

lib/main/integer-times.ath

```

1 #####
2 #
3 # Integer multiplication operator, Z.*
4 #
5
6 load "nat-times"
7 load "integer-plus"
8
9 extend-module Z {
10 declare *: [Z Z] -> Z
11 set-precedence * (get-precedence N.*)
12 module Times {
13 overload * N.*
14 define [x y] := [?x:N ?y:N]
15 assert axioms :=
16   (fun [(pos x * pos y) = (pos (x * y))
17         (pos x * neg y) = (neg (x * y))
18         (neg x * pos y) = (neg (x * y))
19         (neg x * neg y) = (pos (x * y))])
20 define [pos-pos pos-neg neg-pos neg-neg] := axioms
21
22 define associative := (forall a b c . (a * b) * c = a * (b * c))
23 define commutative := (forall a b . a * b = b * a)
24
25 # Unlike the case with addition, the signed integer representation is better
26 # than the Z.NN representation for proving these properties. First, consider
27 # commutativity - since it involves only two variables, there are only four
28 # cases to consider.
29
30 datatype-cases commutative {
31   (pos x) =>
32     datatype-cases (forall ?b . pos x * ?b = ?b * pos x) {
33       (pos y) =>
34         (!chain [(pos x * pos y)
35                 --> (pos (x * y))           [pos-pos]
36                 --> (pos (y * x))           [N.Times.commutative]
37                 <-- (pos y * pos x)         [pos-pos]])
38       | (neg y) =>
39         (!chain [(pos x * neg y)
40                 --> (neg (x * y))           [pos-neg]
41                 --> (neg (y * x))           [N.Times.commutative]
42                 <-- (neg y * pos x)         [neg-pos]])
43     }
44   | (neg x) =>
45     datatype-cases (forall ?b . neg x * ?b = ?b * neg x) {
46       (pos y) =>
47         (!chain [(neg x * pos y)
48                 --> (neg (x * y))           [neg-pos]
49                 --> (neg (y * x))           [N.Times.commutative]
50                 <-- (pos y * neg x)         [pos-neg]])
51       | (neg y) =>
52         (!chain [(neg x * neg y)
53                 --> (pos (x * y))           [neg-neg]
54                 --> (pos (y * x))           [N.Times.commutative]
55                 <-- (neg y * neg x)         [neg-neg]])
56     }
57 }
58
59 # Since there are three variables, associativity requires eight cases, but each
60 # is straightforward.
61
62 let {assoc := N.Times.associative}
63 datatype-cases associative {
64   (pos x) =>
65     datatype-cases
66       (forall ?b ?c . ((pos x) * ?b) * ?c = (pos x) * (?b Z.* ?c)) {
67     (pos y) =>

```

```

68 datatype-cases
69 (forall ?c . ((pos x) * (pos y)) * ?c = (pos x) * ((pos y) * ?c)) {
70 (pos z) =>
71 (!chain [((pos x) * (pos y)) * (pos z))
72 --> ((pos (x * y)) * (pos z)) [pos-pos]
73 --> (pos ((x * y) * z)) [pos-pos]
74 --> (pos (x * (y * z))) [assoc]
75 <-- ((pos x) * (pos (y * z))) [pos-pos]
76 <-- ((pos x) * ((pos y) * (pos z))) [pos-pos]]
77 | (neg z) =>
78 (!chain [((pos x) * (pos y)) * (neg z))
79 --> ((pos (x * y)) * (neg z)) [pos-pos]
