lib/main/nat-div.ath

```
load "nat-minus"
load "strong-induction"

#..........................................................................
extend-module N {

declare /, %: [N N] -> N [300 [int->nat int->nat]]

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

module Div {
assert basis := (forall x y . x < y ==> x / y = zero)
assert reduction :=
  (forall x y . ~ x < y & zero < y ==> x / y = S ((x - y) / y))
} # close module Div

module Mod {
assert basis := (forall x y . x < y ==> x % y = x)
assert reduction :=
  (forall x y . ~ x < y & zero < y ==> x % y = (x - y) % y)
} # close module Mod

extend-module Div {

define cancellation :=  (forall x y . zero < y ==> (x * y) / y = x)

by-induction cancellation {
  zero => pick-any y
            assume (zero < y)
              (!chain [((zero * y) / y)
                       = (zero / y)      [Times.left-zero]
                       = zero            [basis (zero < y)]])
| (S x) =>
    pick-any y
      let {ind-hyp := (forall ?y . zero < ?y ==> (x * ?y) / ?y = x)}
        assume (zero < y)
          let {B := conclude (~ x * y + y < y)
                      (!chain-> [(~ x * y + y < y)
                             <== (y <= x * y + y)   [Less=.trichotomy3]
                             <== (y <= y + x * y)   [Less=.Plus-commutative]
                             <== (y <= y)           [Less=.Plus-k1]
                             <== true               [Less=.reflexive]])}
            conclude ((((S x) * y) / y) = (S x))
             (!chain [(((S x) * y) / y)
                    = ((y + x * y) / y)            [Times.left-nonzero]
                    = ((x * y + y) / y)            [Plus.commutative]
                    = (S (((x * y + y) - y) / y))  [reduction B]
                    = (S ((x * y) / y))            [Plus.commutative
                                                    Minus.cancellation]
                    = (S x)                        [ind-hyp]])
}
} # close module Div

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

define division-algorithm :=
  (forall x y . zero < y ==> (x / y) * y + x % y = x & x % y < y)

conclude goal := division-algorithm
 (!strong-induction.principle goal
   method (x)
    assume IND-HYP := (strong-induction.hypothesis goal x)
      conclude (strong-induction.conclusion goal x)
       pick-any y
         assume (zero < y)
           conclude ((x / y) * y + x % y = x & x % y < y)
             (!two-cases
               assume (x < y)
```

```
                 let {C1 :=
                        (!chain->
                         [(x < y) ==> (x / y = zero) [Div.basis]]);
                      C2 :=
                        (!chain->
                         [(x < y) ==> (x % y = x)    [Mod.basis]]);
                      C3 :=
                        (!chain
                         [((x / y) * y + (x % y))
                          = (zero * y + x)   [C1 C2]
                          = x [Times.left-zero Plus.left-zero]]);
                      C4 := (!chain->
                             [(x < y) ==> (x % y < y) [C2]])}
                   (!both C3 C4)
               assume (~ x < y)
                 let {C1 :=
                        (!chain->
                         [(~ x < y & zero < y)
                          ==> (x / y = (S ((x - y) / y)))
                                                         [Div.reduction]]);
                      C2 :=
                        (!chain->
                         [(~ x < y & zero < y)
                          ==> (x % y = (x - y) % y)
                                                         [Mod.reduction]]);
                      C3 := (!chain->
                             [(~ x < y) ==> (y <= x)
                                                         [Less=.trichotomy2]]);
                      C4 :=
                        (!chain->
                         [(zero < y & y <= x)
                          ==> (x - y < x)                [Minus.<-left]
                          ==> (forall ?v . zero < ?v ==>
                                (((x - y) / ?v) * ?v + (x - y) % ?v
                                 = x - y &
                                 (x - y) % ?v < ?v))     [IND-HYP]]);
                      C5 :=
                        (!chain->
                         [(zero < y)
                           ==> (((x - y) / y) * y + (x - y) % y = x - y
                                & (x - y) % y < y)          [C4]]);
                      C5a := (!left-and C5);
                      C5b := (!right-and C5);
                      C6 :=
                        (!chain
                         [((x / y) * y + x % y)
                          = ((S ((x - y) / y)) * y + (x - y) % y)
                                                  [C1 C2]
                          = ((y + ((x - y) / y) * y) + (x - y) % y)
                                                  [Times.left-nonzero]
                          = (y + (((x - y) / y) * y + (x - y) % y))
                                                  [Plus.associative]
                          = (y + (x - y))         [C5a]
                          = ((x - y) + y)         [Plus.commutative]
                          = x                     [C3 Minus.Plus-Cancel]])}
                    (!chain->
