

lib/main/nat-half0.ath

```

1  # (half n): floor of (n/2)
2
3  load "nat-less.ath"
4
5  #-----
6  extend-module N {
7  declare half: [N] -> N
8
9  define [n x y] := [?n:N ?x:N ?y:N]
10
11 module half {
12
13 assert axioms :=
14   (fun [ (half zero)      = zero
15         (half (S zero))  = zero
16         (half (S S n))   = (S half n)])
17 define [if-zero if-one nonzero-nonone] := axioms
18
19 define less-S := (forall n . half n < S n)
20 define less := (forall n . n != zero ==> half n < n)
21 define less-equal := (forall n . half n <= n)
22 define less-equal-1 := (forall n . n != zero ==> S half n <= n)
23 define double := (forall n . half (n + n) = n)
24 define Times-two := (forall x . half (two * x) = x)
25 define two-plus := (forall x y . half (two * x + y) = x + half y)
26 define equal-zero := (forall x . half x = zero ==> x = zero | x = one)
27 define twice := (forall x . two * half S S x = S S (two * half x))
28
29 by-induction less-S {
30   zero => (!chain-> [true ==> (zero < S zero)          [Less.<S]
31                      ==> (half zero < S zero)         [if-zero]])
32 | (S zero) =>
33   let {C := (!chain-> [true ==> (zero < S zero)          [Less.<S]
34                      ==> ((half S zero) < S zero)      [if-one]])}
35   (!chain-> [true ==> (S zero < S S zero)                [Less.<S]
36             ==> (S zero < S S zero & C)                 [augment]
37             ==> (half S zero < S S zero)                [Less.transitive]])
38 | (S (S n)) =>
39   let {ind-hyp := (half n < S n);
40       C := (!chain-> [true ==> (S S n < S S S n) [Less.<S]])}
41   (!chain->
42     [ind-hyp ==> (S half n < S S n)                    [Less.injective]
43               ==> (half S S n < S S n)                  [nonzero-nonone]
44               ==> (half S S n < S S n & C)              [augment]
45               ==> (half S S n < S S S n)                [Less.transitive]])
46 }
47
48 datatype-cases less {
49   zero => assume (zero != zero)
50         (!from-complements (half zero < zero)
51          (!reflex zero) (zero != zero))
52 | (S zero) => assume (S zero != zero)
53         (!chain-> [true ==> (zero < S zero)          [Less.<S]
54                      ==> (half S zero < S zero)      [if-one]])
55 | (S (S n)) => assume (S S n != zero)
56         (!chain->
57           [true ==> (half n < S n)                    [less-S]
58               ==> (S half n < S S n)                  [Less.injective]
59               ==> (half S S n < S S n)                  [nonzero-nonone]])
60 }
61
62 datatype-cases less-equal {
63   zero =>
64     conclude (half zero <= zero)
65     (!chain-> [true ==> (zero <= zero)                [Less=.reflexive]
66              ==> (half zero <= zero)                [if-zero]])
67 | (S n) =>
68   conclude (half S n <= S n)

