

## lib/main/nat-half.ath

```

1 load "nat-times"
2 load "nat-less"
3
4 extend-module N {
5
6 declare half: [N] -> N [[int->nat]]
7
8 module half {
9
10  assert* axioms :=
11    [(half zero = zero)
12     (half S zero = zero)
13     (half S S n = S half n)]
14
15  define [if-zero if-one nonzero-nonone] := axioms
16
17  (print "\nHalf of 20: " (eval half 20) "and half of 21: " (eval half 21) "\n")
18
19  define double := (forall n . half (n + n) = n)
20
21  by-induction double {
22    zero => (!chain [(half (zero + zero))
23                    --> (half zero)           [Plus.right-zero]
24                    --> zero                 [if-zero]])
25  | (S zero) =>
26    (!chain [(half (S zero + S zero))
27            --> (half S (S zero + zero))   [Plus.right-nonzero]
28            --> (half S S (zero + zero))   [Plus.left-nonzero]
29            --> (half S S zero)            [Plus.right-zero]
30            --> (S half zero)              [nonzero-nonone]
31            --> (S zero)                   [if-zero]])
32  | (S (S n)) =>
33    let {induction-hypothesis := (half (n + n) = n)}
34      (!chain
35        [(half (S S n + S S n))
36         --> (half S (S S n + S n))         [Plus.right-nonzero]
37         --> (half S S (S S n + n))        [Plus.right-nonzero]
38         --> (S half (S S n + n))          [nonzero-nonone]
39         --> (S half S (S n + n))          [Plus.left-nonzero]
40         --> (S half S S (n + n))          [Plus.left-nonzero]
41         --> (S S half (n + n))            [nonzero-nonone]
42         --> (S S n)                       [induction-hypothesis]])
43  }
44
45  define Times-two := (forall x . half (two * x) = x)
46
47  conclude Times-two
48  pick-any x
49    (!chain [(half (two * x))
50            --> (half (x + x))           [Times.two-times]
51            --> x                         [double]])
52
53  define twice := (forall x . two * half S S x = S S (two * half x))
54
55  conclude twice
56  pick-any x
57    (!chain [(two * half S S x)
58            --> (two * S half x)           [nonzero-nonone]
59            --> ((S half x) + (S half x)) [Times.two-times]
60            --> (S ((half x) + S half x)) [Plus.left-nonzero]
61            --> (S S ((half x) + half x)) [Plus.right-nonzero]
62            --> (S S (two * half x))      [Times.two-times]])
63
64  define two-plus := (forall x y . half (two * x + y) = x + half y)
65
66  by-induction two-plus {
67    zero =>
68      pick-any y

```

```

69     (!chain [(half ((two * zero) + y))
70             --> (half (zero + y))      [Times.right-zero]
71             --> (half y)                [Plus.left-zero]
72             <-- (zero + half y)        [Plus.left-zero]])
73 | (S zero) =>
74   pick-any y
75     (!chain [(half (two * (S zero) + y))
76             <-- (half (two * one + y))  [one-definition]
77             --> (half (two + y))        [Times.right-one]
78             --> (half ((S one) + y))    [two-definition]
79             --> (half S (one + y))      [Plus.left-nonzero]
80             --> (half S ((S zero) + y)) [one-definition]
81             --> (half S S (zero + y))   [Plus.left-nonzero]
82             --> (half S S y)            [Plus.left-zero]
83             --> (S half y)              [nonzero-nonone]
84             <-- (one + half y)          [Plus.left-one]
85             --> ((S zero) + half y)     [one-definition]])
86 | (S (S x)) =>
87   let {induction-hypothesis :=
88         (forall ?y . half (two * x + ?y) = x + half ?y)}
89   pick-any y
90     (!chain
91       [(half (two * (S S x)) + y)
92        --> (half ((S S x) + (S S x)) + y) [Times.two-times]
93        --> (half (S (S ((x + S S x) + y)))) [Plus.left-nonzero]
