

lib/main/nat-times.ath

```

1 # Properties of natural number multiplication operator, Times.
2
3 load "nat-plus"
4
5 #
6 # Multiplication operator, Times
7 #
8
9 extend-module N {
10
11 declare *: [N N] -> N [300]
12
13 module Times {
14
15 open Plus
16
17 # Axioms
18
19 define [x y z] := [?x:N ?y:N ?z:N]
20
21 assert right-zero := (forall x . x * zero = zero)
22 assert right-nonzero := (forall x y . x * (S y) = x * y + x)
23
24 # Theorems:
25
26 define left-zero := (forall x . zero * x = zero)
27 define left-nonzero := (forall x y . (S y) * x = x + y * x)
28
29 by-induction left-zero {
30   zero =>
31     (!chain [(zero * zero) = zero [right-zero]])
32 | (S x) =>
33   let {induction-hypothesis := (zero * x = zero)}
34     (!chain [(zero * (S x))
35              = (zero * x + zero) [right-nonzero]
36              = (zero + zero) [induction-hypothesis]
37              = zero [Plus.right-zero]])
38 }
39
40 by-induction left-nonzero {
41   zero =>
42     pick-any y
43     (!combine-equations
44      (!chain [(S y) * zero = zero [right-zero]])
45      (!chain [(zero + y * zero)
46               = (zero + zero) [right-zero]
47               = zero [Plus.right-zero]]))
48 | (S x) =>
49   pick-any y
50   let {induction-hypothesis := (forall ?y . (S ?y) * x = x + ?y * x)}
51     (!combine-equations
52      (!chain
53       [((S y) * (S x))
54        --> ((S y) * x + (S y)) [right-nonzero]
55        --> ((x + y * x) + (S y)) [induction-hypothesis]
56        --> (S ((x + y * x) + y)) [Plus.right-nonzero]
57        --> (S (x + (y * x + y))) [Plus.associative]])
58      (!chain
59       [((S x) + y * (S x))
60        --> ((S x) + (y * x + y)) [right-nonzero]
61        --> (S (x + (y * x + y))) [Plus.left-nonzero]]))
62 }
63
64 define right-one := (forall x . x * one = x)
65 define left-one := (forall x . one * x = x)
66
67 conclude right-one
68 pick-any x

```

```

69     (!chain [(x * one)
70             --> (x * (S zero))    [one-definition]
71             --> (x * zero + x)   [right-nonzero]
72             --> (zero + x)       [right-zero]
73             --> x                 [Plus.left-zero]])
74
75 conclude left-one
76 pick-any x
77     (!chain [(one * x)
78             --> ((S zero) * x)    [one-definition]
79             --> (x + zero * x)    [left-nonzero]
80             --> (x + zero)        [left-zero]
81             --> x                 [Plus.right-zero]])
82
83 define right-distributive :=
84   (forall x y z . (x + y) * z = x * z + y * z)
85 define left-distributive :=
86   (forall z x y . z * (x + y) = z * x + z * y)
87
88 by-induction right-distributive {
89   zero =>
90     pick-any y z
91     (!combine-equations
92       (!chain [(zero + y) * z) = (y * z)    [Plus.left-zero]])
93       (!chain [(zero * z + y * z)
94               --> (zero + y * z)           [left-zero]
95               --> (y * z)                 [Plus.left-zero]]))
96 | (S x) =>
97   let {induction-hypothesis :=
98       (forall ?y ?z . (x + ?y) * ?z = x * ?z + ?y * ?z)}
99   pick-any y z
100  (!combine-equations
101    (!chain
102      [((S x) + y) * z)
103      --> ((S (x + y)) * z)           [Plus.left-nonzero]
104      --> (z + ((x + y) * z))        [left-nonzero]
105      --> (z + (x * z + y * z))      [induction-hypothesis]])
106    (!chain
107      [((S x) * z + y * z)
108      --> ((z + x * z) + y * z)      [left-nonzero]
109      --> (z + (x * z + y * z))      [Plus.associative]]))
110  }
111
112 # Associativity and commutativity:
113
114 define associative := (forall x y z . (x * y) * z = x * (y * z))
115 define commutative := (forall x y . x * y = y * x)
116
117 by-induction associative {
118   zero =>
119     pick-any y z
120     (!chain [(zero * y) * z)
121             --> (zero * z)           [left-zero]
122             --> zero                 [left-zero]
123             <-- (zero * (y * z))     [left-zero]])
124 | (S x) =>
125   let {induction-hypothesis :=
126       (forall ?y ?z . (x * ?y) * ?z = x * (?y * ?z))}
127   pick-any y z
128   (!chain
129     [((S x) * y) * z)
130     --> ((y + (x * y)) * z)         [left-nonzero]
131     --> (y * z + (x * y) * z)      [right-distributive]
132     --> (y * z + (x * (y * z)))    [induction-hypothesis]
133     <-- ((S x) * (y * z))          [left-nonzero]])
134 }
135
136 by-induction commutative {
137   zero =>
138     conclude (forall ?y . zero * ?y = ?y * zero)