```

```

69      (!chain-> [true ==> (S n /= zero)      [S-not-zero]
70                  ==> (half S n < S n)      [less]
71                  ==> (half S n <= S n)     [Less=.Implied-by-<]])
72  }
73
74  datatype-cases less-equal-1 {
75    zero =>
76      conclude (zero /= zero ==> S half zero <= zero)
77      assume (zero /= zero)
78      (!from-complements (S half zero <= zero)
79      (!reflex zero) (zero /= zero))
80  | (S zero) =>
81      conclude (S zero /= zero ==> S half S zero <= S zero)
82      assume (S zero /= zero)
83      (!chain-> [true ==> (S zero <= S zero)      [Less=.reflexive]
84                  ==> (S half S zero <= S zero) [if-one]])
85  | (S (S n)) =>
86      conclude (S S n /= zero ==> S half S S n <= S S n)
87      assume (S S n /= zero)
88      (!chain-> [true ==> (half n <= n)            [less-equal]
89                  ==> (S half n <= S n)          [Less=.injective]
90                  ==> (S S half n <= S S n)      [Less=.injective]
91                  ==> (S half S S n <= S S n) [nonzero-nonone]])
92  }
93
94  by-induction double {
95    zero => (!chain [(half (zero + zero))
96                    --> (half zero)          [Plus.right-zero]
97                    --> zero                  [if-zero]])
98  | (S zero) =>
99    (!chain [(half ((S zero) + (S zero)))
100            --> (half (S ((S zero) + zero))) [Plus.right-nonzero]
101            --> (half (S (S (zero + zero)))) [Plus.left-nonzero]
102            --> (half (S (S zero)))          [Plus.right-zero]
103            --> (S half zero)                [nonzero-nonone]
104            --> (S zero)                     [if-zero]])
105  | (S (S n)) =>
106    let {induction-hypothesis := (half (n + n) = n)}
107    (!chain
108      [(half ((S S n) + (S S n)))
109       --> (half S ((S S n) + (S n)))      [Plus.right-nonzero]
110       --> (half S S ((S S n) + n))        [Plus.right-nonzero]
111       --> (S half ((S S n) + n))          [nonzero-nonone]
112       --> (S half (S ((S n) + n)))        [Plus.left-nonzero]
113       --> (S half (S S (n + n)))          [Plus.left-nonzero]
114       --> (S S (half (n + n)))            [nonzero-nonone]
115       --> (S S n)                        [induction-hypothesis]])
116  }
117
118  conclude Times-two
119  pick-any x
120    (!chain [(half (two * x))
121            --> (half (x + x))      [Times.two-times]
122            --> x                  [double]])
123
124  by-induction two-plus {
125    zero =>
126      pick-any y
127      (!chain [(half ((two * zero) + y))
128              --> (half (zero + y))    [Times.right-zero]
129              --> (half y)             [Plus.left-zero]
130              <-- (zero + half y)      [Plus.left-zero]])
131  | (S zero) =>
132    pick-any y
133    (!chain [(half (two * (S zero) + y))
134            <-- (half (two * one + y))  [one-definition]
135            --> (half (two + y))        [Times.right-one]
136            --> (half ((S one) + y))    [two-definition]
137            --> (half S (one + y))      [Plus.left-nonzero]
138            --> (half S ((S zero) + y)) [one-definition]