94        --> (S half ((x + (S (S x))) + y)) [nonzero-nonone]
95        --> (S half ((S S (x + x)) + y)) [Plus.right-nonzero]
96        --> (S half S S ((x + x) + y)) [Plus.left-nonzero]
97        --> (S S half ((x + x) + y)) [nonzero-nonone]
98        <-- (S S half (two * x + y)) [Times.two-times]
99        --> (S S (x + half y)) [induction-hypothesis]
100        <-- (S ((S x) + half y)) [Plus.left-nonzero]
101        <-- ((S S x) + half y) [Plus.left-nonzero]])
102   }
103
104 define less-S := (forall n . half n < S n)
105 define less := (forall n . n /= zero ==> half n < n)
106
107 by-induction less-S {
108   zero => (!chain-> [true
109                   ==> (zero < S zero) [Less.<S]
110                   ==> (half zero < S zero) [if-zero]])
111 | (S zero) =>
112   let {C := (!chain-> [true
113                   ==> (zero < S zero) [Less.<S]
114                   ==> (half S zero < S zero) [if-one]])}
115     (!chain-> [true
116             ==> (S zero < S S zero) [Less.<S]
117             ==> (S zero < S S zero & C) [augment]
118             ==> (half S zero < S S zero) [Less.transitive]])
119 | (n as (S (S n'))) =>
120   let {ind-hyp := (half n' < S n');
121         C := (!chain-> [true
122                   ==> (S S n' < S S S n') [Less.<S]])}
123     (!chain-> [ind-hyp
124             ==> (S half n' < S S n') [Less.injective]
125             ==> (half S S n' < S S n') [nonzero-nonone]
126             ==> (half S S n' < S S n' & C) [augment]
127             ==> (half S S n' < S S S n') [Less.transitive]])
128   }
129
130 datatype-cases less {
131   zero => assume (zero /= zero)
132           (!from-complements (half zero < zero)
133                               (!reflex zero)
134                               (zero /= zero))
135 | (S zero) =>
136   assume (S zero /= zero)
137     (!chain-> [true
138             ==> (zero < S zero) [Less.<S]

```

```

139         ==> (half S zero < S zero) [if-one]])
140 | (n as (S (S m))) =>
141   assume (S S m /= zero)
142     (!chain-> [true
143       ==> (half m < S m) [less-S]
144       ==> (S half m < S S m) [Less.injective]
145       ==> (half S S m < S S m) [nonzero-nonone]])
146 }
147
148 define equal-zero :=
149   (forall x . half x = zero ==> x = zero | x = one)
150
151 datatype-cases equal-zero {
152   zero =>
153     assume (half zero = zero)
154     (!left-either (!reflex zero) (zero = one))
155 | (S zero) =>
156   assume (half S zero = zero)
157   let {B := (!chain [(S zero) = one [one-definition]])}
158     (!right-either (S zero = zero) B)
159 | (S (S n)) =>
160   assume A := (half S S n = zero)
161   let {is := (!chain-> [zero = (half S S n) [A]
162     = (S half n) [nonzero-nonone]
163     ==> (S half n = zero) [sym]])};
164     is-not := (!chain->
165       [true ==> (S half n /= zero) [S-not-zero]])}
166     (!from-complements (S S n = zero | S S n = one) is is-not)
167 }
168
169 define less-equal := (forall n . half n <= n)
170
171 datatype-cases less-equal {
172   zero =>
173     conclude (half zero <= zero)
174     (!chain-> [true ==> (zero <= zero) [Less=.reflexive]
175       ==> (half zero <= zero) [if-zero]])
176 | (S n) =>
177   conclude (half S n <= S n)
178   (!chain-> [true ==> (S n /= zero) [S-not-zero]
179     ==> (half S n < S n) [less]
180     ==> (half S n <= S n) [Less=.Implied-by-<]])
181 }
182
183 define less-equal-1 := (forall n . n /= zero ==> S half n <= n)
184
185 datatype-cases less-equal-1 {
186   zero =>
187     conclude (zero /= zero ==> S half zero <= zero)
188     assume (zero /= zero)
189     (!from-complements (S half zero <= zero)
190       (!reflex zero) (zero /= zero))
191 | (S zero) =>
192   conclude (S zero /= zero ==> S half S zero <= S zero)
193   assume (S zero /= zero)
