lib/main/integer-times.ath

```
#####################################################
#
# Integer multiplication operator, Z.*
#

load "nat-times"
load "integer-plus"

extend-module Z {
declare *: [Z Z] -> Z
set-precedence * (get-precedence N.*)
module Times {
overload * N.*
define [x y] := [?x:N ?y:N]
assert axioms :=
  (fun [(pos x * pos y) = (pos (x * y))
        (pos x * neg y) = (neg (x * y))
        (neg x * pos y) = (neg (x * y))
        (neg x * neg y) = (pos (x * y))])
define [pos-pos pos-neg neg-pos neg-neg] := axioms

define associative := (forall a b c . (a * b) * c = a * (b * c))
define commutative := (forall a b . a * b = b * a)

# Unlike the case with addition, the signed integer representation is better
# than the Z.NN representation for proving these properties.  First, consider
# commutativity - since it involves only two variables, there are only four
# cases to consider.

datatype-cases commutative {
   (pos x) =>
     datatype-cases (forall ?b . pos x * ?b = ?b * pos x) {
       (pos y) =>
         (!chain [(pos x * pos y)
                  --> (pos (x * y))        [pos-pos]
                  --> (pos (y * x))        [N.Times.commutative]
                  <-- (pos y * pos x)      [pos-pos]])
     | (neg y) =>
         (!chain [(pos x * neg y)
                  --> (neg (x * y))        [pos-neg]
                  --> (neg (y * x))        [N.Times.commutative]
                  <-- (neg y * pos x)      [neg-pos]])
     }
 | (neg x) =>
     datatype-cases (forall ?b . neg x * ?b = ?b * neg x) {
       (pos y) =>
         (!chain [(neg x * pos y)
                  --> (neg (x * y))        [neg-pos]
                  --> (neg (y * x))        [N.Times.commutative]
                  <-- (pos y * neg x)      [pos-neg]])
     | (neg y) =>
         (!chain [(neg x * neg y)
                  --> (pos (x * y))        [neg-neg]
                  --> (pos (y * x))        [N.Times.commutative]
                  <-- (neg y * neg x)      [neg-neg]])
     }
}

# Since there are three variables, associativity requires eight cases, but each
# is straightforward.

let {assoc := N.Times.associative}
datatype-cases associative {
    (pos x) =>
      datatype-cases
        (forall ?b ?c . ((pos x) * ?b) * ?c = (pos x) * (?b Z.* ?c)) {
        (pos y) =>
```

```
          datatype-cases
             (forall ?c . ((pos x) * (pos y)) * ?c = (pos x) * ((pos y) * ?c)) {
            (pos z) =>
              (!chain [(((pos x) * (pos y)) * (pos z))
                   --> ((pos (x * y)) * (pos z))        [pos-pos]
                   --> (pos ((x * y) * z))              [pos-pos]
                   --> (pos (x * (y * z)))              [assoc]
                   <-- ((pos x) * (pos (y * z)))        [pos-pos]
                   <-- ((pos x) * ((pos y) * (pos z)))  [pos-pos]])
          | (neg z) =>
              (!chain [(((pos x) * (pos y)) * (neg z))
                   --> ((pos (x * y)) * (neg z))        [pos-pos]
                   --> (neg ((x * y) * z))              [pos-neg]
                   --> (neg (x * (y * z)))              [assoc]
                   <-- ((pos x) * (neg (y * z)))        [pos-neg]
                   <-- ((pos x) * ((pos y) * (neg z)))  [pos-neg]])
          }
      | (neg y) =>
          datatype-cases
             (forall ?c . ((pos x) * (neg y)) * ?c = (pos x) * ((neg y) * ?c)) {
            (pos z) =>