```

```

139         --> (half S S (zero + y))      [Plus.left-nonzero]
140         --> (half S S y)                [Plus.left-zero]
141         --> (S half y)                  [nonzero-nonone]
142         <-- (one + half y)              [Plus.left-one]
143         --> ((S zero) + half y)        [one-definition]]
144 | (S (S x)) =>
145   let {induction-hypothesis :=
146     (forall ?y . half (two * x + ?y) = x + half ?y)}
147   pick-any y
148   (!chain
149     [(half (two * (S S x)) + y)
150      --> (half (((S S x) + (S S x)) + y))      [Times.two-times]
151      --> (half (S (S ((x + S S x) + y))))      [Plus.left-nonzero]
152      --> (S half ((x + (S (S x))) + y))        [nonzero-nonone]
153      --> (S half ((S S (x + x)) + y))          [Plus.right-nonzero]
154      --> (S half S S ((x + x) + y))            [Plus.left-nonzero]
155      --> (S S half ((x + x) + y))              [nonzero-nonone]
156      <-- (S S half (two * x + y))              [Times.two-times]
157      --> (S S (x + half y))                    [induction-hypothesis]
158      <-- (S ((S x) + half y))                  [Plus.left-nonzero]
159      <-- ((S S x) + half y)                    [Plus.left-nonzero]])
160   }
161
162 datatype-cases equal-zero {
163   zero =>
164     assume (half zero = zero)
165     (!left-either (!reflex zero) (zero = one))
166 | (S zero) =>
167   assume (half S zero = zero)
168   let {B := (!chain [(S zero) = one      [one-definition]]})
169   (!right-either (S zero = zero) B)
170 | (S (S n)) =>
171   assume a := (half S S n = zero)
172   let {is := (!chain-> [zero = (half S S n)      [a]
173                       = (S half n)              [nonzero-nonone]
174                       ==> (S half n = zero)      [sym]]);
175     is-not := (!chain->
176       [true ==> (S half n /= zero) [S-not-zero]]})
177   (!from-complements (S S n = zero | S S n = one) is is-not)
178 }
179
180 conclude twice
181 pick-any x
182   (!chain [(two * half S S x)
183            --> (two * S half x)                [nonzero-nonone]
184            --> ((S half x) + (S half x))      [Times.two-times]
185            --> (S ((half x) + (S half x)))    [Plus.left-nonzero]
186            --> (S S ((half x) + (half x)))    [Plus.right-nonzero]
187            --> (S S (two * (half x)))         [Times.two-times]])
188 } # half
189
190 #.....
191
192 declare Even, Odd: [N] -> Boolean
193 module EO {
194
195   assert Even-definition := (fun [(Even x) <==> (two * half x = x)])
196   assert Odd-definition := (fun [(Odd ?x) <==> (two * (half x) + one = x)])
197
198   define Even-zero := (Even zero)
199   define Odd-one := (Odd S zero)
200   define Even-S-S := (forall n . Even S S n <==> Even n)
201   define Odd-S-S := (forall n . Odd S S n <==> Odd n)
202   define Odd-if-not-Even := (forall x . ~ Even x ==> Odd x)
203   define not-Odd-if-Even := (forall x . Even x ==> ~ Odd x)
204   define Even-iff-not-Odd := (forall x . Even x <==> ~ Odd x)
205   define not-Even-if-Odd := (forall x . Odd x ==> ~ Even x)
206   define half-nonzero-if-nonzero-Even :=
207     (forall n . n /= zero & Even n ==> half n /= zero)
208   define half-nonzero-if-nonone-Odd :=

```

```

209 (forall n . n /= one & Odd n ==> half n /= zero)
210 define Even-twice := (forall x . Even (two * x))
211 define Even-square := (forall x . Even x <==> Even square x)
212
213 conclude Even-zero
214 (!chain-> [(two * half zero)
215 --> ((half zero) + (half zero)) [Times.two-times]
216 --> (zero + zero) [half.if-zero]
217 --> zero [Plus.right-zero]
218 ==> (Even zero) [Even-definition]])
219
220 conclude Odd-one
221 (!chain-> [(two * (half S zero) + one)
222 --> (S (two * (half S zero))) [Plus.right-one]
223 --> (S (two * zero)) [half.if-one]
224 --> (S zero) [Times.right-zero]
225 ==> (Odd S zero) [Odd-definition]])
226
227 conclude Even-S-S
228 pick-any n
229 let {right := assume (Even S S n)
230 (!chain->
231 [(S S (two * (half n)))
232 <-- (two * half S S n) [half.twice]
233 --> (S S n) [Even-definition]
234 ==> ((S (two * half n)) = S n) [S-injective]
235 ==> (two * (half n) = n) [S-injective]
236 ==> (Even n) [Even-definition]]);
237 left := assume (Even n)
238 (!chain->
239 [(two * half S S n)
240 --> (S S (two * half n)) [half.twice]
241 --> (S S n) [Even-definition]
242 ==> (Even S S n) [Even-definition]]);
243 (!equiv right left)
244
245 conclude Odd-S-S
246 pick-any n
247 let {right :=
248 assume (Odd S S n)
249 (!chain->
250 [(S S S (two * half n))
251 <-- (S (two * half S S n)) [half.twice]
252 <-- (two * (half S S n) + one) [Plus.right-one]
253 --> (S S n) [Odd-definition]
254 ==> (S S (two * half n) = S n) [S-injective]
255 ==> (S (two * half n) = n) [S-injective]
256 ==> (two * (half n) + one = n) [Plus.right-one]
257 ==> (Odd n) [Odd-definition]]);
258 left :=
259 assume (Odd n)
260 (!chain->
261 [((two * (half S S n)) + one)
262 --> (S (two * half S S n)) [Plus.right-one]
263 --> (S S S (two * half n)) [half.twice]
264 <-- (S S (two * (half n) + one)) [Plus.right-one]
265 --> (S S n) [Odd-definition]
266 ==> (Odd S S n) [Odd-definition]]);
267 (!equiv right left)
268
269 by-induction Odd-if-not-Even {
270 zero => assume (~ Even zero)
271 (!from-complements
272 (Odd zero) Even-zero (~ Even zero))
273 | (S zero) =>
274 assume (~ (Even (S zero)))
275 (!chain->
276 [((two * (half S zero)) + one)
277 --> (S (two * half S zero)) [Plus.right-one]
278 --> (S (two * zero)) [half.if-one]