194     (!chain-> [true ==> (S zero <= S zero) [Less=.reflexive]
195       ==> (S half S zero <= S zero) [if-one]])
196 | (S (S n)) =>
197   conclude (S S n /= zero ==> S half S S n <= S S n)
198   assume (S S n /= zero)
199     (!chain-> [true ==> (half n <= n) [less-equal]
200       ==> (S half n <= S n) [Less=.injective]
201       ==> (S S half n <= S S n) [Less=.injective]
202       ==> (S half S S n <= S S n) [nonzero-nonone]])
203 }
204
205
206
207 } # close module half
208

```

```

209 declare even, odd: [N] -> Boolean [[int->nat]]
210 module EO {
211
212   assert* even-definition := [(even x <==> two * half x = x)]
213
214   assert* odd-definition := [(odd x <==> two * (half x) + one = x)]
215
216
217   (print "\nis 20 even?: " (eval even 20))
218   (print "\nis 20 odd?: " (eval odd 20))
219   (print "\nis 21 even?: " (eval even 21))
220   (print "\nis 21 odd?: " (eval odd 21))
221
222   #assert even-definition := (fun [(even x) <==> (two * half x = x)])
223   #assert odd-definition := (fun [(odd ?x) <==> (two * (half x) + one = x)])
224
225   define even-zero := (even zero)
226   define odd-one := (odd S zero)
227   define even-S-S := (forall n . even S S n <==> even n)
228   define odd-S-S := (forall n . odd S S n <==> odd n)
229   define odd-if-not-even := (forall x . ~ even x ==> odd x)
230   define not-odd-if-even := (forall x . even x ==> ~ odd x)
231   define even-iff-not-odd := (forall x . even x <==> ~ odd x)
232   define not-even-if-odd := (forall x . odd x ==> ~ even x)
233   define half-nonzero-if-nonzero-even :=
234     (forall n . n /= zero & even n ==> half n /= zero)
235   define half-nonzero-if-nonone-odd :=
236     (forall n . n /= one & odd n ==> half n /= zero)
237   define even-twice := (forall x . even (two * x))
238   define even-square := (forall x . even x <==> even square x)
239
240   (!force even-zero)
241   (!force not-odd-if-even)
242
243   conclude even-S-S
244   pick-any n
245     let {right := assume (even S S n)
246         (!chain->
247           [(S S (two * (half n)))
248            <-- (two * half S S n)           [half.twice]
249            --> (S S n)                       [even-definition]
250            ==> ((S (two * half n)) = S n)    [S-injective]
251            ==> (two * (half n) = n)         [S-injective]
252            ==> (even n)                     [even-definition])];
253         left := assume (even n)
254           (!chain->
255             [(two * half S S n)
256              --> (S S (two * half n))       [half.twice]
257              --> (S S n)                     [even-definition]
258              ==> (even S S n)               [even-definition])];
259         (!equiv right left)
260
261   conclude odd-S-S
262   pick-any n
263     let {right :=
264         assume (odd S S n)
265         (!chain->
266           [(S S S (two * half n))
267            <-- (S (two * half S S n))         [half.twice]
268            <-- (two * (half S S n) + one)     [Plus.right-one]
269            --> (S S n)                       [odd-definition]
270            ==> (S S (two * half n) = S n)    [S-injective]
271            ==> (S (two * half n) = n)       [S-injective]
272            ==> (two * (half n) + one = n)   [Plus.right-one]
273            ==> (odd n)                       [odd-definition])];
274         left :=
275         assume (odd n)
276         (!chain->
277           [(two * (half S S n)) + one)
278            --> (S (two * half S S n))       [Plus.right-one]

```

```

279         --> (S S S (two * half n))           [half.twice]
280         <-- (S S (two * (half n) + one))     [Plus.right-one]
281         --> (S S n)                           [odd-definition]