80 --> (neg ((x * y) * z)) [pos-neg]
81 --> (neg (x * (y * z))) [assoc]
82 <-- ((pos x) * (neg (y * z))) [pos-neg]
83 <-- ((pos x) * ((pos y) * (neg z))) [pos-neg]]
84 }
85 | (neg y) =>
86 datatype-cases
87 (forall ?c . ((pos x) * (neg y)) * ?c = (pos x) * ((neg y) * ?c)) {
88 (pos z) =>
89 (!chain [((pos x) * (neg y)) * (pos z))
90 --> ((neg (x * y)) * (pos z)) [pos-neg]
91 --> (neg ((x * y) * z)) [neg-pos]
92 --> (neg (x * (y * z))) [assoc]
93 <-- ((pos x) * (neg (y * z))) [pos-neg]
94 <-- ((pos x) * ((neg y) * (pos z))) [neg-pos]]
95 | (neg z) =>
96 (!chain [((pos x) * (neg y)) * (neg z))
97 --> ((neg (x * y)) * (neg z)) [pos-neg]
98 --> (pos ((x * y) * z)) [neg-neg]
99 --> (pos (x * (y * z))) [assoc]
100 <-- ((pos x) * (pos (y * z))) [pos-pos]
101 <-- ((pos x) * ((neg y) * (neg z))) [neg-neg]]
102 }
103 }
104 | (neg x) =>
105 datatype-cases
106 (forall ?b ?c . ((neg x) * ?b) * ?c = (neg x) * (?b Z.* ?c)) {
107 (pos y) =>
108 datatype-cases
109 (forall ?c . ((neg x) * (pos y)) * ?c = (neg x) * ((pos y) * ?c)) {
110 (pos z) =>
111 (!chain [((neg x) * (pos y)) * (pos z))
112 --> ((neg (x * y)) * (pos z)) [neg-pos]
113 --> (neg ((x * y) * z)) [neg-pos]
114 --> (neg (x * (y * z))) [assoc]
115 <-- ((neg x) * (pos (y * z))) [neg-pos]
116 <-- ((neg x) * ((pos y) * (pos z))) [pos-pos]]
117 | (neg z) =>
118 (!chain [((neg x) * (pos y)) * (neg z))
119 --> ((neg (x * y)) * (neg z)) [neg-pos]
120 --> (pos ((x * y) * z)) [neg-neg]
121 --> (pos (x * (y * z))) [assoc]
122 <-- ((neg x) * (neg (y * z))) [neg-neg]
123 <-- ((neg x) * ((pos y) * (neg z))) [pos-neg]]
124 }
125 | (neg y) =>
126 datatype-cases
127 (forall ?c . ((neg x) * (neg y)) * ?c = (neg x) * ((neg y) * ?c)) {
128 (pos z) =>
129 (!chain [((neg x) * (neg y)) * (pos z))
130 --> ((pos (x * y)) * (pos z)) [neg-neg]
131 --> (pos ((x * y) * z)) [pos-pos]
132 --> (pos (x * (y * z))) [assoc]
133 <-- ((neg x) * (neg (y * z))) [neg-neg]
134 <-- ((neg x) * ((neg y) * (pos z))) [neg-pos]]
135 | (neg z) =>
136 (!chain [((neg x) * (neg y)) * (neg z))
137 --> ((pos (x * y)) * (neg z)) [neg-neg]

```

```

138         --> (neg ((x * y) * z))           [pos-neg]
139         --> (neg (x * (y * z)))           [assoc]
140         <-- ((neg x) * (pos (y * z)))      [neg-pos]
141         <-- ((neg x) * ((neg y) * (neg z))) [neg-neg]]
142     }
143 }
144 }
145
146 #
147
148 define Right-Distributive :=
149     (forall a b c . (a + b) * c = a * c + b * c)
150
151 define Left-Distributive :=
152     (forall a b c . c * (a + b) = c * a + c * b)
153
154 } # Times
155
156 #.....
157 # To prove Right Distributive, it seems best to use the Z->NN and NN->Z mappings.
158
159 extend-module NN {
160 overload * N.*
161 define-sort NN := Z.NN
162 declare *': [NN NN] -> NN
163 set-precedence *' (get-precedence *)
164 module Times {
165 define [a1 a2 b1 b2] := [?a1:N ?a2:N ?b1:N ?b2:N]
166 assert definition :=
167     (forall a1 a2 b1 b2 .
