

lib/main/nat-div.ath

```

1 load "nat-minus"
2 load "strong-induction"
3
4 #.....
5 extend-module N {
6
7 declare /, %: [N N] -> N [300 [int->nat int->nat]]
8
9 define [x y z] := [?x:N ?y:N ?z:N]
10
11 module Div {
12 assert basis := (forall x y . x < y ==> x / y = zero)
13 assert reduction :=
14   (forall x y . ~ x < y & zero < y ==> x / y = S ((x - y) / y))
15 } # close module Div
16
17 module Mod {
18 assert basis := (forall x y . x < y ==> x % y = x)
19 assert reduction :=
20   (forall x y . ~ x < y & zero < y ==> x % y = (x - y) % y)
21 } # close module Mod
22
23 extend-module Div {
24
25 define cancellation := (forall x y . zero < y ==> (x * y) / y = x)
26
27 by-induction cancellation {
28   zero => pick-any y
29     assume (zero < y)
30       (!chain [((zero * y) / y)
31               = (zero / y)      [Times.left-zero]
32               = zero           [basis (zero < y)]]
33 | (S x) =>
34   pick-any y
35     let {ind-hyp := (forall ?y . zero < ?y ==> (x * ?y) / ?y = x)}
36       assume (zero < y)
37         let {B := conclude (~ x * y + y < y)
38             (!chain-> [(~ x * y + y < y)
39                       <== (y <= x * y + y) [Less=.trichotomy3]
40                       <== (y <= y + x * y) [Plus.commutative]
41                       <== (y <= y)       [Less=.Plus-k1]
42                       <== true           [Less=.reflexive]]]}
43           conclude (((S x) * y) / y) = (S x))
44             (!chain [(((S x) * y) / y)
45                     = ((y + x * y) / y)      [Times.left-nonzero]
46                     = ((x * y + y) / y)      [Plus.commutative]
47                     = (S ((x * y + y) - y) / y) [reduction B]
48                     = (S ((x * y) / y))      [Plus.commutative]
49                     = (S x)                  [Minus.cancellation]
50                     = (S x)                  [ind-hyp]]]
51 }
52 } # close module Div
53
54 #.....
55
56 define division-algorithm :=
57   (forall x y . zero < y ==> (x / y) * y + x % y = x & x % y < y)
58
59 conclude goal := division-algorithm
60 (!strong-induction.principle goal
61   method (x)
62     assume IND-HYP := (strong-induction.hypothesis goal x)
63     conclude (strong-induction.conclusion goal x)
64     pick-any y
65       assume (zero < y)
66         conclude ((x / y) * y + x % y = x & x % y < y)
67         (!two-cases
68           assume (x < y)

```

```

69     let {C1 :=
70         (!chain->
71             [(x < y) ==> (x / y = zero) [Div.basis]]);
72     C2 :=
73         (!chain->
74             [(x < y) ==> (x % y = x) [Mod.basis]]);
75     C3 :=
76         (!chain
77             [(x / y) * y + (x % y)
78              = (zero * y + x) [C1 C2]
79              = x [Times.left-zero Plus.left-zero]]);
80     C4 := (!chain->
81             [(x < y) ==> (x % y < y) [C2]]})
82     (!both C3 C4)
83     assume (~ x < y)
84     let {C1 :=
85         (!chain->
86             [(~ x < y & zero < y)
87              ==> (x / y = (S ((x - y) / y)))
88                  [Div.reduction]]);
89     C2 :=
90         (!chain->
91             [(~ x < y & zero < y)
92              ==> (x % y = (x - y) % y)
93                  [Mod.reduction]]);
94     C3 := (!chain->
95             [(~ x < y) ==> (y <= x)
96                  [Less=.trichotomy2]]);
97     C4 :=
98         (!chain->
99             [(zero < y & y <= x)
100              ==> (x - y < x) [Minus.<-left]
101              ==> (forall ?v . zero < ?v ==>
102                  ((x - y) / ?v) * ?v + (x - y) % ?v
103                   = x - y &
104                   (x - y) % ?v < ?v)) [IND-HYP]]);
105     C5 :=
106         (!chain->
107             [(zero < y)
108              ==> (((x - y) / y) * y + (x - y) % y = x - y
109                  & (x - y) % y < y) [C4]]);
110     C5a := (!left-and C5);
111     C5b := (!right-and C5);
112     C6 :=
113         (!chain
114             [(x / y) * y + x % y
115              = ((S ((x - y) / y)) * y + (x - y) % y)
116                  [C1 C2]
117              = ((y + ((x - y) / y) * y) + (x - y) % y)
118                  [Times.left-nonzero]
119              = (y + (((x - y) / y) * y + (x - y) % y))
120                  [Plus.associative]
121              = (y + (x - y)) [C5a]
122              = ((x - y) + y) [Plus.commutative]
123              = x [C3 Minus.Plus-Cancel]})
124     (!chain->
125         [C5b ==> (x % y < y) [C2]
126          ==> (C6 & (x % y < y)) [augment]]))
127
128     define division-algorithm-corollary1 :=
129         (forall x y . zero < y ==> (x / y) * y + x % y = x)
130     define division-algorithm-corollary2 :=
131         (forall x y . zero < y ==> x % y < y)
132
133     conclude corollary := division-algorithm-corollary1
134     let {theorem := division-algorithm
135         (!mp (!taut (theorem ==> corollary))
136             theorem)
137
138     conclude corollary := division-algorithm-corollary2

