# lib/main/nat-power0.ath

```
# Properties of natural number exponentiation operator, Power.

load "nat-times"

#
# Exponentiation operator, **
#

extend-module N {
open Times

declare **: [N N] -> N [400]

module Power {

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

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

define Plus-case := (forall m n x . x ** (m + n) = x ** m * x ** n)
define left-one := (forall n . one ** n = one)
define right-one := (forall x . x ** one = x)
define right-two := (forall x . x ** two = x * x)
define left-times :=
  (forall n x y . (x * y) ** n = x ** n * y ** n)
define right-times :=
  (forall m n x . x ** (m * n) = (x ** m) ** n)
define two-case := (forall x . square x = x ** two)

by-induction Plus-case {
  zero =>
    conclude (forall ?n ?x . x ** (zero + ?n) = ?x ** zero * ?x ** ?n)
      pick-any n x
        (!chain [(x ** (zero + n))
                 --> (x ** n)              [Plus.left-zero]
                 <-- (one * x ** n)        [Times.left-one]
                 <-- (x ** zero * x ** n)  [if-zero]])
| (S m) =>
  let {induction-hypothesis :=
           (forall ?n ?x . ?x ** (m + ?n) = ?x ** m * ?x ** ?n)}
    conclude (forall ?n ?x .
                 ?x ** ((S m) + ?n) = ?x ** (S m) * ?x ** ?n)
      pick-any n x
        (!combine-equations
         (!chain
          [(x ** ((S m) + n))
           --> (x ** (S (m + n)))       [Plus.left-nonzero]
           --> (x * x ** (m + n))       [if-nonzero]
           --> (x * (x ** m * x ** n)) [induction-hypothesis]])
         (!chain
          [(x ** (S m) * x ** n)
           --> ((x * (x ** m)) * x ** n) [if-nonzero]
           --> (x * (x ** m * x ** n))   [Times.associative]]))
}

by-induction left-times {
  zero =>
   conclude (forall ?x ?y . (?x * ?y) ** zero = ?x ** zero * ?y ** zero)
     pick-any x y
       (!chain [((x * y) ** zero)
                --> one                        [if-zero]
                <-- (one * one)                [Times.right-one]
                <-- (x ** zero * y ** zero)    [if-zero]])
| (S n) =>
```

```
    conclude (forall ?x ?y .
                (?x * ?y) ** (S n) = ?x ** (S n) * ?y ** (S n))
      let {induction-hypothesis :=
             (forall ?x ?y . (?x * ?y) ** n = ?x ** n * ?y ** n)}
      pick-any x y
        (!combine-equations
         (!chain
          [((x * y) ** (S n))
           --> ((x * y) * (x * y) ** n)       [if-nonzero]
           --> ((x * y) * (x ** n * y ** n)) [induction-hypothesis]
           --> (x * y * x ** n * y ** n)      [Times.associative]])
         (!chain
          [(x ** (S n) * y ** (S n))
           --> ((x * (x ** n)) * y * y ** n) [if-nonzero]
           --> (x * x ** n * y * y ** n)      [Times.associative]
           <-- (x * (x ** n * y) * y ** n)    [Times.associative]
           <-- (x * (y * x ** n) * y ** n)    [Times.commutative]
           --> (x * y * x ** n * y ** n)    [Times.associative]]))
}

by-induction left-one {
    zero => (!chain [(one ** zero) --> one  [if-zero]])
| (S n) =>
    let {induction-hypothesis := (one ** n = one)}
      (!chain [(one ** (S n))
               --> (one * (one ** n)) [if-nonzero]
               --> (one ** n)          [Times.left-one]
               --> one                 [induction-hypothesis]])
}

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

conclude right-two
  pick-any x
    (!chain [(x ** two)
             --> (x ** (S one))    [two-definition]
             --> (x * x ** one)    [if-nonzero]
             --> (x * x)           [right-one]])

by-induction right-times {
  zero =>
    conclude (forall ?n ?x . ?x ** (zero * ?n) = (?x ** zero) ** ?n)
      pick-any n x
        (!chain [(x ** (zero * n))
                 --> (x ** zero)        [Times.left-zero]
                 --> one                [if-zero]
                 <-- (one ** n)         [left-one]
                 <-- ((x ** zero) ** n) [if-zero]])
| (S m) =>
    let {induction-hypothesis :=
           (forall ?n ?x . ?x ** (m * ?n) = (?x ** m) ** ?n)}
      conclude (forall ?n ?x . ?x ** ((S m) * ?n) = (?x ** (S m)) ** ?n)
        pick-any n x
          (!combine-equations
           (!chain [(x ** ((S m) * n))
                    --> (x ** (n + m * n))          [Times.left-nonzero]
                    --> (x ** n * x ** (m * n))     [Plus-case]
                    --> (x ** n * ((x ** m) ** n)) [induction-hypothesis]])
           (!chain [((x ** (S m)) ** n)
                    --> ((x * (x ** m)) ** n)       [if-nonzero]
                    --> ((x ** n) * (x ** m) ** n) [left-times]]))
}

conclude two-case
```

```
  pick-any x
    (!combine-equations
     (!chain
      [(square x) = (x * x)  [square.definition]])
     (!chain
      [(x ** two)
       = (x ** (S one))       [two-definition]
       = (x ** (S (S zero))) [one-definition]
       = (x * x ** (S zero)) [if-nonzero]
       = (x * x * x ** zero) [if-nonzero]
       = (x * x * one)       [if-zero]
       = (x * x)             [Times.right-one]]))

} # Power
} # N
```