282         ==> (odd S S n)                       [odd-definition]])}
283     (!equiv right left)
284
285 by-induction odd-if-not-even {
286   zero => assume (~ even zero)
287         (!from-complements
288          (odd zero) even-zero (~ even zero))
289 | (S zero) =>
290   assume (~ (even (S zero)))
291     (!chain->
292      [((two * (half S zero)) + one)
293       --> (S (two * half S zero))           [Plus.right-one]
294       --> (S (two * zero))                 [half.if-one]
295       --> (S zero)                         [Times.right-zero]
296       ==> (odd S zero)                     [odd-definition]])
297 | (S (S x)) =>
298   let {induction-hypothesis := (~ even x ==> odd x)}
299       conclude (~ even S S x ==> odd S S x)
300       assume hyp := (~ even S S x)
301       let {_ := (!by-contradiction (~ even x)
302              (!chain [(even x)
303                       ==> (even S S x)           [even-S-S]
304                       ==> (hyp & even S S x)     [augment]
305                       ==> false                 [prop-taut]]))}
306         (!chain-> [(~ even x)
307                   ==> (odd x)                   [induction-hypothesis]
308                   ==> (odd S S x)              [odd-S-S]])
309 }
310
311 conclude even-zero
312   (!chain-> [(two * half zero)
313             --> ((half zero) + (half zero)) [Times.two-times]
314             --> (zero + zero)             [half.if-zero]
315             --> zero                       [Plus.right-zero]
316             ==> (even zero)               [even-definition]])
317
318
319 conclude odd-one
320   (!chain-> [(two * (half S zero) + one)
321             --> (S (two * (half S zero)))   [Plus.right-one]
322             --> (S (two * zero))           [half.if-one]
323             --> (S zero)                   [Times.right-zero]
324             ==> (odd S zero)               [odd-definition]])
325
326 conclude even-twice
327   pick-any x
328     (!chain-> [(two * half (two * x))
329               --> (two * x)                 [half.Times-two]
330               ==> (even (two * x))         [even-definition]])
331
332 declare square: [N] -> N [[int->nat]]
333 module square {
334   assert* definition := [(square x = x * x)]
335
336   (print "\nsquare of 12: " (eval square 12) "\n")
337 } # close module square
338
339
340 define even-square := (forall x . even x <==> even square x)
341
342 conclude even-square
343   pick-any x
344     let {right :=
345           assume (even x)
346           let {i := conclude (two * half square x = square x)
347                   (!combine-equations
348                    (!chain

```

```

349      [(two * half square x)
350 <-- (two * half square (two * half x))
351      [even-definition]
352 --> (two * half ((two * (half x)) *
353      (two * (half x))))
354      [square.definition]
355 --> (two * half two * ((half x) * (two * half x)))
356      [Times.associative]
357 --> (two * ((half x) * (two * half x)))
358      [half.Times-two]]
359      (!chain
360      [(square x)
361 <-- (square (two * half x))
362      [even-definition]
363 --> ((two * half x) * (two * half x))
364      [square.definition]
365 --> (two * ((half x) * (two * half x)))
366      [Times.associative]]))
367      (!chain-> [i ==> (even square x) [even-definition]]);
368 left :=
369   assume (even square x)
370   (!by-contradiction (even x)
371   assume hyp := (~ even x)
372   let { _ := (!chain-> [hyp ==> (odd x) [odd-if-not-even]]);
373     A := conclude (two * (half square x) + one = square x)
374     let { i := conclude (square x =
375     two * ((half x) * x) + x)
376     (!chain
377     [(square x)
378 --> (x * x) [square.definition]
379 <-- (((two * half x) + one) * x)