```

```

139 let {theorem := division-algorithm}
140     (!mp (!taut (theorem ==> corollary))
141         theorem)
142
143 #.....
144
145 declare divides: [N N] -> Boolean [300 [int->nat int->nat]]
146
147 module divides {
148
149 assert left-positive :=
150     (forall x y . zero < y ==> y divides x <==> x % y = zero)
151
152 assert left-zero :=
153     (forall x y . y = zero ==> y divides x <==> x = zero)
154
155 define characterization :=
156     (forall x y . y divides x <==> exists z . y * z = x)
157
158 conclude characterization
159
160 pick-any x y
161     (!two-cases
162         assume (zero < y)
163             (!equiv
164                 assume A := (y divides x)
165                 let {B := (!chain-> [A ==> (x % y = zero) [left-positive]])}
166                     (!chain-> [(zero < y)
167                         ==> ((x / y) * y + x % y = x)
168                             [division-algorithm-corollary1]
169                         ==> ((x / y) * y + zero = x) [B]
170                         ==> ((x / y) * y = x) [Plus.right-zero]
171                         ==> (y * (x / y) = x) [Times.commutative]
172                         ==> (exists ?z . y * ?z = x) [existence]])]
173                 assume A := (exists ?z . y * ?z = x)
174                 pick-witness z for A A-w
175                 (!by-contradiction (y divides x)
176                     assume B := (~ y divides x)
177                     let {C := (!chain-> [(zero < y)
178                         ==> (y divides x <==> x % y = zero)
179                             [left-positive]])}
180                         (!absurd
181                             (!chain->
182                                 [B ==> (x % y != zero) [C]
183                                 ==> (zero < x % y) [Less.zero<]
184                                 ==> (zero + (x / y) * y < x % y + (x / y) * y)
185                                     [Less.Plus-k]
186                                 ==> ((x / y) * y < (x / y) * y + x % y)
187                                     [Plus.left-zero Plus.commutative]
188                                 ==> ((x / y) * y < x)
189                                     [division-algorithm-corollary1]
190                                 ==> (y * (x / y) < y * z) [A-w Times.commutative]
191                                 ==> (x / y < z) [Times.<-cancellation]
192                                 ==> ((y * z) / y < z) [A-w]
193                                 ==> (z < z) [Times.commutative Div.cancellation]))]
194                             (!chain-> [true ==> (~ z < z) [Less.irreflexive]])))
195                     assume (~ zero < y)
196                     let {C := (!chain-> [(~ zero < y) ==> (y = zero) [Less.=zero]])}
197                         (!equiv
198                             assume A := (y divides x)
199                             (!chain-> [A ==> (x = zero) [left-zero]
200                                 ==> (zero = x) [sym]
201                                 ==> (y * zero = x) [Times.right-zero]
202                                 ==> (exists ?z . y * ?z = x) [existence]])]
203                             assume A := (exists ?z . y * ?z = x)
204                             let {B := (!chain->
205                                 [C ==> (y divides x <==> x = zero) [left-zero]])}
206                                 pick-witness z for A A-w
207                                 (!chain-> [x = (y * z) [A-w]
208                                     = (zero * z) [C]
209                                     = zero [Times.left-zero]
210                                     ==> (y divides x) [B]])
211                                 )))

