

lib/main/nat-plus.ath

```

1 #####
2 #
3 # Natural number datatype and Plus function
4 #
5
6 datatype N := zero | (S N)
7 set-precedence S 350
8 assert (datatype-axioms "N")
9
10 # Procedures for transforming an Athena int to a ground-term N and vice-versa
11
12 define (int->nat n) :=
13   (check ((integer-numeral? n)
14     (check ((n less? 1) zero)
15       (else (S (int->nat (n minus 1))))))
16     (else n))
17
18 define (nat->int n) :=
19   match n {
20     zero => 0
21     | (S k) => (plus (nat->int k) 1)
22     | _ => n
23   }
24
25 define (nat->int n) :=
26   match n {
27     zero => 0
28     | (S k) => (plus (nat->int k) 1)
29     | ((some-symbol f) (some-list args)) => try { (make-term f (map nat->int args)) | n }
30     | (list-of h rest) => (add (nat->int h) (map nat->int rest))
31     | _ => n
32   }
33
34 module N {
35
36   define [zero S] := [zero S]
37
38   declare one, two: N
39
40   define [k m n p x y z] := [?k:N ?m:N ?n:N ?p:N ?x:N ?y:N ?z:N]
41
42   assert one-definition := (one = (S zero))
43   assert two-definition := (two = (S one))
44
45   define S-not-zero      := (forall n . (S n) /= zero)
46   define one-not-zero   := (one /= zero)
47   define S-injective     := (forall m n . (S m) = (S n) <==> m = n)
48
49   # S-not-zero is essentially the same as one of the propositions
50   # returned by (datatype-axioms "N"):
51
52   conclude S-not-zero
53   pick-any n
54     (!sym (!instance (first (datatype-axioms "N")) n))
55
56   conclude S-not-zero
57   pick-any n
58     (!chain->
59       [true ==> (zero /= (S n)) [(datatype-axioms "N")]
60         ==> ((S n) /= zero) [sym]])
61
62   # Next we use S-not-zero to prove one-not-zero.
63
64   (!by-contradiction one-not-zero
65     assume (one = zero)
66     let {is := conclude ((S zero) = zero)
67       (!chain

```

```

68         [(S zero)
69         <-- one           [one-definition]
70         --> zero         [(one = zero)]];
71     is-not := (!chain-> [true ==> ((S zero) /= zero)
72                           [S-not-zero]])
73     (!absurd is is-not))
74
75 # One direction of S-injective is the second proposition
76 # returned by (datatype-axioms "N")
77
78 conclude S-injective
79 pick-any m:N n:N
80     let {right := (!chain [(S m) = (S n)] ==> (m = n)
81                           [(second (datatype-axioms "N"))])];
82         left := assume (m = n)
83                 (!chain [(S m) --> (S n) [(m = n)]])}
84     (!equiv right left)
85
86 # The following is equivalent to another of the propositions
87 # returned by (datatype-axioms "N"), but here we show
88 # it is a theorem.
89
90 define nonzero-S :=
91     (forall n . n /= zero ==> (exists m . n = (S m)))
92
93 define S-not-same := (forall n . (S n) /= n)
94
95 by-induction nonzero-S {
96     zero => assume (zero /= zero)
97             (!from-complements (exists ?m (zero = (S ?m)))
98                               (!reflex zero)
99                               (zero /= zero))
100 | (S m) => assume ((S m) /= zero)
101     let { _ := (!reflex (S m)) }
102         (!legen (exists ?m . (S m) = (S ?m)) m)
103 }
104
105 by-induction S-not-same {
106     zero => conclude ((S zero) /= zero)
107             (!instance S-not-zero zero)
108 | (S n) =>
109     let {induction-hypothesis := ((S n) /= n)}
110         (!chain-> [induction-hypothesis
111                   ==> ((S (S n)) /= (S n)) [S-injective]])
112 }
113
114 #####
115 #
116 # Addition operator, Plus
117 #
118
119 declare +: [N N] -> N [200]
120
121 module Plus {
122
123 # Axioms:
124 assert* Plus-def := [(n + zero = n)
125                     (n + S m = S (n + m))]
126
127 define [right-zero right-nonzero] := Plus-def
128 #assert right-zero := (forall n . n + zero = n)
129 #assert right-nonzero := (forall m n . n + (S m) = (S (n + m)))
130
131 # Theorems:
132
133 define left-zero := (forall n . zero + n = n)
134 define left-nonzero := (forall n m . (S m) + n = (S (m + n)))
135
136 by-induction left-zero {
137     zero => conclude (zero + zero = zero)

