lib/algebra/Z-poly.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 for 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 + := Z.+

define zero := Z.zero


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

assert zero-definition := (forall i . (Apply zero i) = zero)

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

module Plus {

assert definition :=
(forall p q i . (Apply (p + q) i) = (Apply p i) + (Apply q 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
    (Apply (p + zero) i)
    = (Apply p i) + (Apply zero i) [definition]
    = (Apply p i) + zero [zero-definition]
    = (Apply p i) [Z.Plus.Right-Identity])}

(conclude-> lemma==> (p + zero = p) [equality])

conclude left-identity

pick-any p
let {lemma :=
  pick-any i
  (!chain
    (Apply (zero + p) i)
    = (Apply zero i) + (Apply p i) [definition]
    = zero + (Apply p i) [zero-definition]
    = (Apply p i) [Z.Plus.Left-Identity])}

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

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

declare Negate: [(Fun N Z)] -> (Fun N Z)
module Negate {
  assert definition := (forall p i . (Apply (Negate p) i) = (Z.negate (Apply p i)))
} # Negate

declare -: [(Fun N Z) (Fun N Z)] -> (Fun N Z)
module Minus {
  assert definition := (forall p q . p - q = p + Negate q)
} # 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
      [(Apply (p + (Negate p)) i)
        = ((Apply p i) +' (Apply (Negate p) i)) [Plus-definition]
        = (Negate (Apply p i)) [Negate.definition]
        = zero [Z.Plus.Right-Inverse]
        = (Apply zero i) [zero-definition]])
      (!chain-> [lemma ==> ((p + (Negate p)) = zero) [equality]])
    } # Plus

  conclude left-inverse
  pick-any p
  (!chain [( (Negate p) + p)
    = (p + (Negate p)) [commutative]
    = zero [right-inverse]]})

  conclude zero
  pick-any p
  (!chain [(p + (Negate p))
    = zero [right-inverse]])

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

define <= := N.<=

assert poly-definition :=
  (forall poly .
   [poly p] <=>
     p = zero | (exists k . (Apply p k) /= Z.zero &
        (forall i . k <= i ==> (Apply p i) = Z.zero)))

} # ZPS