```

```

209
210 define elim :=
211   method (x y)
212     let {v := (fresh-var (sort-of x))}
213       (!chain->
214         [(divides x y) ==> (exists v . x * v = y) [characterization]])
215
216 define reflexive := (forall x . x divides x)
217 define right-zero := (forall x . x divides zero)
218 define left-zero := (forall x . zero divides x <==> x = zero)
219
220 conclude reflexive
221   pick-any x
222     (!chain->
223       [true ==> (x * one = x) [Times.right-one]
224         ==> (exists ?y . x * ?y = x) [existence]
225         ==> (x divides x) [characterization]])
226
227 conclude right-zero
228   pick-any x
229     (!chain->
230       [true ==> (x * zero = zero) [Times.right-zero]
231         ==> (exists ?y . x * ?y = zero) [existence]
232         ==> (x divides zero) [characterization]])
233
234 conclude left-zero
235   pick-any x
236     let {right := conclude (zero divides x ==> x = zero)
237           assume (zero divides x)
238           let {C1 := (!elim zero x)}
239             pick-witness y for C1 C1-w
240               (!chain
241                 [x = (zero * y) [C1-w]
242                   = zero [Times.left-zero]]);
243           left := conclude (x = zero ==> zero divides x)
244                 assume (x = zero)
245                 (!chain->
246                   [true ==> (zero * zero = zero) [Times.left-zero]
247                     ==> (exists ?y . zero * ?y = zero) [existence]
248                     ==> (zero divides zero) [characterization]
249                     ==> (zero divides x) [(x = zero)]])}
250     (!equiv right left)
251
252 #.....
253 define sum-lemmal :=
254   (forall x y z . x divides y & x divides z ==> x divides (y + z))
255 define sum-lemma2 :=
256   (forall x y z . x divides y & x divides (y + z) ==> x divides z)
257 define sum :=
258   (forall x y z . x divides y & x divides z
259     <==> x divides y & x divides (y + z))
260
261 conclude sum-lemmal
262   pick-any x y z
263     assume (x divides y & x divides z)
264     pick-witness u for (!elim x y)
265     pick-witness v for (!elim x z)
266     let {witnessed1 := (x * u = y);
267           witnessed2 := (x * v = z)}
268     conclude goal := (x divides (y + z))
269     (!chain->
270       [(x * (u + v))
271         = (x * u + x * v) [Times.left-distributive]
272         = (y + z) [witnessed1 witnessed2]
273         ==> (exists ?w . x * ?w = y + z) [existence]
274         ==> goal [characterization]])
275
276
277 conclude sum-lemma2
278   pick-any x y z

```

```

279   assume (x divides y & x divides (y + z))
280   pick-witness u for (!elim x y)
281   pick-witness v for (!elim x (y + z))
282   conclude goal := (x divides z)
283   let {w1 := (x * u = y);
284       w2 := (x * v = y + z)}
285   (!chain->
286     [(x * (v - u))
287      = (x * v - x * u) [Minus.Times-Distributivity]
288      = ((y + z) - y) [w1 w2]
289      = z [Minus.cancellation]
290      ==> (exists ?w . x * ?w = z) [existence]
291      ==> goal [characterization]])
292
293 conclude sum
294 pick-any x y z
295   let {right := assume A := (x divides y & x divides z)
296       (!chain->
297         [A ==> (x divides (y + z)) [sum-lemmal]
298          ==> (x divides y & x divides (y + z))
299           [augment]]);
300       left := assume A := (x divides y & x divides (y + z))
301       (!chain->
302         [A ==> (x divides z) [sum-lemma2]
303          ==> (x divides y & x divides z) [augment]])}
304   (!equiv right left)
305
306 #.....
307 define product-lemma :=
308   (forall x y z . x divides y | x divides z ==> x divides y * z)
309 define product-left-lemma :=
310   (forall x y z . x divides y ==> x divides y * z)
311
312 conclude product-left-lemma
313 pick-any x y z
314   assume A := (x divides y)
315   pick-witness u for (!elim x y) witnessed
316   (!chain->
317     [(y * z) = ((x * u) * z) [witnessed]
318      = (x * (u * z)) [Times.associative]
319      ==> (x * (u * z) = y * z) [sym]
320      ==> (exists ?v . x * ?v = y * z) [existence]
321      ==> (x divides y * z) [characterization]])
322
323 conclude product-lemma
324 pick-any x y z
325   assume A := (x divides y | x divides z)
326   conclude goal := (x divides y * z)
327   (!cases A
328     assume A1 := (x divides y)
329     (!chain-> [A1 ==> goal [product-left-lemma]])
330     assume A2 := (x divides z)
331     (!chain->
332       [A2 ==> (x divides z * y) [product-left-lemma]
333        ==> goal [Times.commutative]]))
334
335 #.....
336 define first-lemma :=
337   (forall x y z .
338     zero < y & z divides y & z divides x % y ==> z divides x)
339
340 conclude first-lemma
341 pick-any x y z
342   assume A := (zero < y & z divides y & z divides x % y)
343   conclude goal := (z divides x)
344   pick-witness u for (!elim z y) witnessed1
345   pick-witness v for (!elim z (x % y)) witnessed2
346   (!chain->
347     [x = ((x / y) * y + x % y)
348      [(zero < y) division-algorithm-corollary1]