```

```

279      --> (S zero)                                [Times.right-zero]
280      ==> (Odd S zero)                            [Odd-definition]]
281 | (S (S x)) =>
282   let {induction-hypothesis := (¬ Even x ==> Odd x)}
283   conclude (¬ Even S S x ==> Odd S S x)
284   assume hyp := (¬ Even S S x)
285   let {_ := (!by-contradiction (¬ Even x)
286     (!chain [(Even x)
287       ==> (Even S S x)           [Even-S-S]
288       ==> (hyp & Even S S x)    [augment]
289       ==> false                 [prop-taut]]))}
290     (!chain-> [(¬ Even x)
291       ==> (Odd x)               [induction-hypothesis]
292       ==> (Odd S S x)          [Odd-S-S]])
293 }
294
295 conclude not-Odd-if-Even
296 pick-any x
297 assume (Even x)
298 (!by-contradiction (¬ Odd x)
299   assume (Odd x)
300   let {equal :=
301     (!chain
302       [(S x)
303         <-- (S (two * half x))    [Even-definition]
304         <-- (two * (half x) + one) [Plus.right-one]
305         --> x                    [Odd-definition]]);
306     unequal :=
307       (!chain-> [true ==> (S x /= x) [S-not-same]])}
308     (!absurd equal unequal))
309
310 conclude Even-iff-not-Odd
311 pick-any x
312 let {right := (!chain
313   [(Even x) ==> (¬ Odd x)    [not-Odd-if-Even]]);
314   left := assume (¬ Odd x)
315     (!by-contradiction (Even x)
316       (!chain [(¬ Even x)
317         ==> (Odd x)           [Odd-if-not-Even]
318         ==> (¬ Odd x & Odd x) [augment]
319         ==> false            [prop-taut]]))}
320   (!equiv right left)
321
322 conclude not-Even-if-Odd
323 pick-any x
324 assume (Odd x)
325 (!by-contradiction (¬ Even x)
326   assume (Even x)
327   let {not-odd := (!chain-> [(Even x) ==> (¬ Odd x)
328     [not-Odd-if-Even]])}
329     (!absurd (Odd x) not-odd))
330
331 conclude half-nonzero-if-nonzero-Even
332 pick-any n
333 assume (n /= zero & Even n)
334 (!by-contradiction (half n /= zero)
335   assume opposite := (half n = zero)
336   let {is := (!chain [n <-- (two * half n) [Even-definition]
337     --> (two * zero)    [opposite]
338     --> zero           [Times.right-zero]]);
339     is-not := (n /= zero)}
340     (!absurd is is-not))
341
342 conclude half-nonzero-if-nonone-Odd
343 pick-any n
344 assume (n /= one & Odd n)
345 (!by-contradiction (half n /= zero)
346   assume opposite := (half n = zero)
347   let {n-one := (!chain
348     [n <-- (two * (half n) + one) [Odd-definition]

```

```

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

```

```

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

```