lib/main/nat-half0.ath

```
# (half n): floor of (n/2)

load "nat-less.ath"

#-------------------------------------------------------------------------
extend-module N {
declare half: [N] -> N

define [n x y] := [?n:N ?x:N ?y:N]

module half {

assert axioms :=
  (fun [(half zero)      = zero
        (half (S zero)) = zero
        (half (S S n))  = (S half n)])
define [if-zero if-one nonzero-nonone] := axioms

define less-S := (forall n . half n < S n)
define less := (forall n . n =/= zero ==> half n < n)
define less-equal := (forall n . half n <= n)
define less-equal-1 := (forall n . n =/= zero ==> S half n <= n)
define double := (forall n . half (n + n) = n)
define Times-two := (forall x . half (two * x) = x)
define two-plus := (forall x y . half (two * x + y) = x + half y)
define equal-zero := (forall x . half x = zero ==> x = zero | x = one)
define twice := (forall x . two * half S S x = S S (two * half x))

by-induction less-S {
  zero => (!chain-> [true ==> (zero < S zero)       [Less.<S]
                          ==> (half zero < S zero)   [if-zero]])
| (S zero) =>
    let {C := (!chain-> [true ==> (zero < S zero)             [Less.<S]
                              ==> ((half S zero) < S zero)    [if-one]])}
    (!chain-> [true ==> (S zero < S S zero)         [Less.<S]
                    ==> (S zero < S S zero & C)     [augment]
                    ==> (half S zero < S S zero)    [Less.transitive]])
| (S (S n)) =>
    let {ind-hyp := (half n < S n);
         C := (!chain-> [true ==> (S S n < S S S n) [Less.<S]])}
    (!chain->
     [ind-hyp ==> (S half n < S S n)              [Less.injective]
              ==> (half S S n < S S n)            [nonzero-nonone]
              ==> (half S S n < S S n & C)        [augment]
              ==> (half S S n < S S S n)          [Less.transitive]])
}

datatype-cases less {
  zero => assume (zero =/= zero)
            (!from-complements (half zero < zero)
                               (!reflex zero) (zero =/= zero))
| (S zero) => assume (S zero =/= zero)
                (!chain-> [true ==> (zero < S zero)          [Less.<S]
                                ==> (half S zero < S zero)   [if-one]])
| (S (S n)) => assume (S S n =/= zero)
                 (!chain->
                  [true ==> (half n < S n)          [less-S]
                        ==> (S half n < S S n)      [Less.injective]
                        ==> (half S S n < S S n)    [nonzero-nonone]])
}

datatype-cases less-equal {
  zero =>
  conclude (half zero <= zero)
    (!chain-> [true ==> (zero <= zero)         [Less=.reflexive]
                    ==> (half zero <= zero)    [if-zero]])
| (S n) =>
  conclude (half S n <= S n)
```

```
    (!chain-> [true ==> (S n =/= zero)       [S-not-zero]
                    ==> (half S n < S n)     [less]
                    ==> (half S n <= S n)    [Less=.Implied-by-<]])
}

datatype-cases less-equal-1 {
  zero =>
  conclude (zero =/= zero ==> S half zero <= zero)
    assume (zero =/= zero)
     (!from-complements (S half zero <= zero)
      (!reflex zero) (zero =/= zero))
| (S zero) =>
  conclude (S zero =/= zero ==> S half S zero <= S zero)
    assume (S zero =/= zero)
      (!chain-> [true ==> (S zero <= S zero)         [Less=.reflexive]
                      ==> (S half S zero <= S zero) [if-one]])
| (S (S n)) =>
  conclude (S S n =/= zero ==> S half S S n <= S S n)
    assume (S S n =/= zero)
      (!chain-> [true ==> (half n <= n)            [less-equal]
                      ==> (S half n <= S n)        [Less=.injective]
                      ==> (S S half n <= S S n)    [Less=.injective]
                      ==> (S half S S n <= S S n) [nonzero-nonone]])
}

by-induction double {
  zero => (!chain [(half (zero + zero))
                   --> (half zero)          [Plus.right-zero]
                   --> zero                 [if-zero]])
| (S zero) =>
    (!chain [(half ((S zero) + (S zero)))
             --> (half (S ((S zero) + zero))) [Plus.right-nonzero]