```

```

138      (!chain [(zero + zero) --> zero [right-zero]])
139 | (S n) => conclude (zero + (S n) = (S n))
140      let {induction-hypothesis := (zero + n = n)}
141      (!chain [(zero + (S n))
142              --> (S (zero + n)) [right-nonzero]
143              --> (S n)          [induction-hypothesis]])
144 }
145
146 by-induction left-nonzero {
147   zero =>
148     pick-any m
149     (!chain [((S m) + zero)
150             --> (S m)          [right-zero]
151             <-- (S (m + zero)) [right-zero]])
152 | (S n) =>
153   let {induction-hypothesis := (forall ?m . (S ?m) + n = (S (?m + n)))}
154   pick-any m
155   (!chain [((S m) + (S n))
156           --> (S ((S m) + n)) [right-nonzero]
157           --> (S (S (m + n))) [induction-hypothesis]
158           <-- (S (m + (S n))) [right-nonzero]])
159 }
160
161 # Adding one is the same as applying S
162
163 define right-one := (forall n . n + one = (S n))
164 define left-one  := (forall n . one + n = (S n))
165
166 conclude right-one
167   pick-any n
168   (!chain [(n + one)
169           --> (n + (S zero)) [one-definition]
170           --> (S (n + zero)) [right-nonzero]
171           --> (S n)         [right-zero]])
172
173 conclude left-one
174   pick-any n
175   (!chain [(one + n)
176           --> ((S zero) + n) [one-definition]
177           --> (S (zero + n)) [left-nonzero]
178           --> (S n)         [left-zero]])
179
180 # Associativity and commutativity:
181
182 define associative := (forall m p n . (m + p) + n = m + (p + n))
183 define commutative := (forall n m . m + n = n + m)
184
185 by-induction associative {
186   zero =>
187     pick-any p n
188     (!chain [((zero + p) + n)
189             --> (p + n)          [left-zero]
190             <-- (zero + (p + n)) [left-zero]])
191 | (S m) =>
192   let {induction-hypothesis :=
193         (forall ?p ?n . (m + ?p) + ?n = m + (?p + ?n))}
194   pick-any p n
195   (!chain
196     [(((S m) + p) + n)
197     --> ((S (m + p)) + n) [left-nonzero]
198     --> (S ((m + p) + n)) [left-nonzero]
199     --> (S (m + (p + n))) [induction-hypothesis]
200     <-- ((S m) + (p + n)) [left-nonzero]])
201 }
202
203 by-induction commutative {
204   zero =>
205     pick-any m
206     (!chain [(m + zero)
207             --> m          [right-zero]

```

```

208         <-- (zero + m)      [left-zero]])
209 | (S n) =>
210   pick-any m
211     let {induction-hypothesis := (forall ?m . ?m + n = n + ?m)}
212       (!chain [(m + (S n))
213               --> (S (m + n)) [right-nonzero]
214               --> (S (n + m)) [induction-hypothesis]
215               <-- ((S n) + m) [left-nonzero]])
216   }
217
218   # A cancellation property
219
220   define ==-cancellation :=
221     (forall k m n . m + k = n + k ==> m = n)
222
223   by-induction ==-cancellation {
224     zero =>
225       pick-any m n
226         assume assumption := (m + zero = n + zero)
227         (!chain [m <-- (m + zero) [right-zero]
228                --> (n + zero) [assumption]
229                --> n          [right-zero]])
230   | (S k) =>
231     let {induction-hypothesis :=
232           (forall ?m ?n . ?m + k = ?n + k ==> ?m = ?n)}
233       pick-any m n
234         assume assumption := (m + S k = n + S k)
235         (!chain->
236           [(S (m + k))
237            <-- (m + S k)      [right-nonzero]
238            --> (n + S k)      [assumption]
239            --> (S (n + k))     [right-nonzero]
240            ==> (m + k = n + k) [S-injective]
241            ==> (m = n)        [induction-hypothesis]])
242   }
243
244   # If a sum of two natural numbers is zero, each is zero. (Here we only show
245   # the first is zero.)
246
247   define squeeze-property := (forall m n . m + n = zero ==> m = zero)
248
249   conclude squeeze-property
250   pick-any m n
251     assume A := (m + n = zero)
252     (!by-contradiction (m = zero)
253       assume (m /= zero)
254         let {C := (!chain->
255                 [(m /= zero)
256                  ==> (exists ?k . m = (S ?k)) [nonzero-S]])}
257           pick-witness k for C witnessed
258             let {is := conclude ((S (k + n)) = zero)
259                 (!chain [(S (k + n))
260                          <-- ((S k) + n) [left-nonzero]
261                          <-- (m + n)      [witnessed]
262                          --> zero        [A]]);
263                 is-not := (!chain-> [true ==> ((S (k + n)) /= zero)
264                                     [S-not-zero]])}
265             (!absurd is is-not))
266 } # module N.Plus
267 } # module N

```