

lib/algebra/Z-poly.ath

```

1 # Power-series over Z. A power-series is represented as a function p
2 # from N to Z that gives the coefficients of the series; i.e.,
3
4 #   sum   (p i) * x**i
5 #   i>=0
6
7 # except that instead of "(p i)" we write (Apply p i), so that we can
8 # work in first-order logic. In defining arithmetic we only work with
9 # the coefficient functions, not with the monomial terms.
10
11 # There is no attempt to define arithmetic on this power series
12 # representation algorithmically; it is pure specification because of
13 # the universal quantification over all natural numbers.
14
15 # Note: For any power series p, p is a polynomial if it is identically
16 # zero or there is some maximal k such that (p k) /= 0. This is
17 # formally stated at the end of the file but is not further developed.
18
19 load "integer-plus"
20
21 module ZPS {
22
23   domain (Fun N Z)
24   declare zero: (Fun N Z)
25   declare Apply: [(Fun N Z) N] -> Z
26
27   define +' := Z.+
28   define zero' := Z.zero
29
30   define [p q r i k] := [?p:(Fun N Z) ?q:(Fun N Z) ?r:(Fun N Z) ?i:N ?k:N]
31
32   assert equality :=
33     (forall p q . (p = q <==> (forall i . (Apply p i) = (Apply q i))))
34   assert zero-definition := (forall i . (Apply zero i) = zero')
35
36   declare +: [(Fun N Z) (Fun N Z)] -> (Fun N Z)
37
38   module Plus {
39
40     assert definition :=
41       (forall p q i . (Apply (p + q) i) = (Apply p i) +' (Apply q i))
42
43     define right-identity := (forall p . p + zero = p)
44     define left-identity := (forall p . zero + p = p)
45
46     conclude right-identity
47     pick-any p
48     let {lemma :=
49         pick-any i
50         (!chain
51           [(Apply (p + zero) i)
52            = ((Apply p i) +' (Apply zero i)) [definition]
53            = ((Apply p i) +' zero') [zero-definition]
54            = (Apply p i) [Z.Plus.Right-Identity]]})
55       (!chain-> [lemma ==> (p + zero = p) [equality]])
56
57     conclude left-identity
58     pick-any p
59     let {lemma :=
60         pick-any i
61         (!chain
62           [(Apply (zero + p) i)
63            = ((Apply zero i) +' (Apply p i)) [definition]
64            = (zero' +' (Apply p i)) [zero-definition]
65            = (Apply p i) [Z.Plus.Left-Identity]]})
66       (!chain-> [lemma ==> (zero + p = p) [equality]])
67

```

```

68 define commutative := (forall p q . p + q = q + p)
69 define associative := (forall p q r . (p + q) + r = p + (q + r))
70
71 conclude commutative
72 pick-any p:(Fun N Z) q:(Fun N Z)
73   let {lemma :=
74     pick-any i:N
75       (!chain [(Apply (p + q) i)
76               = ((Apply p i) +' (Apply q i)) [definition]
77               = ((Apply q i) +' (Apply p i)) [Z.Plus.commutative]
78               = (Apply (q + p) i) [definition]]})
79     (!chain-> [lemma ==> (p + q = q + p) [equality]])
80
81 conclude associative
82 pick-any p:(Fun N Z) q:(Fun N Z) r:(Fun N Z)
83   let {lemma :=
84     pick-any i:N
85       (!chain
86         [(Apply ((p + q) + r) i)
87          = ((Apply (p + q) i) +' (Apply r i)) [definition]
88          = (((Apply p i) +' (Apply q i)) +' (Apply r i)) [definition]
89          = ((Apply p i) +' ((Apply q i) +' (Apply r i))) [Z.Plus.associative]
90          = ((Apply p i) +' (Apply (q + r) i)) [definition]
91          = (Apply (p + (q + r)) i) [definition]]})
92     (!chain-> [lemma ==> ((p + q) + r = p + (q + r)) [equality]])
93 } # Plus
94
95 declare Negate: [(Fun N Z)] -> (Fun N Z)
96
97 module Negate {
98 assert definition :=
99   (forall p i . (Apply (Negate p) i) = (Z.negate (Apply p i)))
100 } # Negate
101
102 declare -: [(Fun N Z) (Fun N Z)] -> (Fun N Z)
103
104 module Minus {
105 assert definition := (forall p q . p - q = p + Negate q)
106 } # Minus
107
108 extend-module Plus {
109 define Plus-definition := definition
110 open Negate
111 open Minus
112
113 define right-inverse := (forall p . p + (Negate p) = zero)
114 define left-inverse := (forall p . (Negate p) + p = zero)
115
116 conclude right-inverse
117 pick-any p
118   let {lemma :=
119     pick-any i
120       (!chain
121         [(Apply (p + (Negate p)) i)
122          = ((Apply p i) +' (Apply (Negate p) i)) [Plus-definition]
123          = ((Apply p i) +' Z.negate (Apply p i)) [Negate.definition]
124          = zero' [Z.Plus.Right-Inverse]
125          = (Apply zero i) [zero-definition]]})
126     (!chain-> [lemma ==> ((p + (Negate p)) = zero) [equality]])
127
128 conclude left-inverse
129 pick-any p
130   (!chain [(Negate p) + p)
131           = (p + (Negate p)) [commutative]
132           = zero [right-inverse]])
133 } # Plus
134
135 } # Plus
136
137 # (define-symbol poly

```

```
138 # (forall p .
139 #   (poly p) <==>
140 #   p = zero | (exists k . (Apply p k) /= Z.zero &
141 #     (forall i . k <= i ==>
142 #       (Apply p i) = Z.zero))))
143 # The above yields an error: ill-formed symbol definition.
144
145 declare poly: [(Fun N Z)] -> Boolean
146
147 define <= := N.<=
148
149 assert poly-definition :=
150   (forall p .
151     (poly p) <==>
152     p = zero | (exists k . (Apply p k) /= Z.zero &
153       (forall i . k <= i ==>
154         (Apply p i) = Z.zero)))
155
156 } # ZPS
```