             --> (half (S (S (zero + zero)))) [Plus.left-nonzero]
             --> (half (S (S zero)))          [Plus.right-zero]
             --> (S half zero)                [nonzero-nonone]
             --> (S zero)                     [if-zero]])
| (S (S n)) =>
   let {induction-hypothesis := (half (n + n) = n)}
    (!chain
     [(half ((S S n) + (S S n)))
      --> (half S ((S S n) + (S n)))     [Plus.right-nonzero]
      --> (half S S ((S S n) + n))       [Plus.right-nonzero]
      --> (S half ((S S n) + n))         [nonzero-nonone]
      --> (S half (S ((S n) + n)))       [Plus.left-nonzero]
      --> (S half (S S (n + n)))         [Plus.left-nonzero]
      --> (S S (half (n + n)))           [nonzero-nonone]
      --> (S S n)                        [induction-hypothesis]])
}

conclude Times-two
  pick-any x
    (!chain [(half (two * x))
             --> (half (x + x))     [Times.two-times]
             --> x                  [double]])

by-induction two-plus {
  zero =>
   pick-any y
     (!chain [(half ((two * zero) + y))
              --> (half (zero + y))    [Times.right-zero]
              --> (half y)             [Plus.left-zero]
              <-- (zero + half y)      [Plus.left-zero]])
| (S zero) =>
    pick-any y
      (!chain [(half (two * (S zero) + y))
               <-- (half (two * one + y))     [one-definition]
               --> (half (two + y))           [Times.right-one]
               --> (half ((S one) + y))       [two-definition]
               --> (half S (one + y))         [Plus.left-nonzero]
               --> (half S ((S zero) + y))    [one-definition]
```

```
                 --> (half S S (zero + y))      [Plus.left-nonzero]
                 --> (half S S y)               [Plus.left-zero]
                 --> (S half y)                 [nonzero-nonone]
                 <-- (one + half y)             [Plus.left-one]
                 --> ((S zero) + half y)        [one-definition]])
| (S (S x)) =>
    let {induction-hypothesis :=
           (forall ?y . half (two * x + ?y) = x + half ?y)}
    pick-any y
      (!chain
       [(half (two * (S S x)) + y)
        --> (half (((S S x) + (S S x)) + y))     [Times.two-times]
        --> (half (S (S ((x + S S x) + y))))     [Plus.left-nonzero]
        --> (S half ((x + (S (S x))) + y))       [nonzero-nonone]
        --> (S half ((S S (x + x)) + y))         [Plus.right-nonzero]
        --> (S half S S ((x + x) + y))           [Plus.left-nonzero]
        --> (S S half ((x + x) + y))             [nonzero-nonone]
        <-- (S S half (two * x + y))             [Times.two-times]
        --> (S S (x + half y))                   [induction-hypothesis]
        <-- (S ((S x) + half y))                 [Plus.left-nonzero]
        <-- ((S S x) + half y)                   [Plus.left-nonzero]])
}

datatype-cases equal-zero {
  zero =>
    assume (half zero = zero)
      (!left-either (!reflex zero) (zero = one))
| (S zero) =>
    assume (half S zero = zero)
      let {B := (!chain [(S zero) = one   [one-definition]])}
        (!right-either (S zero = zero) B)
| (S (S n)) =>
    assume a := (half S S n = zero)
      let {is := (!chain-> [zero = (half S S n)      [a]
                                 = (S half n)        [nonzero-nonone]
                             ==> (S half n = zero)   [sym]]);
           is-not := (!chain->
                        [true ==> (S half n =/= zero)  [S-not-zero]])}
      (!from-complements (S S n = zero | S S n = one) is is-not)
}

conclude twice
  pick-any x
    (!chain [(two * half S S x)
             --> (two * S half x)             [nonzero-nonone]
             --> ((S half x) + (S half x))    [Times.two-times]
             --> (S ((half x) + (S half x)))  [Plus.left-nonzero]
             --> (S S ((half x) + (half x)))  [Plus.right-nonzero]
             --> (S S (two * (half x)))       [Times.two-times]])
} # half

#.....................................................................