                     [C5b ==> (x % y < y)         [C2]
                          ==> (C6 & (x % y < y)) [augment]]))))

define division-algorithm-corollary1 :=
   (forall x y . zero < y ==> (x / y) * y + x % y = x)
define division-algorithm-corollary2 :=
   (forall x y . zero < y ==> x % y < y)

conclude corollary := division-algorithm-corollary1
  let {theorem := division-algorithm}
    (!mp (!taut (theorem ==> corollary))
         theorem)

conclude corollary := division-algorithm-corollary2
```

```
  let {theorem := division-algorithm}
    (!mp (!taut (theorem ==> corollary))
         theorem)

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

declare divides: [N N] -> Boolean  [300 [int->nat int->nat]]

module divides {

assert left-positive :=
  (forall x y . zero < y ==> y divides x <==> x % y = zero)
assert left-zero :=
  (forall x y . y = zero ==> y divides x <==> x = zero)

define characterization :=
  (forall x y . y divides x <==> exists z . y * z = x)

conclude characterization
  pick-any x y
   (!two-cases
    assume (zero < y)
       (!equiv
        assume A := (y divides x)
          let {B := (!chain-> [A ==> (x % y = zero)  [left-positive]])}
            (!chain-> [(zero < y)
                   ==> ((x / y) * y + x % y = x)
                                          [division-algorithm-corollary1]
                   ==> ((x / y) * y + zero = x)      [B]
                   ==> ((x / y) * y = x)             [Plus.right-zero]
                   ==> (y * (x / y) = x)             [Times.commutative]
                   ==> (exists ?z . y * ?z = x)      [existence]])
        assume A := (exists ?z . y * ?z = x)
          pick-witness z for A A-w
            (!by-contradiction (y divides x)
             assume B := (~ y divides x)
               let {C := (!chain-> [(zero < y)
                                ==> (y divides x <==> x % y = zero)
                                                 [left-positive]])}
                (!absurd
                 (!chain->
                  [B ==> (x % y =/= zero) [C]
                     ==> (zero < x % y)   [Less.zero<]
                     ==> (zero + (x / y) * y < x % y + (x / y) * y)
                                            [Less.Plus-k]
                     ==> ((x / y) * y < (x / y) * y + x % y)
                                        [Plus.left-zero Plus.commutative]
                     ==> ((x / y) * y < x)
                                        [division-algorithm-corollary1]
                     ==> (y * (x / y) < y * z)  [A-w Times.commutative]
                     ==> (x / y < z)            [Times.<-cancellation]
                     ==> ((y * z) / y < z)      [A-w]
                     ==> (z < z)   [Times.commutative Div.cancellation]])
                 (!chain-> [true ==> (~ z < z)  [Less.irreflexive]])))))
    assume (~ zero < y)
      let {C := (!chain-> [(~ zero < y) ==> (y = zero)  [Less.=zero]])}
       (!equiv
        assume A := (y divides x)
          (!chain-> [A ==> (x = zero)                [left-zero]
                       ==> (zero = x)                [sym]
                       ==> (y * zero = x)            [Times.right-zero]
                       ==> (exists ?z . y * ?z = x)  [existence]])
        assume A := (exists ?z . y * ?z = x)
          let {B := (!chain->
                     [C ==> (y divides x <==> x = zero) [left-zero]])}
          pick-witness z for A A-w
            (!chain-> [x = (y * z)             [A-w]
                         = (zero * z)          [C]
                         = zero                [Times.left-zero]
                         ==> (y divides x)     [B]])))
```

```
define elim :=
  method (x y)
    let {v := (fresh-var (sort-of x))}
      (!chain->
       [(divides x y) ==> (exists v . x * v = y) [characterization]])

define reflexive := (forall x . x divides x)
define right-zero := (forall x . x divides zero)
define left-zero := (forall x . zero divides x <==> x = zero)

conclude reflexive
  pick-any x
    (!chain->
     [true ==> (x * one = x)              [Times.right-one]
           ==> (exists ?y . x * ?y = x)   [existence]
           ==> (x divides x)              [characterization]])

conclude right-zero
  pick-any x
    (!chain->
     [true ==> (x * zero = zero)            [Times.right-zero]
           ==> (exists ?y . x * ?y = zero) [existence]
           ==> (x divides zero)             [characterization]])

conclude left-zero
  pick-any x
   let {right := conclude (zero divides x ==> x = zero)
                   assume (zero divides x)
                     let {C1 := (!elim zero x)}
                       pick-witness y for C1 C1-w
                         (!chain
                          [x = (zero * y)   [C1-w]
                             = zero         [Times.left-zero]]);
        left := conclude (x = zero ==> zero divides x)
                  assume (x = zero)
                    (!chain->
                     [true ==> (zero * zero = zero)   [Times.left-zero]
                           ==> (exists ?y . zero * ?y = zero)   [existence]
                           ==> (zero divides zero)    [characterization]
                           ==> (zero divides x)       [(x = zero)]])}
     (!equiv right left)

#..................................................................