168         (nn a1 a2) *' (nn b1 b2) =
169         (nn (a1 * b1 + a2 * b2)
170         (a1 * b2 + a2 * b1)))
171 } # Times
172 } # NN
173
174 extend-module Z-NN {
175 overload * N.*
176 define *' := NN.*'
177
178 define multiplicative-homomorphism :=
179     (forall a b . (Z->NN (a * b)) = (Z->NN a) *' (Z->NN b))
180
181 let {f:(OP 1) := Z->NN; definition := NN.Times.definition}
182 datatype-cases multiplicative-homomorphism {
183     (pos x) =>
184         datatype-cases
185         (forall ?b . (f ((pos x) * ?b)) = (f (pos x)) *' (f ?b)) {
186             (pos y) =>
187                 (!combine-equations
188                 (!chain [(f ((pos x) * (pos y)))
189                     --> (f (pos (x * y)))           [Times.pos-pos]
190                     --> (nn (x * y) Top.zero)       [to-pos]])
191                 (!chain [(f (pos x)) *' (f (pos y))]
192                     --> ((nn x Top.zero) *' (nn y Top.zero)) [to-pos]
193                     --> (nn (x * y + Top.zero * Top.zero)
194                         (x * Top.zero + Top.zero * y))
195                         [definition]
196                     --> (nn (x * y + Top.zero) (Top.zero + Top.zero))
197                         [N.Times.right-zero
198                         N.Times.left-zero]
199                     --> (nn (x * y) Top.zero)         [N.Plus.right-zero]]))
186             (neg y) =>
187                 (!combine-equations
188                 (!chain [(f ((pos x) * (neg y)))
189                     --> (f (neg (x * y)))           [Times.pos-neg]
190                     --> (nn Top.zero (x * y))       [to-neg]])
191                 (!chain [(f (pos x)) *' (f (neg y))]
192                     --> ((nn x Top.zero) *' (nn Top.zero y))
193                         [to-pos to-neg]

```

```

208         --> (nn (x * Top.zero + Top.zero * y)
209             (x * y + Top.zero * Top.zero)) [definition]
210         --> (nn (Top.zero + Top.zero) (x * y + Top.zero))
211             [N.Times.right-zero
212              N.Times.left-zero]
213         --> (nn Top.zero x * y) [N.Plus.right-zero]])
214     }
215 | (neg x) =>
216   datatype-cases
217   (forall ?b . (f ((neg x) * ?b)) = (f (neg x)) *' (f ?b)) {
218     (pos y) =>
219       (!combine-equations
220         (!chain [(f ((neg x) * (pos y)))
221                 --> (f (neg (x * y))) [Times.neg-pos]
222                 --> (nn Top.zero (x * y)) [to-neg]])]
223         (!chain [(f (neg x)) *' (f (pos y))]
224                 --> ((nn Top.zero x) *' (nn y Top.zero)) [to-neg to-pos]
225                 --> (nn (Top.zero * y + x * Top.zero)
226                     (Top.zero * Top.zero + x * y))
227                     [definition]
228                 --> (nn (Top.zero + Top.zero) (Top.zero + x * y))
229                     [N.Times.right-zero
230                      N.Times.left-zero]
231                 --> (nn Top.zero (x * y)) [N.Plus.left-zero]]))
232     | (neg y) =>
233       (!combine-equations
234         (!chain [(f ((neg x) * (neg y)))
235                 --> (f (pos (x * y))) [Times.neg-neg]
236                 --> (nn (x * y) Top.zero) [to-pos]])]
237         (!chain [(f (neg x)) *' (f (neg y))]
238                 --> ((nn Top.zero x) *' (nn Top.zero y)) [to-neg]
239                 --> (nn (Top.zero * Top.zero + x * y)
240                     (Top.zero * y + x * Top.zero))
241                     [definition]
242                 --> (nn (Top.zero + x * y) (Top.zero + Top.zero))
243                     [N.Times.right-zero
244                      N.Times.left-zero]
245                 --> (nn (x * y) Top.zero) [N.Plus.left-zero]]))
246   }
247 }
248 } # Z-NN
249
250 #-----
251 extend-module NN {
252 extend-module Times {
253 define Right-Distributive :=
254   (forall a b c . (a +' b) *' c = a *' c +' b *' c)
255
256 datatype-cases Right-Distributive {
257   (Z.nn a1 a2) =>
258     datatype-cases
259     (forall ?b ?c . ((nn a1 a2) +' ?b) *' ?c =
260                     (nn a1 a2) *' ?c +' ?b *' ?c) {
261       (Z.nn b1 b2) =>
262         datatype-cases
263         (forall ?c .