declare Even, Odd: [N] -> Boolean
module EO {

assert Even-definition := (fun [(Even x) <==> (two * half x = x)])
assert Odd-definition := (fun [(Odd ?x) <==> (two * (half x) + one = x)])

define Even-zero := (Even zero)
define Odd-one := (Odd S zero)
define Even-S-S := (forall n . Even S S n <==> Even n)
define Odd-S-S := (forall n . Odd S S n <==> Odd n)
define Odd-if-not-Even  := (forall x . ~ Even x ==> Odd x)
define not-Odd-if-Even := (forall x . Even x ==> ~ Odd x)
define Even-iff-not-Odd := (forall x . Even x <==> ~ Odd x)
define not-Even-if-Odd := (forall x . Odd x ==> ~ Even x)
define half-nonzero-if-nonzero-Even :=
  (forall n . n =/= zero & Even n ==> half n =/= zero)
define half-nonzero-if-nonone-Odd :=
```

```
  (forall n . n =/= one & Odd n ==> half n =/= zero)
define Even-twice := (forall x . Even (two * x))
define Even-square := (forall x . Even x <==> Even square x)

conclude Even-zero
  (!chain-> [(two * half zero)
             --> ((half zero) + (half zero)) [Times.two-times]
             --> (zero + zero)               [half.if-zero]
             --> zero                        [Plus.right-zero]
             ==> (Even zero)                 [Even-definition]])

conclude Odd-one
  (!chain-> [(two * (half S zero) + one)
             --> (S (two * (half S zero)))   [Plus.right-one]
             --> (S (two * zero))            [half.if-one]
             --> (S zero)                    [Times.right-zero]
             ==> (Odd S zero)                [Odd-definition]])

conclude Even-S-S
  pick-any n
    let {right := assume (Even S S n)
                    (!chain->
                     [(S S (two * (half n)))
                  <-- (two * half S S n)          [half.twice]
                  --> (S S n)                     [Even-definition]
                  ==> ((S (two * half n)) = S n)  [S-injective]
                  ==> (two * (half n) = n)        [S-injective]
                  ==> (Even n)                    [Even-definition]]);
         left := assume (Even n)
                   (!chain->
                    [(two * half S S n)
                 --> (S S (two * half n))       [half.twice]
                 --> (S S n)                    [Even-definition]
                 ==> (Even S S n)               [Even-definition]])}
      (!equiv right left)

conclude Odd-S-S
  pick-any n
    let {right :=
           assume (Odd S S n)
             (!chain->
              [(S S S (two * half n))
               <-- (S (two * half S S n))        [half.twice]
               <-- (two * (half S S n) + one)    [Plus.right-one]
               --> (S S n)                       [Odd-definition]
               ==> (S S (two * half n) = S n)    [S-injective]
               ==> (S (two * half n) = n)        [S-injective]
               ==> (two * (half n) + one = n)    [Plus.right-one]
               ==> (Odd n)                       [Odd-definition]]);
         left :=
           assume (Odd n)
             (!chain->
              [((two * (half S S n)) + one)
               --> (S (two * half S S n))       [Plus.right-one]
               --> (S S S (two * half n))       [half.twice]
               <-- (S S (two * (half n) + one)) [Plus.right-one]
               --> (S S n)                      [Odd-definition]
               ==> (Odd S S n)                  [Odd-definition]])}
       (!equiv right left)

by-induction Odd-if-not-Even {
  zero => assume (~ Even zero)
            (!from-complements