```

```

349         = ((x / y) * (z * u) + z * v)
350           [witnessed1 witnessed2]
351         = (((x / y) * u) * z + v * z) [Times.commutative
352                                         Times.associative]
353         = (((x / y) * u + v) * z) [Times.right-distributive]
354         = (z * ((x / y) * u + v)) [Times.commutative]
355         ==> (z * ((x / y) * u + v) = x) [sym]
356         ==> (exists ?w . z * ?w = x)      [existence]
357         ==> goal                          [characterization]]
358
359 #.....
360 define antisymmetric :=
361   (forall x y . x divides y & y divides x ==> x = y)
362
363 conclude antisymmetric
364   pick-any x y
365     assume (x divides y & y divides x)
366       pick-witness u for (!elim x y)
367       pick-witness v for (!elim y x)
368       let {witnessed1 := (x * u = y);
369           witnessed2 := (y * v = x)}
370         (!two-cases
371           assume A1 := (x = zero)
372             (!chain->
373               [witnessed1 ==> (zero * u = y) [A1]
374                 ==> (zero = y) [Times.left-zero]
375                 ==> (x = y) [A1]])
376           assume A2 := (x /= zero)
377             let {C1 := (!chain-> [A2 ==> (zero < x) [Less.zero<]]);
378                 C2 :=
379                   (!chain->
380                     [x = (y * v) [witnessed2]
381                       = ((x * u) * v) [witnessed1]
382                       = (x * (u * v)) [Times.associative]
383                       ==> (x * (u * v) = x) [sym]
384                       ==> (u * v = one) [C1 Times.identity-lemma1]
385                       ==> (u = one) [Times.identity-lemma2]])
386                   (!chain
387                     [x = (x * one) [Times.right-one]
388                       = (x * u) [C2]
389                       = y [witnessed1]])
390
391 #.....
392 define transitive :=
393   (forall x y z . x divides y & y divides z ==> x divides z)
394
395 conclude transitive
396   pick-any x y z
397     assume (x divides y & y divides z)
398     pick-witness u for (!elim x y) witnessed1
399     pick-witness v for (!elim y z) witnessed2
400     (!chain->
401       [(x * (u * v))
402         = ((x * u) * v) [Times.associative]
403         = (y * v) [witnessed1]
404         = z [witnessed2]
405         ==> (exists ?w . x * ?w = z) [existence]
406         ==> (x divides z) [characterization]])
407
408 #.....
409 define Minus-lemma :=
410   (forall x y z . x divides y & x divides z ==> x divides (y - z))
411
412 conclude Minus-lemma
413   pick-any x y z
414     assume (x divides y & x divides z)
415     pick-witness u for (!elim x y) witnessed1
416     pick-witness v for (!elim x z) witnessed2
417     (!chain->
418       [(y - z)

```

```

419     = (x * u - x * v) [witnessed1 witnessed2]
420     = (x * (u - v)) [Minus.Times-Distributivity]
421     ==> (x * (u - v) = y - z) [sym]
422     ==> (exists ?w . x * ?w = y - z) [existence]
423     ==> (x divides (y - z)) [characterization]]
424
425 define Mod-lemma :=
426   (forall x y z . x divides y & x divides z & zero < z
427     ==> x divides y % z)
428
429 conclude Mod-lemma
430 pick-any x y z
431 assume (x divides y & x divides z & zero < z)
432 let {C1 := (!chain->
433   [(zero < z)
434    ==> ((y / z) * z + y % z = y)
435     [division-algorithm-corollary1]]);
436   C2 :=
437     conclude (x divides (y / z) * z)
438     (!chain->
439       [(x divides z)
440        ==> (x divides z * (y / z)) [product-left-lemma]
441         ==> (x divides (y / z) * z) [Times.commutative]])}
442   (!chain->
443     [(x divides y)
444      ==> (x divides ((y / z) * z + y % z)) [C1]
445       ==> (C2 & (x divides ((y / z) * z + y % z))) [augment]
446        ==> (x divides y % z) [sum-lemma2]])
447
448 } # close module divides
449 } # close module N

```