264           ((nn a1 a2) +' (nn b1 b2)) *' ?c =
265            (nn a1 a2) *' ?c +' (nn b1 b2) *' ?c)
266         {
267           (Z.nn c1 c2) =>
268             (!combine-equations
269               (!chain
270                 [((nn a1 a2) +' (nn b1 b2)) *' (nn c1 c2))
271                  = ((nn (a1 + b1) (a2 + b2)) *' (nn c1 c2))
272                  [Plus.definition]
273                  = (nn ((a1 + b1) * c1 + (a2 + b2) * c2)
274                      ((a1 + b1) * c2 + (a2 + b2) * c1))
275                  [definition]
276                  = (nn ((a1 * c1 + b1 * c1) + (a2 * c2 + b2 * c2))
277                      ((a1 * c2 + b1 * c2) + (a2 * c1 + b2 * c1)))

```

```

278                                     [N.Times.right-distributive]])
279 (!chain [(nn a1 a2) *' (nn c1 c2)
280           +' (nn b1 b2) *' (nn c1 c2)])
281 = ((nn (a1 * c1 + a2 * c2) (a1 * c2 + a2 * c1))
282    +' (nn (b1 * c1 + b2 * c2)
283           (b1 * c2 + b2 * c1)))
284                                     [definition]
285 = (nn ((a1 * c1 + a2 * c2) + (b1 * c1 + b2 * c2))
286      ((a1 * c2 + a2 * c1) + (b1 * c2 + b2 * c1)))
287                                     [Plus.definition]
288 = (nn ((a1 * c1 + b1 * c1) + (a2 * c2 + b2 * c2))
289      ((a1 * c2 + b1 * c2) + (a2 * c1 + b2 * c1)))
290                                     [N.Plus.commutative
291                                     N.Plus.associative]])
292   }
293 }
294 }
295 } # Times
296 } # NN
297
298 extend-module Times {
299 define +' := NN.+
300 define *' := NN.*
301
302 conclude Right-Distributive
303 pick-any a:Z b:Z c:Z
304 let {f:(OP 1) := Z->NN; g:(OP 1) := NN->Z;
305      f-application :=
306        conclude ((f ((a + b) * c)) = (f (a * c + b * c)))
307          (!chain [(f ((a + b) * c))
308                  --> ((f (a + b)) *' (f c))
309                      [Z-NN.multiplicative-homomorphism]
310                  --> (((f a) +' (f b)) *' (f c))
311                      [Z-NN.additive-homomorphism]
312                  --> (((f a) *' (f c)) +' ((f b) *' (f c)))
313                      [NN.Times.Right-Distributive]
314                  <-- ((f (a * c)) +' (f (b * c)))
315                      [Z-NN.multiplicative-homomorphism]
316                  <-- (f (a * c + b * c)) [Z-NN.additive-homomorphism]])}
317 conclude ((a + b) * c = a * c + b * c)
318   (!chain [(a + b) * c
319            <-- (g (f ((a + b) * c))) [Z-NN.inverse]
320            --> (g (f (a * c + b * c))) [f-application]
321            --> (a * c + b * c) [Z-NN.inverse]])
322
323 # Since we already have proved commutativity, we can use it for
324 # Left-Distributive.
325
326 conclude Left-Distributive
327 pick-any a:Z b:Z c:Z
328   (!chain [(c * (a + b))
329            --> ((a + b) * c) [commutative]
330            --> (a * c + b * c) [Right-Distributive]
331            --> (c * a + c * b) [commutative]])
332
333 define Right-Identity := (forall a . a * one = a)
334 define Left-Identity := (forall a . one * a = a)
335
336 datatype-cases Right-Identity {
337   (pos x) =>
338     (!chain [(pos x) * one
339              --> ((pos x) * (pos N.one)) [one-definition]
340              --> (pos (x * N.one)) [pos-pos]
341              --> (pos x) [N.Times.right-one]])
342 | (neg x) =>
343   (!chain [(neg x) * one
344            --> ((neg x) * (pos N.one)) [one-definition]
345            --> (neg (x * N.one)) [neg-pos]
346            --> (neg x) [N.Times.right-one]])
347 }

```

```

348
349 # Since we already have proved commutativity, we can use it for Left-Identity.