```

```

139     pick-any y
140     (!chain [(zero * y)
141             --> zero           [left-zero]
142             <-- (y * zero)    [right-zero]])
143 | (S x) =>
144     let {induction-hypothesis := (forall ?y . (x * ?y = ?y * x))}
145     conclude (forall ?y . (S x) * ?y = ?y * (S x))
146     pick-any y
147     (!combine-equations
148     (!chain [(S x) * y)
149             --> (y + x * y) [left-nonzero]
150             --> (y + y * x) [induction-hypothesis]])
151     (!chain [(y * (S x))
152             --> (y * x + y) [right-nonzero]
153             --> (y + y * x) [Plus.commutative]]))
154 }
155
156 conclude left-distributive
157 pick-any z x y
158 (!chain [(z * (x + y))
159         --> ((x + y) * z) [commutative]
160         --> (x * z + y * z) [right-distributive]
161         --> (z * x + z * y) [commutative]])
162
163 define no-zero-divisors :=
164   (forall x y . x * y = zero ==> x = zero | y = zero)
165
166 conclude no-zero-divisors
167 pick-any x y
168 assume (x * y = zero)
169 (!two-cases
170  assume (x = zero)
171  (!left-either (x = zero) (y = zero))
172  assume A1 := (x != zero)
173  let {C1 := (!chain->
174            [A1 ==> (exists ?u . x = (S ?u))
175                 [nonzero-S]])}
176  pick-witness u for C1
177  let {C3 :=
178  (!by-contradiction (y = zero)
179  assume A2 := (y != zero)
180  let {C2 := (!chain->
181            [A2 ==> (exists ?v . y = (S ?v))
182                 [nonzero-S]])}
183  pick-witness v for C2
184  let {equal := (zero = (S ((S u) * v + u)))}
185  (!absurd
186   conclude equal
187   (!chain
188    [zero
189     <-- (x * y)           [(x * y = zero)]
190     --> ((S u) * (S v))   [(x = (S u)) (y = (S v))]
191     --> ((S u) * v + (S u)) [right-nonzero]
192     --> (S ((S u) * v + u)) [Plus.right-nonzero]])
193   conclude (~ equal)
194   (!chain->
195    [true ==> ((S ((S u) * v + u)) != zero)
196              [S-not-zero]
197              ==> (~ equal) [sym]]))
198  (!right-either (x = zero) C3))
199
200 # Alternative proof using datatype-cases:
201
202 datatype-cases no-zero-divisors {
203   zero =>
204     conclude (forall ?y . zero * ?y = zero ==> zero = zero | ?y = zero)
205     pick-any y
206     assume (zero * y = zero)
207     (!left-either (!reflex zero) (y = zero))
208 | (S x) =>

```

```

209 datatype-cases (forall ?y . (S x) * ?y = zero ==> (S x) = zero | ?y = zero)
210 {
211   zero =>
212     conclude ((S x) * zero = zero ==> (S x) = zero | zero = zero)
213     assume ((S x) * zero = zero)
214     (!right-either ((S x) = zero) (!reflex zero))
215   | (S y) =>
216     conclude ((S x) * (S y) = zero ==> (S x) = zero | (S y) = zero)
217     assume is-zero := ((S x) * (S y) = zero)
218     let {C :=
219       conclude ((S x) * (S y) = (S ((S x) * y + x)))
220       (!chain [((S x) * (S y))
221               --> ((S x) * y + (S x)) [right-nonzero]
222               --> (S ((S x) * y + x)) [Plus.right-nonzero]]);
223       is-not :=
224         (!chain->
225           [true ==> ((S ((S x) * y + x)) /= zero) [S-not-zero]
226                 ==> ((S x) * (S y) /= zero) [C]])}
227       (!from-complements ((S x) = zero | (S y) = zero) is-zero is-not)
228   }
229 }
230
231 define two-times := (forall x . two * x = x + x)
232
233 conclude two-times
234   pick-any x
235     (!chain [(two * x)
236             --> ((S one) * x) [two-definition]
237             --> (x + one * x) [left-nonzero]
238             --> (x + x) [left-one]])
239
240 } # Times
241
242 #####
243 # square function:
244
245 declare square: [N] -> N
246 module square {
247   define x := ?x:N
248
249   assert def := (forall x . square x = x * x)
250
251   define zero-property := (forall x . square x = zero ==> x = zero)
252
253   conclude zero-property
254     pick-any x
255       assume A := ((square x) = zero)
256       conclude (x = zero)
257         (!chain-> [(x * x)
258                 <-- (square x) [def]
259                 --> zero [A]
260                 ==> (x = zero | x = zero) [Times.no-zero-divisors]
261                 ==> (x = zero) [prop-taut]])
262
263 } # square
264
265 #####
266
267 } # N

```