             (Odd zero) Even-zero (~ Even zero))
| (S zero) =>
    assume (~ (Even (S zero)))
      (!chain->
       [((two * (half S zero)) + one)
        --> (S (two * half S zero))   [Plus.right-one]
        --> (S (two * zero))          [half.if-one]
```

```
        --> (S zero)                  [Times.right-zero]
        ==> (Odd S zero)              [Odd-definition]])
| (S (S x)) =>
    let {induction-hypothesis := (~ Even x ==> Odd x)}
      conclude (~ Even S S x ==> Odd S S x)
        assume hyp := (~ Even S S x)
          let {_ := (!by-contradiction (~ Even x)
                     (!chain [(Even x)
                          ==> (Even S S x)        [Even-S-S]
                          ==> (hyp & Even S S x)  [augment]
                          ==> false               [prop-taut]]))}
            (!chain-> [(~ Even x)
                       ==> (Odd x)         [induction-hypothesis]
                       ==> (Odd S S x)     [Odd-S-S]])
}

conclude not-Odd-if-Even
  pick-any x
    assume (Even x)
      (!by-contradiction (~ Odd x)
         assume (Odd x)
           let {equal :=
                 (!chain
                  [(S x)
                   <-- (S (two * half x))       [Even-definition]
                   <-- (two * (half x) + one)   [Plus.right-one]
                   --> x                        [Odd-definition]]);
                unequal :=
                 (!chain-> [true ==> (S x =/= x)  [S-not-same]])}
           (!absurd equal unequal))

conclude Even-iff-not-Odd
  pick-any x
    let {right := (!chain
                   [(Even x) ==> (~ Odd x)   [not-Odd-if-Even]]);
         left := assume (~ Odd x)
                   (!by-contradiction (Even x)
                    (!chain [(~ Even x)
                         ==> (Odd x)           [Odd-if-not-Even]
                         ==> (~ Odd x & Odd x) [augment]
                         ==> false             [prop-taut]]))}
      (!equiv right left)

conclude not-Even-if-Odd
  pick-any x
    assume (Odd x)
      (!by-contradiction (~ Even x)
        assume (Even x)
          let {not-odd := (!chain-> [(Even x) ==> (~ Odd x)
                                    [not-Odd-if-Even]])}
            (!absurd (Odd x) not-odd))

conclude half-nonzero-if-nonzero-Even
  pick-any n
    assume (n =/= zero & Even n)
      (!by-contradiction (half n =/= zero)
        assume opposite := (half n = zero)
          let {is := (!chain [n <-- (two * half n) [Even-definition]
                                --> (two * zero)   [opposite]
                                --> zero       [Times.right-zero]]);
               is-not := (n =/= zero)}
          (!absurd is is-not))

conclude half-nonzero-if-nonone-Odd
  pick-any n
    assume (n =/= one & Odd n)
      (!by-contradiction (half n =/= zero)
        assume opposite := (half n = zero)
          let {n-one := (!chain
                         [n <-- (two * (half n) + one)  [Odd-definition]
```

```
                              --> (two * zero + one)      [opposite]
                              --> (zero + one)            [Times.right-zero]
                              --> one                     [Plus.left-zero]])}
          (!absurd n-one (n =/= one)))

conclude Even-twice
  pick-any x
    (!chain-> [(two * half (two * x))
               --> (two * x)            [half.Times-two]
               ==> (Even (two * x))     [Even-definition]])

#.........................................................................