350
351 conclude Left-Identity
352 pick-any a:Z
353   (!chain [(one * a)
354            --> (a * one)           [commutative]
355            --> a                   [Right-Identity]])
356
357 define No-Zero-Divisors :=
358   (forall a b . a * b = zero ==> a = zero | b = zero)
359
360 datatype-cases No-Zero-Divisors {
361   (pos x) =>
362     datatype-cases
363       (forall ?b . (pos x) * ?b = zero ==>
364                   (pos x) = zero | ?b = zero) {
365         (pos y) =>
366           assume ((pos x) * (pos y) = zero)
367           let {C :=
368               (!chain->
369                [(pos (x * y))
370                 <-- ((pos x) * (pos y)) [pos-pos]
371                  --> zero                [((pos x) * (pos y) = zero)]
372                  --> (pos Top.zero)      [zero-definition]
373                  ==> (x * y = Top.zero)   [Z-structure-axioms]
374                  ==> (x = Top.zero | y = Top.zero) [N.Times.no-zero-divisors]]]}
375           (!cases C
376            assume (x = Top.zero)
377            let { _ := (!chain [(pos x)
378                               --> (pos Top.zero) [(x = Top.zero)]
379                               <-- zero           [zero-definition]]]}
380            (!left-either ((pos x) = zero) ((pos y) = zero))
381            assume (y = Top.zero)
382            let { _ := (!chain [(pos y)
383                               --> (pos Top.zero) [(y = Top.zero)]
384                               <-- zero           [zero-definition]]]}
385            (!right-either ((pos x) = zero) ((pos y) = zero)))
386         | (neg y) =>
387           assume ((pos x) * (neg y) = zero)
388           let {C :=
389               (!chain->
390                [(neg (x * y))
391                 <-- ((pos x) * (neg y)) [pos-neg]
392                  --> zero                [((pos x) * (neg y) = zero)]
393                  --> (neg Top.zero)     [zero-property]
394                  ==> (x * y = Top.zero)  [Z-structure-axioms]
395                  ==> (x = Top.zero | y = Top.zero) [N.Times.no-zero-divisors]]]}
396           (!cases C
397            assume (x = Top.zero)
398            let { _ := (!chain [(pos x)
399                               --> (pos Top.zero) [(x = Top.zero)]
400                               <-- zero           [zero-definition]]]}
401            (!left-either ((pos x) = zero) ((neg y) = zero))
402            assume (y = Top.zero)
403            let { _ := (!chain [(neg y)
404                               --> (neg Top.zero) [(y = Top.zero)]
405                               <-- zero           [zero-property]]]}
406            (!right-either ((pos x) = zero) ((neg y) = zero)))
407         }
408   | (neg x) =>
409     datatype-cases
410       (forall ?b . (neg x) * ?b = zero ==> (neg x) = zero | ?b = zero)
411     { (pos y) =>
412       assume ((neg x) * (pos y) = zero)
413       let {C := (!chain->
414               [(neg (x * y))
415                <-- ((neg x) * (pos y)) [neg-pos]
416                 --> zero                [(((neg x) * (pos y)) = zero)]
417                 --> (neg Top.zero)     [zero-property]

```

```

418         ==> (x * y = Top.zero)          [Z-structure-axioms]
419         ==> (x = Top.zero | y = Top.zero) [N.Times.no-zero-divisors]])}
420     (!cases C
421       assume (x = Top.zero)
422       let {_ := (!chain [(neg x)
423         --> (neg Top.zero) [(x = Top.zero)]
424         <-- zero          [zero-property]])}
425       (!left-either ((neg x) = zero) ((pos y) = zero))
426       assume (y = Top.zero)
427       let {_ := (!chain [(pos y)
428         --> (pos Top.zero) [(y = Top.zero)]
429         <-- zero          [zero-definition]])}
430       (!right-either ((neg x) = zero) ((pos y) = zero)))
431 | (neg y) =>
432   assume ((neg x) * (neg y) = zero)
433   let {C := (!chain->
434     [(pos (x * y))
435     <-- ((neg x) * (neg y)) [neg-neg]
436     --> zero                [((neg x) * (neg y) = zero)]
437     --> (pos Top.zero)     [zero-definition]
438     ==> (x * y = Top.zero) [Z-structure-axioms]
439     ==> (x = Top.zero | y = Top.zero)
440       [N.Times.no-zero-divisors]])}
441   (!cases C
442     assume (x = Top.zero)
443     let {_ := (!chain [(neg x)
444       --> (neg Top.zero) [(x = Top.zero)]
445       <-- zero          [zero-property]])}
446     (!left-either ((neg x) = zero) ((neg y) = zero))
447     assume (y = Top.zero)
448     let {_ := (!chain [(neg y)
449       --> (neg Top.zero) [(y = Top.zero)]
450       <-- zero          [zero-property]])}
451     (!right-either ((neg x) = zero) ((neg y) = zero)))
452   }
453 }
454
455 define Nonzero-Product :=
456   (forall a b . ~ (a = zero | b = zero) ==> a * b /= zero)
457
458 conclude Nonzero-Product
459 pick-any a b
460   (!contra-pos (!instance No-Zero-Divisors [a b]))
461 } # Times
462 } # Z

```