              (!chain [(((pos x) * (neg y)) * (pos z))
                   --> ((neg (x * y)) * (pos z))        [pos-neg]
                   --> (neg ((x * y) * z))              [neg-pos]
                   --> (neg (x * (y * z)))              [assoc]
                   <-- ((pos x) * (neg (y * z)))        [pos-neg]
                   <-- ((pos x) * ((neg y) * (pos z)))  [neg-pos]])
          | (neg z) =>
              (!chain [(((pos x) * (neg y)) * (neg z))
                   --> ((neg (x * y)) * (neg z))        [pos-neg]
                   --> (pos ((x * y) * z))              [neg-neg]
                   --> (pos (x * (y * z)))              [assoc]
                   <-- ((pos x) * (pos (y * z)))        [pos-pos]
                   <-- ((pos x) * ((neg y) * (neg z)))  [neg-neg]])
          }
      }
  | (neg x) =>
      datatype-cases
         (forall ?b ?c . ((neg x) * ?b) * ?c = (neg x) * (?b Z.* ?c)) {
        (pos y) =>
          datatype-cases
             (forall ?c .((neg x) * (pos y)) * ?c = (neg x) * ((pos y) * ?c)) {
            (pos z) =>
              (!chain [(((neg x) * (pos y)) * (pos z))
                   --> ((neg (x * y)) * (pos z))        [neg-pos]
                   --> (neg ((x * y) * z))              [neg-pos]
                   --> (neg (x * (y * z)))              [assoc]
                   <-- ((neg x) * (pos (y * z)))        [neg-pos]
                   <-- ((neg x) * ((pos y) * (pos z)))  [pos-pos]])
          | (neg z) =>
              (!chain [(((neg x) * (pos y)) * (neg z))
                   --> ((neg (x * y)) * (neg z))        [neg-pos]
                   --> (pos ((x * y) * z))              [neg-neg]
                   --> (pos (x * (y * z)))              [assoc]
                   <-- ((neg x) * (neg (y * z)))        [neg-neg]
                   <-- ((neg x) * ((pos y) * (neg z)))  [pos-neg]])
          }
      | (neg y) =>
          datatype-cases
             (forall ?c . ((neg x) * (neg y)) * ?c = (neg x) * ((neg y) * ?c)) {
            (pos z) =>
              (!chain [(((neg x) * (neg y)) * (pos z))
                   --> ((pos (x * y)) * (pos z))        [neg-neg]
                   --> (pos ((x * y) * z))              [pos-pos]
                   --> (pos (x * (y * z)))              [assoc]
                   <-- ((neg x) * (neg (y * z)))        [neg-neg]
                   <-- ((neg x) * ((neg y) * (pos z)))  [neg-pos]])
          | (neg z) =>
              (!chain [(((neg x) * (neg y)) * (neg z))
                   --> ((pos (x * y)) * (neg z))        [neg-neg]
```

```
                    --> (neg ((x * y) * z))               [pos-neg]
                    --> (neg (x * (y * z)))               [assoc]
                    <-- ((neg x) * (pos (y * z)))         [neg-pos]
                    <-- ((neg x) * ((neg y) * (neg z)))   [neg-neg]])
          }
      }
}

#

define Right-Distributive :=
   (forall a b c . (a + b) * c = a * c + b * c)

define Left-Distributive :=
   (forall a b c . c * (a + b) = c * a + c * b)

} # Times

#.......................................................................
# To prove Right Distributive, it seems best to use the Z->NN and NN->Z mappings.

extend-module NN {
overload * N.*
define-sort NN := Z.NN
declare *': [NN NN] -> NN
set-precedence *' (get-precedence *)
module Times {
define [a1 a2 b1 b2] := [?a1:N ?a2:N ?b1:N ?b2:N]
assert definition :=
      (forall a1 a2 b1 b2 .