define sum-lemma1 :=
  (forall x y z . x divides y & x divides z ==> x divides (y + z))
define sum-lemma2 :=
  (forall x y z . x divides y & x divides (y + z) ==> x divides z)
define sum :=
  (forall x y z . x divides y & x divides z
                <==> x divides y & x divides (y + z))

conclude sum-lemma1
  pick-any x y z
    assume (x divides y & x divides z)
      pick-witness u for (!elim x y)
        pick-witness v for (!elim x z)
          let {witnessed1 := (x * u = y);
               witnessed2 := (x * v = z)}
          conclude goal := (x divides (y + z))
            (!chain->
             [(x * (u + v))
              = (x * u + x * v)  [Times.left-distributive]
              = (y + z)          [witnessed1 witnessed2]
              ==> (exists ?w . x * ?w = y + z)   [existence]
              ==> goal           [characterization]])


conclude sum-lemma2
  pick-any x y z
```

```
    assume (x divides y & x divides (y + z))
      pick-witness u for (!elim x y)
        pick-witness v for (!elim x (y + z))
          conclude goal := (x divides z)
            let {w1 := (x * u = y);
                 w2 := (x * v = y + z)}
            (!chain->
             [(x * (v - u))
              = (x * v - x * u)  [Minus.Times-Distributivity]
              = ((y + z) - y)    [w1 w2]
              = z                [Minus.cancellation]
              ==> (exists ?w . x * ?w = z)  [existence]
              ==> goal                      [characterization]])

conclude sum
  pick-any x y z
    let {right := assume A := (x divides y & x divides z)
                    (!chain->
                     [A ==> (x divides (y + z))        [sum-lemma1]
                        ==> (x divides y & x divides (y + z))
                                                       [augment]]);
         left := assume A := (x divides y & x divides (y + z))
                     (!chain->
                      [A ==> (x divides z)               [sum-lemma2]
                         ==> (x divides y & x divides z) [augment]])}
       (!equiv right left)

#.........................................................................
define product-lemma :=
  (forall x y z . x divides y | x divides z ==> x divides y * z)
define product-left-lemma :=
  (forall x y z . x divides y ==> x divides y * z)

conclude product-left-lemma
  pick-any x y z
    assume A := (x divides y)
      pick-witness u for (!elim x y) witnessed
        (!chain->
         [(y * z) = ((x * u) * z)           [witnessed]
                  = (x * (u * z))           [Times.associative]
                  ==> (x * (u * z) = y * z) [sym]
                  ==> (exists ?v . x * ?v = y * z) [existence]
                  ==> (x divides y * z)     [characterization]])

conclude product-lemma
  pick-any x y z
    assume A := (x divides y | x divides z)
      conclude goal := (x divides y * z)
        (!cases A
          assume A1 := (x divides y)
            (!chain-> [A1 ==> goal  [product-left-lemma]])
          assume A2 := (x divides z)
            (!chain->
             [A2 ==> (x divides z * y)  [product-left-lemma]
                 ==> goal               [Times.commutative]]))

#.........................................................................
define first-lemma :=
  (forall x y z .
