-definition :=
  (forall p .
    (poly p) <==>
    p = zero | (exists ?k . (apply p ?k) =/= Z.zero &
                            (forall i . ?k <= i ==>
```

```
                                             (apply p i) = Z.zero)))

} # ZPS
```