              (nn a1 a2) *' (nn b1 b2) =
              (nn (a1 * b1 + a2 * b2)
                  (a1 * b2 + a2 * b1)))
} # Times
} # NN

extend-module Z-NN {
overload * N.*
define *' := NN.*'

define multiplicative-homomorphism :=
   (forall a b . (Z->NN (a * b)) = (Z->NN a) *' (Z->NN b))

let {f:(OP 1) := Z->NN; definition := NN.Times.definition}
  datatype-cases multiplicative-homomorphism {
    (pos x) =>
      datatype-cases
         (forall ?b . (f ((pos x) * ?b)) = (f (pos x)) *' (f ?b)) {
        (pos y) =>
          (!combine-equations
           (!chain [(f ((pos x) * (pos y)))
                --> (f (pos (x * y)))              [Times.pos-pos]
                --> (nn (x * y) Top.zero)          [to-pos]])
           (!chain [((f (pos x)) *' (f (pos y)))
                --> ((nn x Top.zero) *' (nn y Top.zero)) [to-pos]
                --> (nn (x * y + Top.zero * Top.zero)
                        (x * Top.zero + Top.zero * y))
                                                 [definition]
                --> (nn (x * y + Top.zero) (Top.zero + Top.zero))
                                                 [N.Times.right-zero
                                                  N.Times.left-zero]
                --> (nn (x * y) Top.zero)         [N.Plus.right-zero]]))
      | (neg y) =>
          (!combine-equations
           (!chain [(f ((pos x) * (neg y)))
                --> (f (neg (x * y)))            [Times.pos-neg]
                --> (nn Top.zero (x * y))         [to-neg]])
           (!chain [((f (pos x)) *' (f (neg y)))
                --> ((nn x Top.zero) *' (nn Top.zero y))
                                                 [to-pos to-neg]
```

```
                   --> (nn (x * Top.zero + Top.zero * y)
                            (x * y + Top.zero * Top.zero))  [definition]
                   --> (nn (Top.zero + Top.zero) (x * y + Top.zero))
                                                  [N.Times.right-zero
                                                   N.Times.left-zero]
                   --> (nn Top.zero x * y)          [N.Plus.right-zero]]))
         }
  | (neg x) =>
         datatype-cases
            (forall ?b . (f ((neg x) * ?b)) = (f (neg x)) *' (f ?b)) {
           (pos y) =>
             (!combine-equations
              (!chain [(f ((neg x) * (pos y)))
                   --> (f (neg (x * y)))            [Times.neg-pos]
                   --> (nn Top.zero (x * y))         [to-neg]])
              (!chain [((f (neg x)) *' (f (pos y)))
                   --> ((nn Top.zero x) *' (nn y Top.zero)) [to-neg to-pos]
                   --> (nn (Top.zero * y + x * Top.zero)
                            (Top.zero * Top.zero + x * y))
                                                  [definition]
                   --> (nn (Top.zero + Top.zero) (Top.zero + x * y))
                                                  [N.Times.right-zero
                                                   N.Times.left-zero]
                   --> (nn Top.zero (x * y))         [N.Plus.left-zero]]))
         | (neg y) =>
             (!combine-equations
              (!chain [(f ((neg x) * (neg y)))
                   --> (f (pos (x * y)))            [Times.neg-neg]
                   --> (nn (x * y) Top.zero)         [to-pos]])
              (!chain [((f (neg x)) *' (f (neg y)))
                   --> ((nn Top.zero x) *' (nn Top.zero y)) [to-neg]
                   --> (nn (Top.zero * Top.zero + x * y)
                            (Top.zero * y + x * Top.zero))
                                                  [definition]
                   --> (nn (Top.zero + x * y) (Top.zero + Top.zero))
                                                  [N.Times.right-zero
                                                   N.Times.left-zero]
                   --> (nn (x * y) Top.zero)         [N.Plus.left-zero]]))
         }
  }
} # Z-NN

#-----------------------------------------------------------------------
extend-module NN {
extend-module Times {
define Right-Distributive :=
    (forall a b c . (a +' b) *' c = a *' c +' b *' c)

datatype-cases Right-Distributive {
  (Z.nn a1 a2) =>
     datatype-cases
        (forall ?b ?c . ((nn a1 a2) +' ?b) *' ?c =
                         (nn a1 a2) *' ?c +' ?b *' ?c) {
       (Z.nn b1 b2) =>
         datatype-cases
             (forall ?c .