380     [odd-definition]
381 --> ((two * half x) * x + one * x)
382     [Times.right-distributive]
383 --> (two * ((half x) * x) + x)
384     [Times.associative
385     Times.left-one]]);
386     ii := conclude (half square x =
387     (half x) * x + half x)
388     (!chain
389     [(half square x)
390 --> (half (two * ((half x) * x) + x))
391     [i]
392 --> ((half x) * x + half x)
393     [half.two-plus]]);
394     iii := conclude
395     (two * (half square x) + one =
396     two * ((half x) * x) + x)
397     (!chain
398     [(two * (half square x) + one)
399 --> (two * ((half x) * x + half x)
400     + one) [ii]
401 --> ((two * ((half x) * x) +
402     two * half x) + one)
403     [Times.left-distributive]
404 --> (two * ((half x) * x) +
405     two * (half x) + one)
406     [Plus.associative]
407 --> (two * ((half x) * x) + x)
408     [odd-definition]]);
409     (!combine-equations iii i)}
410   (!absurd
411   (!chain-> [A ==> (odd square x) [odd-definition]]
412   (!chain-> [(even square x) ==> (~ odd square x)
413   [not-odd-if-even]])))
414   (!equiv right left)
415
416 conclude half-nonzero-if-nonzero-even
417 pick-any n
418 assume (n /= zero & even n)

```

```

419     (!by-contradiction (half n /= zero)
420     assume opposite := (half n = zero)
421     let {is := (!chain [n <-- (two * half n) [even-definition]
422                       --> (two * zero) [opposite]
423                       --> zero [Times.right-zero]]);
424     is-not := (n /= zero)}
425     (!absurd is is-not))
426
427 conclude half-nonzero-if-nonone-odd
428 pick-any n
429     assume (n /= one & odd n)
430     (!by-contradiction (half n /= zero)
431     assume opposite := (half n = zero)
432     let {n-one := (!chain
433               [n <-- (two * (half n) + one) [odd-definition]
434                 --> (two * zero + one) [opposite]
435                 --> (zero + one) [Times.right-zero]
436                 --> one [Plus.left-zero]]);
437     (!absurd n-one (n /= one))}
438
439
440 } # close EO
441
442 declare parity: [N] -> N
443
444 module parity {
445 assert if-even := (forall n . even n ==> parity n = zero)
446 assert if-odd := (forall n . ~ even n ==> parity n = one)
447
448 define half-case := (forall n . two * (half n) + parity n = n)
449 define plus-half := (forall n . n /= zero ==> (half n) + parity n /= zero)
450
451 conclude half-case
452 pick-any n
453     (!two-cases
454     assume (even n)
455     (!chain [(two * (half n) + parity n)
456             --> (two * (half n) + zero) [if-even]
457             --> (two * half n) [Plus.right-zero]
458             --> n [EO.even-definition]])
459     assume (~ (even n))
460     (!chain-> [(~ even n)
461             ==> (odd n) [EO.odd-if-not-even]
462             ==> (two * (half n) + one = n) [EO.odd-definition]
463             ==> (two * (half n) + parity n = n) [if-odd]]))
464
465 conclude plus-half
466 pick-any n
467     assume A := (n /= zero)
468     (!two-cases
469     assume B := (even n)
470     let {C := (!chain
471               [(half n) + parity n)
472               = ((half n) + zero) [if-even]
473               = (half n) [Plus.right-zero]})
474     (!chain-> [(A & B)
475             ==> (half n /= zero)
476             [EO.half-nonzero-if-nonzero-even]
477             ==> ((half n) + parity n /= zero) [C]])
478     assume (~ even n)
479     let {C := (!chain
480               [(half n) + parity n)
481               = ((half n) + S zero) [if-odd one-definition]
482               = (S ((half n) + zero)) [Plus.right-nonzero]})
483     (!chain-> [true ==> (S ((half n) + zero) /= zero)
484             [S-not-zero]
485             ==> ((half n) + parity n /= zero) [C]])
486 } # parity
487
488 } # N

```