conclude Even-square
  pick-any x
    let {right :=
          assume (Even x)
           let {i := conclude (two * half square x = square x)
                      (!combine-equations
                       (!chain
                        [(two * half square x)
                     <-- (two * half square (two * half x))
                                         [Even-definition]
                     --> (two * half ((two * (half x)) *
                                       (two * (half x))))
                                         [square.definition]
                     --> (two * half two * ((half x) * (two * half x)))
                                         [Times.associative]
                     --> (two * ((half x) * (two * half x)))
                                         [half.Times-two]])
                       (!chain
                        [(square x)
                     <-- (square (two * half x))
                                         [Even-definition]
                     --> ((two * half x) * (two * half x))
                                         [square.definition]
                     --> (two * ((half x) * (two * half x)))
                                         [Times.associative]]))}
             (!chain-> [i ==> (Even square x) [Even-definition]]);
         left :=
          assume (Even square x)
            (!by-contradiction (Even x)
             assume hyp := (~ Even x)
               let {_ := (!chain-> [hyp ==> (Odd x) [Odd-if-not-Even]]);
                    A := conclude (two * (half square x) + one = square x)
                           let {i := conclude (square x =
                                                two * ((half x) * x) + x)
                                      (!chain
                                       [(square x)
                                    --> (x * x) [square.definition]
                                    <-- (((two * half x) + one) * x)
                                                [Odd-definition]
                                    --> ((two * half x) * x + one * x)
                                                [Times.right-distributive]
                                    --> (two * ((half x) * x) + x)
                                                [Times.associative
                                                 Times.left-one]]);
                                ii := conclude (half square x =
                                                 (half x) * x + half x)
                                        (!chain
                                         [(half square x)
                                      --> (half (two * ((half x) * x) + x))
                                                [i]
                                      --> ((half x) * x + half x)
                                                [half.two-plus]]);
                                iii := conclude
                                         (two * (half square x) + one =
                                          two * ((half x) * x) + x)
                                         (!chain
                                          [(two * (half square x) + one)
                                       --> (two * ((half x) * x + half x)
```

```
                                                            + one)    [ii]
                                             --> ((two * ((half x) * x) +
                                                  two * half x) + one)
                                                       [Times.left-distributive]
                                             --> (two * ((half x) * x) +
                                                  two * (half x) + one)
                                                       [Plus.associative]
                                             --> (two * ((half x) * x) + x)
                                                       [Odd-definition]])}
                                 (!combine-equations iii i)}
                     (!absurd
                      (!chain-> [A ==> (Odd square x) [Odd-definition]])
                      (!chain-> [(Even square x) ==> (~ Odd square x)
                                        [not-Odd-if-Even]]))))}
      (!equiv right left)
} # EO

#--------------------------------------------------------------------------

declare parity: [N] -> N
module parity {
assert even := (forall n . Even n ==> parity n = zero)
assert odd :=  (forall n . ~ Even n ==> parity n = one)

define half-case := (forall n . two * (half n) + parity n = n)
define plus-half := (forall n . n =/= zero ==> (half n) + parity n =/= zero)

conclude half-case
  pick-any n
    (!two-cases
      assume (Even n)
       (!chain [(two * (half n) + parity n)
                --> (two * (half n) + zero)     [even]
                --> (two * half n)              [Plus.right-zero]
                --> n                           [EO.Even-definition]])
      assume (~ (Even n))
       (!chain-> [(~ Even n)
                  ==> (Odd n)   [EO.Odd-if-not-Even]
                  ==> (two * (half n) + one = n) [EO.Odd-definition]
                  ==> (two * (half n) + parity n = n)        [odd]]))

conclude plus-half
  pick-any n
    assume A := (n =/= zero)
      (!two-cases
        assume B := (Even n)
          let {C := (!chain
                     [((half n) + parity n)
                      = ((half n) + zero)    [even]
                      = (half n)             [Plus.right-zero]])}
          (!chain-> [(A & B)
                     ==> (half n =/= zero)
                            [EO.half-nonzero-if-nonzero-Even]
                     ==> ((half n) + parity n =/= zero) [C]])
        assume (~ Even n)
         let {C := (!chain
                    [((half n) + parity n)
                     = ((half n) + S zero)    [odd one-definition]
                     = (S ((half n) + zero)) [Plus.right-nonzero]])}
         (!chain-> [true ==> (S ((half n) + zero) =/= zero)
                                                   [S-not-zero]
                         ==> ((half n) + parity n =/= zero) [C]]))
} # parity
} # N
```