lib/main/power.ath

```
#   Abstract Power function

#------------------------------------------------------------------------
load "nat-times"
load "group"
#------------------------------------------------------------------------

extend-module Monoid {

  declare ++*: (S) [S N] -> S [400]

  define [x y m n] := [?x:'T ?y:'T ?m:N ?n:N]

  module Power {

    define right-zero :=  (forall x . x +* zero = <0>)
    define right-nonzero := (forall n x . x ++ (S n) = x + x ++ n)

    (add-axioms theory [right-zero right-nonzero])

  define [+' *'] := [N.+ N.*]

  define right-plus :=
    (forall m n x . x ++ (m +' n) = x +* m + x ++ n)
  define left-neutral := (forall n . <0> ++ n = <0>)
  define right-one := (forall x . x ++ N.one = x)
  define right-two := (forall x . x ++ N.two = x + x)
  define right-times :=
    (forall n m x . x ++ (m *' n) = (x ++ m) ++ n)

  define theorems :=
    [right-plus left-neutral right-one right-two right-times]

define proofs :=
 method (theorem adapt)
  let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
        [+ <0> ++] := (adapt [+ <0> ++])}
   match theorem {
    (val-of right-plus) =>
    by-induction (adapt theorem) {
      zero =>
        conclude
             (forall ?n ?x . ?x ++ (zero +' ?n) = ?x ++ zero + ?x ++ ?n)
          pick-any n x
            (!chain
             [(x ++ (zero +' n))
              --> (x ++ n)                          [N.Plus.left-zero]
              <-- (<0> + (x ++ n))                  [left-identity]
              <-- ((x ++ zero) + (x ++ n))          [right-zero]])
      | (S m) =>
          let {ind-hyp :=
                 (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)))           [N.Plus.left-nonzero]
                 --> (x + (x ++ (m +' n)))         [right-nonzero]
                 --> (x + ((x ++ m) + (x ++ n)))   [ind-hyp]])
               (!chain
                [((x ++ (S m)) + (x ++ n))
                 --> ((x + (x ++ m)) + (x ++ n))   [right-nonzero]
                 --> (x + ((x ++ m) + (x ++ n)))   [associative]]))
```

```
      }
  | (val-of left-neutral) =>
     by-induction (adapt theorem) {
       zero => (!chain [(<0> ** zero) = <0>      [right-zero]])
     | (S n) =>
         let {ind-hyp := (<0> ** n = <0>)}
           conclude (<0> ** (S n) = <0>)
             (!chain [(<0> ** (S n))
                      = (<0> + (<0> ** n))        [right-nonzero]
                      = (<0> + <0>)               [ind-hyp]
                      = <0>                       [right-identity]])
      }
  | (val-of right-one) =>
      pick-any x:(sort-of <0>)
        (!chain [(x ** N.one)
                 --> (x ** (S zero))              [N.one-definition]
                 --> (x + x ** zero)              [right-nonzero]
                 --> (x + <0>)                    [right-zero]
                 --> x                            [right-identity]])
  | (val-of right-two) =>
      let {right-one := (!prove right-one)}
        pick-any x
          (!chain [(x ** N.two)
                   = (x ** (S N.one))             [N.two-definition]
                   = (x + x ** N.one)             [right-nonzero]
                   = (x + x)                      [right-one]])
  | (val-of right-times) =>
      by-induction (adapt theorem) {
        zero =>
          conclude
            (forall ?m ?x . ?x ** (?m *' zero) = (?x ** ?m) ** zero)
            pick-any m x
              (!combine-equations
               (!chain [(x ** (m *' zero))
                        = (x ** zero)         [N.Times.right-zero]
                        = <0>                 [right-zero]])
               (!chain [((x ** m) ** zero)
                        = <0>                 [right-zero]]))
      | (S n) =>
          let {ind-hyp := (forall ?m ?x .
                             ?x ** (?m *' n) = (?x ** ?m) ** n);
               _ := (!prove right-plus);
               _ := (!prove right-one)}
          conclude
            (forall ?m ?x .
               ?x ** (?m *' (S n)) = (?x ** ?m) ** (S n))
            pick-any m x
              (!combine-equations
               (!chain [(x ** (m *' (S n)))
                        = (x ** (m *' n +' m))  [N.Times.right-nonzero]
                        = (x ** (m *' n) + (x ** m))      [right-plus]
                        = ((x ** m) ** n + (x ** m))      [ind-hyp]])
               (!chain
                [((x ** m) ** (S n))
               = ((x ** m) + (x ** m) ** n)               [right-nonzero]
               = ((x ** m) ** N.one + (x ** m) ** n)      [right-one]
               = ((x ** m) ** (N.one +' n))               [right-plus]
               = ((x ** m) ** (n +' N.one))       [N.Plus.commutative]
               = ((x ** m) ** n + (x ** m) ** N.one)      [right-plus]
               = ((x ** m) ** n + x ** m)                 [right-one]]))
      } # by-induction
  } # match

(add-theorems theory |{theorems := proofs}|)
} # Power
} # Monoid

#--------------------------------------------------------------------------
# The following theorem requires commutativity of +.

```

```
extend-module Abelian-Monoid {

  define +* := Monoid.+*

  define right-zero := Monoid.Power.right-zero

  define right-nonzero := Monoid.Power.right-nonzero

  define [x y n] := [?x:'T ?y:'T ?n:N]

  define Power-left-times := (forall n x y . (x + y) +* n = x +* n + y +* n)

define proof :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [+ <0> +*] := (adapt [+ <0> +*])}
    match theorem {
      (val-of Power-left-times) =>
      by-induction (adapt theorem) {
        zero =>
          pick-any x y
            (!combine-equations
             (!chain [((x + y) +* zero)
                      = <0>                   [right-zero]])
             (!chain [(x +* zero + y +* zero)
                      = (<0> + <0>)           [right-zero]
                      = <0>                   [right-identity]]))
      | (S n) =>
          let {ind-hyp := (forall ?x ?y . (?x + ?y) +* n =
                                          ?x +* n + ?y +* n)}
            conclude  (forall ?x ?y . (?x + ?y) +* (S n) =
                                      ?x +* (S n) + ?y +* (S n))
              pick-any x y
                (!chain
                 [((x + y) +* (S n))
                  = ((x + y) + (x + y) +* n)          [right-nonzero]
                  = ((x + y) + (x +* n) + (y +* n))   [ind-hyp]
                  = ((x + (x +* n)) + (y + (y +* n))) [associative
                                                       commutative]
                  = (x +* (S n) + y +* (S n))         [right-nonzero]])
      }
   }

(add-theorems theory |{[Power-left-times] := proof}|)
} # Abelian-Monoid
```