lib/main/nat-times.ath

```
# Properties of natural number multiplication operator, Times.

load "nat-plus"

#
# Multiplication operator, Times
#

extend-module N {

declare *: [N N] -> N [300]

module Times {

open Plus

# Axioms

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

assert right-zero    := (forall x . x * zero = zero)
assert right-nonzero := (forall x y . x * (S y) = x * y + x)

# Theorems:

define left-zero    := (forall x . zero * x = zero)
define left-nonzero := (forall x y . (S y) * x = x + y * x)

by-induction left-zero {
  zero =>
   (!chain [(zero * zero) = zero  [right-zero]])
| (S x) =>
  let {induction-hypothesis := (zero * x = zero)}
  (!chain [(zero * (S x))
          = (zero * x + zero)    [right-nonzero]
          = (zero + zero)        [induction-hypothesis]
          = zero                 [Plus.right-zero]])
}

by-induction left-nonzero {
  zero =>
    pick-any y
      (!combine-equations
       (!chain [((S y) * zero) = zero  [right-zero]])
       (!chain [(zero + y * zero)
             = (zero + zero)     [right-zero]
             = zero              [Plus.right-zero]]))
| (S x) =>
    pick-any y
      let {induction-hypothesis := (forall ?y . (S ?y) * x = x + ?y * x)}
      (!combine-equations
       (!chain
        [((S y) * (S x))
         --> ((S y) * x + (S y))   [right-nonzero]
         --> ((x + y * x) + (S y)) [induction-hypothesis]
         --> (S ((x + y * x) + y)) [Plus.right-nonzero]
         --> (S (x + (y * x + y))) [Plus.associative]])
       (!chain
        [((S x) + y * (S x))
         --> ((S x) + (y * x + y)) [right-nonzero]
         --> (S (x + (y * x + y))) [Plus.left-nonzero]]))
}

define right-one := (forall x . x * one = x)
define left-one  :=  (forall x . one * x = x)

conclude right-one
  pick-any x
```

```
    (!chain [(x * one)
             --> (x * (S zero))   [one-definition]
             --> (x * zero + x)   [right-nonzero]
             --> (zero + x)       [right-zero]
             --> x                [Plus.left-zero]])

conclude left-one
  pick-any x
    (!chain [(one * x)
             --> ((S zero) * x)    [one-definition]
             --> (x + zero * x)    [left-nonzero]
             --> (x + zero)        [left-zero]
             --> x                 [Plus.right-zero]])

define right-distributive :=
  (forall x y z . (x + y) * z = x * z + y * z)
define left-distributive :=
   (forall z x y . z * (x + y) = z * x + z * y)

by-induction right-distributive {
  zero =>
    pick-any y z
      (!combine-equations
       (!chain [((zero + y) * z) = (y * z)   [Plus.left-zero]])
       (!chain [(zero * z + y * z)
                --> (zero + y * z)           [left-zero]
                --> (y * z)                  [Plus.left-zero]]))
| (S x) =>
    let {induction-hypothesis :=
           (forall ?y ?z . (x + ?y) * ?z = x * ?z + ?y * ?z)}
    pick-any y z
    (!combine-equations
     (!chain
      [(((S x) + y) * z)
       --> ((S (x + y)) * z)             [Plus.left-nonzero]
       --> (z + ((x + y) * z))           [left-nonzero]
       --> (z + (x * z + y * z))         [induction-hypothesis]])
     (!chain
      [((S x) * z + y * z)
       --> ((z + x * z) + y * z)   [left-nonzero]
       --> (z + (x * z + y * z))   [Plus.associative]]))
}

# Associativity and commutativity:

define associative := (forall x y z . (x * y) * z = x * (y * z))
define commutative := (forall x y . x * y = y * x)

by-induction associative {
  zero =>
    pick-any y z
      (!chain [((zero * y) * z)
               --> (zero * z)       [left-zero]
               --> zero             [left-zero]
               <-- (zero * (y * z)) [left-zero]])
| (S x) =>
  let {induction-hypothesis :=
          (forall ?y ?z . (x * ?y) * ?z = x * (?y * ?z))}
  pick-any y z
    (!chain
     [(((S x) * y) * z)
      --> ((y + (x * y)) * z)     [left-nonzero]
      --> (y * z + (x * y) * z)   [right-distributive]
      --> (y * z + (x * (y * z))) [induction-hypothesis]
      <-- ((S x) * (y * z))       [left-nonzero]])
}

by-induction commutative {
  zero =>
    conclude (forall ?y . zero * ?y = ?y * zero)
```

```
      pick-any y
        (!chain [(zero * y)
                 --> zero        [left-zero]
                 <-- (y * zero)  [right-zero]])
| (S x) =>
    let {induction-hypothesis := (forall ?y . (x * ?y = ?y * x))}