                ((nn a1 a2) +' (nn b1 b2)) *' ?c =
                 (nn a1 a2) *' ?c +' (nn b1 b2) *' ?c)
         {
           (Z.nn c1 c2) =>
             (!combine-equations
              (!chain
               [(((nn a1 a2) +' (nn b1 b2)) *' (nn c1 c2))
                = ((nn (a1 + b1) (a2 + b2)) *' (nn c1 c2))
                                                  [Plus.definition]
                = (nn ((a1 + b1) * c1 + (a2 + b2) * c2)
                      ((a1 + b1) * c2 + (a2 + b2) * c1))
                                                  [definition]
                = (nn ((a1 * c1 + b1 * c1) + (a2 * c2 + b2 * c2))
                      ((a1 * c2 + b1 * c2) + (a2 * c1 + b2 * c1)))
```

```
                                           [N.Times.right-distributive]])
              (!chain [((nn a1 a2) *' (nn c1 c2)
                       +' (nn b1 b2) *' (nn c1 c2))
                     = ((nn (a1 * c1 + a2 * c2) (a1 * c2 + a2 * c1))
                         +' (nn (b1 * c1 + b2 * c2)
                                    (b1 * c2 + b2 * c1)))
                                         [definition]
                     = (nn ((a1 * c1 + a2 * c2) + (b1 * c1 + b2 * c2))
                          ((a1 * c2 + a2 * c1) + (b1 * c2 + b2 * c1)))
                                         [Plus.definition]
                     = (nn ((a1 * c1 + b1 * c1) + (a2 * c2 + b2 * c2))
                          ((a1 * c2 + b1 * c2) + (a2 * c1 + b2 * c1)))
                                         [N.Plus.commutative
                                          N.Plus.associative]]))
          }
      }
}
} # Times
} # NN

extend-module Times {
define +' := NN.+'
define *' := NN.*'

conclude Right-Distributive
  pick-any a:Z b:Z c:Z
    let {f:(OP 1) := Z->NN; g:(OP 1) := NN->Z;
         f-application :=
           conclude ((f ((a + b) * c)) = (f (a * c + b * c)))
             (!chain [(f ((a + b) * c))
             --> ((f (a + b)) *' (f c))
                                    [Z-NN.multiplicative-homomorphism]
             --> (((f a) +' (f b)) *' (f c))
                                    [Z-NN.additive-homomorphism]
             --> (((f a) *' (f c)) +' ((f b) *' (f c)))
                                    [NN.Times.Right-Distributive]
             <-- ((f (a * c)) +' (f (b * c)))
                                    [Z-NN.multiplicative-homomorphism]
             <-- (f (a * c + b * c)) [Z-NN.additive-homomorphism]])}
    conclude ((a + b) * c = a * c + b * c)
      (!chain [((a + b) * c)
          <-- (g (f ((a + b) * c)))     [Z-NN.inverse]
          --> (g (f (a * c + b * c)))   [f-application]
          --> (a * c + b * c)           [Z-NN.inverse]])

# Since we already have proved commutativity, we can use it for
# Left-Distributive.

conclude Left-Distributive
  pick-any a:Z b:Z c:Z
    (!chain [(c * (a + b))
             --> ((a + b) * c)     [commutative]
             --> (a * c + b * c)   [Right-Distributive]
             --> (c * a + c * b)   [commutative]])

define Right-Identity := (forall a . a * one = a)
define Left-Identity :=  (forall a . one * a = a)

datatype-cases Right-Identity {
  (pos x) =>
    (!chain [((pos x) * one)
         --> ((pos x) * (pos N.one)) [one-definition]
         --> (pos (x * N.one))       [pos-pos]
         --> (pos x)                 [N.Times.right-one]])
| (neg x) =>
    (!chain [((neg x) * one)
         --> ((neg x) * (pos N.one)) [one-definition]
         --> (neg (x * N.one))       [neg-pos]
         --> (neg x)                 [N.Times.right-one]])
}
```

```

