# lib/main/nat-half.ath

```
load "nat-times"
load "nat-less"

extend-module N {

declare half: [N] -> N [[int->nat]]

module half {

    assert* axioms :=
      [(half zero = zero)
        (half S zero = zero)
        (half S S n = S half n)]

define [if-zero if-one nonzero-nonone] := axioms

(print "\nHalf of 20: " (eval half 20) "and half of 21: " (eval half 21) "\n")

define double := (forall n . half (n + n) = n)

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]])
}

define Times-two := (forall x . half (two * x) = x)

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

define twice := (forall x . two * half S S x = S S (two * half x))

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]])

define two-plus := (forall x y . half (two * x + y) = x + half y)

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]])
}

define less-S := (forall n . half n < S n)
define less := (forall n . n =/= zero ==> half n < n)

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]])
| (n as (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]])
| (n as (S (S m))) =>
    assume (S S m =/= zero)
      (!chain-> [true
             ==> (half m < S m)          [less-S]
             ==> (S half m < S S m)      [Less.injective]
             ==> (half S S m < S S m)    [nonzero-nonone]])
}

define equal-zero :=
  (forall x . half x = zero ==> x = zero | x = one)

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)
}

define less-equal := (forall n . half n <= n)

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-<]])
}

define less-equal-1 := (forall n . n =/= zero ==> S half n <= n)

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]])
}



} # close module half
```

```
declare even, odd: [N] -> Boolean [[int->nat]]
module EO {

   assert* even-definition := [(even x <==> two * half x = x)]

   assert* odd-definition :=  [(odd  x <==> two * (half x) + one = x)]


   (print "\nis 20 even?: " (eval even 20))
   (print "\nis 20 odd?: " (eval odd 20))
   (print "\nis 21 even?: " (eval even 21))
   (print "\nis 21 odd?: " (eval odd 21))

#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)

(!force even-zero)
(!force not-odd-if-even)

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

declare square: [N] -> N  [[int->nat]]
module square {
  assert* definition := [(square x = x * x)]

    (print "\nsquare of 12: " (eval square 12) "\n")
} # close module square


define even-square := (forall x . even x <==> even square x)

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)

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)))


} # close EO

declare parity: [N] -> N

module parity {
assert if-even := (forall n . even n ==> parity n = zero)
assert if-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)    [if-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)       [if-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)   [if-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)   [if-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
```