    conclude (forall ?y . (S x) * ?y = ?y * (S x))
      pick-any y
        (!combine-equations
          (!chain [((S x) * y)
                   --> (y + x * y)  [left-nonzero]
                   --> (y + y * x)  [induction-hypothesis]])
          (!chain [(y * (S x))
                   --> (y * x + y)  [right-nonzero]
                   --> (y + y * x)  [Plus.commutative]]))
}

conclude left-distributive
  pick-any z x y
    (!chain [(z * (x + y))
             --> ((x + y) * z)      [commutative]
             --> (x * z + y * z)    [right-distributive]
             --> (z * x + z * y)    [commutative]])

define no-zero-divisors :=
  (forall x y . x * y = zero ==> x = zero | y = zero)

conclude no-zero-divisors
  pick-any x y
    assume (x * y = zero)
      (!two-cases
        assume (x = zero)
          (!left-either (x = zero) (y = zero))
        assume A1 := (x =/= zero)
          let {C1 := (!chain->
                      [A1 ==> (exists ?u . x = (S ?u))
                                          [nonzero-S]])}
           pick-witness u for C1
            let {C3 :=
              (!by-contradiction (y = zero)
               assume A2 := (y =/= zero)
                 let {C2 := (!chain->
                              [A2 ==> (exists ?v . y = (S ?v))
                                          [nonzero-S]])}
                  pick-witness v for C2
                    let {equal := (zero = (S ((S u) * v + u)))}
                      (!absurd
                        conclude equal
                          (!chain
                           [zero
                            <-- (x * y)              [(x * y = zero)]
                            --> ((S u) * (S v))      [(x = (S u)) (y = (S v))]
                            --> ((S u) * v + (S u))  [right-nonzero]
                            --> (S ((S u) * v + u))  [Plus.right-nonzero]])
                        conclude (~ equal)
                          (!chain->
                           [true ==> ((S ((S u) * v + u)) =/= zero)
                                                     [S-not-zero]
                                 ==> (~ equal)   [sym]])))}
                (!right-either (x = zero) C3))

# Alternative proof using datatype-cases:

datatype-cases no-zero-divisors {
  zero =>
    conclude (forall ?y . zero * ?y = zero ==> zero = zero | ?y = zero)
      pick-any y
        assume (zero * y = zero)
          (!left-either (!reflex zero) (y = zero))
| (S x) =>
```

```
  datatype-cases (forall ?y . (S x) * ?y = zero ==> (S x) = zero | ?y = zero)
  {
    zero =>
      conclude ((S x) * zero = zero ==> (S x) = zero | zero = zero)
        assume ((S x) * zero = zero)
          (!right-either ((S x) = zero) (!reflex zero))
  | (S y) =>
      conclude ((S x) * (S y) = zero ==> (S x) = zero | (S y) = zero)
        assume is-zero := ((S x) * (S y) = zero)
          let {C :=
                conclude ((S x) * (S y) = (S ((S x) * y + x)))
                  (!chain [((S x) * (S y))
                           --> ((S x) * y + (S x)) [right-nonzero]
                           --> (S ((S x) * y + x)) [Plus.right-nonzero]]);
               is-not :=
                 (!chain->
                  [true ==> ((S ((S x) * y + x)) =/= zero)[S-not-zero]
                        ==> ((S x) * (S y) =/= zero)      [C]])}
          (!from-complements ((S x) = zero | (S y) = zero) is-zero is-not)
  }
}

define two-times := (forall x . two * x = x + x)

conclude two-times
  pick-any x
    (!chain [(two * x)
             --> ((S one) * x)   [two-definition]
             --> (x + one * x)   [left-nonzero]
             --> (x + x)         [left-one]])

} # Times

#####################################################
# square function:

declare square: [N] -> N
module square {
define x := ?x:N

assert def := (forall x . square x = x * x)

define zero-property := (forall x . square x = zero ==> x = zero)

conclude zero-property
  pick-any x
    assume A := ((square x) = zero)
      conclude (x = zero)
        (!chain-> [(x * x)
                   <-- (square x)    [def]
                   --> zero          [A]
           ==> (x = zero | x = zero) [Times.no-zero-divisors]
           ==> (x = zero)            [prop-taut]])

} # square

####################################################################

} # N
```