# Since we already have proved commutativity, we can use it for Left-Identity.

conclude Left-Identity
  pick-any a:Z
    (!chain [(one * a)
         --> (a * one)              [commutative]
         --> a                      [Right-Identity]])

define No-Zero-Divisors :=
   (forall a b . a * b = zero ==> a = zero | b = zero)

datatype-cases No-Zero-Divisors {
  (pos x) =>
    datatype-cases
       (forall ?b . (pos x) * ?b = zero ==>
                    (pos x) = zero | ?b = zero) {
      (pos y) =>
        assume ((pos x) * (pos y) = zero)
          let {C :=
                (!chain->
                 [(pos (x * y))
                  <-- ((pos x) * (pos y)) [pos-pos]
                 --> zero                [((pos x) * (pos y) = zero)]
                 --> (pos Top.zero)        [zero-definition]
                 ==> (x * y = Top.zero)    [Z-structure-axioms]
                 ==> (x = Top.zero | y = Top.zero)  [N.Times.no-zero-divisors]])}
          (!cases C
            assume (x = Top.zero)
              let {_ := (!chain [(pos x)
                                 --> (pos Top.zero)  [(x = Top.zero)]
                                 <-- zero          [zero-definition]])}
              (!left-either ((pos x) = zero) ((pos y) = zero))
            assume (y = Top.zero)
              let {_ := (!chain [(pos y)
                                 --> (pos Top.zero)   [(y = Top.zero)]
                                 <-- zero           [zero-definition]])}
              (!right-either ((pos x) = zero) ((pos y) = zero)))
    | (neg y) =>
        assume ((pos x) * (neg y) = zero)
          let {C :=
                (!chain->
                 [(neg (x * y))
                  <-- ((pos x) * (neg y))      [pos-neg]
                  --> zero                     [((pos x) * (neg y) = zero)]
                  --> (neg Top.zero)             [zero-property]
                  ==> (x * y = Top.zero)         [Z-structure-axioms]
                  ==> (x = Top.zero | y = Top.zero) [N.Times.no-zero-divisors]])}
          (!cases C
            assume (x = Top.zero)
              let {_ := (!chain [(pos x)
                                 --> (pos Top.zero)  [(x = Top.zero)]
                                 <-- zero          [zero-definition]])}
              (!left-either ((pos x) = zero) ((neg y) = zero))
            assume (y = Top.zero)
              let {_ := (!chain [(neg y)
                                 --> (neg Top.zero)  [(y = Top.zero)]
                                 <-- zero          [zero-property]])}
              (!right-either ((pos x) = zero) ((neg y) = zero)))
    }
 | (neg x) =>
   datatype-cases
      (forall ?b . (neg x) * ?b = zero ==> (neg x) = zero | ?b = zero)
   { (pos y) =>
       assume ((neg x) * (pos y) = zero)
         let {C := (!chain->
                     [(neg (x * y))
                      <-- ((neg x) * (pos y))    [neg-pos]
                      --> zero                   [(((neg x) * (pos y)) = zero)]
                      --> (neg Top.zero)           [zero-property]
```

```
                    ==> (x * y = Top.zero)      [Z-structure-axioms]
                    ==> (x = Top.zero | y = Top.zero) [N.Times.no-zero-divisors]])}
         (!cases C
           assume (x = Top.zero)
             let {_ := (!chain [(neg x)
                                --> (neg Top.zero)  [(x = Top.zero)]
                                <-- zero       [zero-property]])}
               (!left-either ((neg x) = zero) ((pos y) = zero))
           assume (y = Top.zero)
             let {_ := (!chain [(pos y)
                                --> (pos Top.zero)  [(y = Top.zero)]
                                <-- zero       [zero-definition]])}
               (!right-either ((neg x) = zero) ((pos y) = zero)))
   | (neg y) =>
       assume ((neg x) * (neg y) = zero)
         let {C := (!chain->
                    [(pos (x * y))
                     <-- ((neg x) * (neg y))[neg-neg]
                     --> zero              [((neg x) * (neg y) = zero)]
                     --> (pos Top.zero)     [zero-definition]
                     ==> (x * y = Top.zero) [Z-structure-axioms]
                     ==> (x = Top.zero | y = Top.zero)
                                           [N.Times.no-zero-divisors]])}
         (!cases C
           assume (x = Top.zero)
             let {_ := (!chain [(neg x)
                                --> (neg Top.zero)  [(x = Top.zero)]
                                <-- zero            [zero-property]])}
               (!left-either ((neg x) = zero) ((neg y) = zero))
           assume (y = Top.zero)
             let {_ := (!chain [(neg y)
                                --> (neg Top.zero)  [(y = Top.zero)]
                                <-- zero            [zero-property]])}
               (!right-either ((neg x) = zero) ((neg y) = zero)))
   }
}

define Nonzero-Product :=
   (forall a b . ~ (a = zero | b = zero) ==> a * b =/= zero)

conclude Nonzero-Product
  pick-any a b
    (!contra-pos (!instance No-Zero-Divisors [a b]))
} # Times
} # Z
```