    zero < y & z divides y & z divides x % y ==> z divides x)

conclude first-lemma
  pick-any x y z
    assume A := (zero < y & z divides y & z divides x % y)
      conclude goal := (z divides x)
        pick-witness u for (!elim z y) witnessed1
          pick-witness v for (!elim z (x % y)) witnessed2
            (!chain->
             [x = ((x / y) * y + x % y)
                        [(zero < y) division-algorithm-corollary1]
```

```
                = ((x / y) * (z * u) + z * v)
                           [witnessed1 witnessed2]
                = (((x / y) * u) * z + v * z) [Times.commutative
                                                 Times.associative]
                = (((x / y) * u + v) * z) [Times.right-distributive]
                = (z * ((x / y) * u + v)) [Times.commutative]
                ==> (z * ((x / y) * u + v) = x) [sym]
                ==> (exists ?w . z * ?w = x)      [existence]
                ==> goal                          [characterization]])

#......................................................................
define antisymmetric :=
  (forall x y . x divides y & y divides x ==> x = y)

conclude antisymmetric
  pick-any x y
    assume (x divides y & y divides x)
      pick-witness u for (!elim x y)
       pick-witness v for (!elim y x)
        let {witnessed1 := (x * u = y);
             witnessed2 := (y * v = x)}
         (!two-cases
          assume A1 := (x = zero)
            (!chain->
             [witnessed1 ==> (zero * u = y)  [A1]
                         ==> (zero = y)      [Times.left-zero]
                         ==> (x = y)         [A1]])
          assume A2 := (x =/= zero)
            let {C1 := (!chain-> [A2 ==> (zero < x) [Less.zero<]]);
                 C2 :=
                   (!chain->
                    [x = (y * v)          [witnessed2]
                       = ((x * u) * v)    [witnessed1]
                       = (x * (u * v))    [Times.associative]
                       ==> (x * (u * v) = x)    [sym]
                       ==> (u * v = one)  [C1 Times.identity-lemma1]
                       ==> (u = one)      [Times.identity-lemma2]])}
            (!chain
             [x = (x * one)          [Times.right-one]
                = (x * u)            [C2]
                = y                  [witnessed1]]))

#......................................................................
define transitive :=
  (forall x y z . x divides y & y divides z ==> x divides z)

conclude transitive
  pick-any x y z
    assume (x divides y & y divides z)
      pick-witness u for (!elim x y) witnessed1
        pick-witness v for (!elim y z) witnessed2
          (!chain->
           [(x * (u * v))
            = ((x * u) * v)              [Times.associative]
            = (y * v)                    [witnessed1]
            = z                          [witnessed2]
            ==> (exists ?w . x * ?w = z) [existence]
            ==> (x divides z)            [characterization]])

#......................................................................
define Minus-lemma :=
  (forall x y z . x divides y & x divides z ==> x divides (y - z))

conclude Minus-lemma
  pick-any x y z
    assume (x divides y & x divides z)
      pick-witness u for (!elim x y) witnessed1
        pick-witness v for (!elim x z) witnessed2
          (!chain->
           [(y - z)
```

```
            = (x * u - x * v)  [witnessed1 witnessed2]
            = (x * (u - v))    [Minus.Times-Distributivity]
            ==> (x * (u - v) = y - z)          [sym]
            ==> (exists ?w . x * ?w = y - z)  [existence]
            ==> (x divides (y - z))           [characterization]])

define Mod-lemma :=
  (forall x y z . x divides y & x divides z & zero < z
                  ==> x divides y % z)

conclude Mod-lemma
  pick-any x y z
    assume (x divides y & x divides z & zero < z)
      let {C1 := (!chain->
                  [(zero < z)
                   ==> ((y / z) * z + y % z = y)
                            [division-algorithm-corollary1]]);
           C2 :=
            conclude (x divides (y / z) * z)
              (!chain->
               [(x divides z)
                ==> (x divides z * (y / z)) [product-left-lemma]
                ==> (x divides (y / z) * z) [Times.commutative]])}
      (!chain->
       [(x divides y)
        ==> (x divides ((y / z) * z + y % z))          [C1]
        ==> (C2 & (x divides ((y / z) * z + y % z)))  [augment]
        ==> (x divides y % z)                          [sum-lemma2]])

} # close module divides
} # close module N
```