lib/main/integer-plus.ath

```
#..........................................................................
#
# Integer datatype, Z, and an addition operator, Z.Plus
#

load "nat-less"

load "nat-minus"

structure Z := (pos N) | (neg N)

expand-input pos, neg [int->nat]

set-precedence (pos neg) (plus 1 (get-precedence *))

assert* Z-structure-axioms :=
        [(pos x = pos y <==> x = y)
         (neg x = neg y <==> x = y)
         ((exists y . x = pos y) |
          (exists y . x = neg y))]

set-precedence (pos neg) (plus (get-precedence *) 1)

module Z {
open N
declare zero, one: Z
assert zero-definition := (zero = pos N.zero)
assert zero-property := (zero = neg N.zero)
assert one-definition := (one = pos N.one)

define [a b c] := [?a:Z ?b:Z ?c:Z]

declare +: [Z Z] -> Z

module Plus {
overload + N.+
define [x y] := [?x:N ?y:N]

assert axioms :=
 (fun [(pos x + pos y) = (pos (x + y))     #pos-pos
       (pos x + neg y) =
           [(neg (y - x)) when (x < y)     #pos-neg-case1
            (pos (x - y)) when (~ x < y)] #pos-neg-case2
       (neg x + pos y) =
           [(pos (y - x)) when (x < y)     #neg-pos-case1
            (neg (x - y)) when (~ x < y)] #neg-pos-case2
       (neg x + neg y) = (neg (x + y))])  #neg-neg

# Alternatively:
assert* axioms :=
  [(pos x + pos y = pos (x + y))              #pos-pos
   (  x < y ==> pos x + neg y = neg (y - x)) #pos-neg-case1
   (~ x < y ==> pos x + neg y = pos (x - y)) #pos-neg-case2
   (  x < y ==> neg x + pos y = pos (y - x)) #neg-pos-case1
   (~ x < y ==> neg x + pos y = neg (x - y)) #neg-pos-case2
   (neg x + neg y = neg (x + y))]            #neg-neg

define [pos-pos pos-neg-case1 pos-neg-case2
        neg-pos-case1 neg-pos-case2 neg-neg] := axioms
} # close module Plus

declare negate: [Z] -> Z [120]

module Negate {
define x := ?x:N
assert positive := (forall x . negate pos x = neg x)
assert negative := (forall x . negate neg x = pos x)
```

```
} # close module Negate

declare -: [Z Z] -> Z

module Minus {
assert definition :=  (forall a b . a - b =  a + negate b)
} # close module Minus

extend-module Plus {
open Minus
overload - N.-

define Right-Inverse := (forall a . a + negate a = zero)

datatype-cases Right-Inverse {
  (pos x) =>
    conclude (pos x + negate pos x = zero)
      let {_ := (!chain-> [true ==> (~ x < x) [Less.irreflexive]])}
      (!chain [(pos x + negate pos x)
           --> (pos x + neg x)      [Negate.positive]
           --> (pos (x - x))        [pos-neg-case2]
           --> (pos Top.zero)       [N.Minus.second-equal]
           <-- zero                [zero-definition]])
| (neg x) =>
    conclude (neg x + negate neg x = zero)
      let {_ := (!chain-> [true ==> (~ x < x) [Less.irreflexive]])}
      (!chain [(neg x + negate neg x)
           --> (neg x + pos x)      [Negate.negative]
           --> (neg (x - x))        [neg-pos-case2]
           --> (neg Top.zero)       [N.Minus.second-equal]
           <-- zero                 [zero-property]])
}

define Right-Identity := (forall a . a + zero = a)
define Left-Identity := (forall a . zero + a = a)

datatype-cases Right-Identity {
  (pos x) =>
    conclude (pos x + zero = pos x)
     (!chain [(pos x + zero)
          --> (pos x + pos Top.zero)    [zero-definition]
          --> (pos (x + Top.zero))      [pos-pos]
          --> (pos x)                   [N.Plus.right-zero]])
| (neg x) =>
    conclude (neg x + zero = neg x)
      let {_ := (!chain-> [true ==> (~ x < Top.zero) [Less.not-zero]])}
      (!chain [(neg x + zero)
           --> (neg x + pos Top.zero)  [zero-definition]
           --> (neg (x - Top.zero))    [neg-pos-case2]
           --> (neg x)                 [N.Minus.zero-right]])
}

datatype-cases Left-Identity {
  (pos x) =>
    conclude (zero + pos x = pos x)
      (!chain [(zero + pos x)
           --> (pos Top.zero + pos x)   [zero-definition]
           --> (pos (Top.zero + x))     [pos-pos]
           --> (pos x)                  [N.Plus.left-zero]])
| (neg x) =>
    conclude (zero + neg x = neg x)
      (!chain [(zero + neg x)
          --> (neg Top.zero + neg x)  [zero-property]
          --> (neg (Top.zero + x))    [neg-neg]
          --> (neg x)                 [N.Plus.left-zero]])
}

define associative := (forall a b c . (a + b) + c = a + (b + c))

# A direct proof of Z.Plus.associative would be very long and tedious,
```

```
# as there are three variables to consider in combinations of positive
# and negative values and the pos-neg and neg-pos
# cases require lemmas about reassociating Minus terms.  Instead we
# transform to a representation where the proof is easy (though some of
# the proofs about the tranformation are somewhat long themselves).

} # close module Plus

structure NN := (nn N N)

module NN {
overload + N.+

define [a b c] := [?a:NN ?b:NN ?c:NN]

declare +': [NN NN] -> NN [110]

module Plus {
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)))

define associative := (forall a b c . (a +' b) +' c = a +' (b +' c))

datatype-cases associative {
  (nn a1 a2) =>
    datatype-cases
        (forall b c . ((nn a1 a2) +' b) +' c =
                        (nn a1 a2) +' (b +' c)) {
      (nn b1 b2) =>
        datatype-cases
            (forall c . ((nn a1 a2) +' (nn b1 b2)) +' c =
                         (nn a1 a2) +' ((nn b1 b2) +' c)) {
          (nn c1 c2) =>
            (!chain
             [(((nn a1 a2) +' (nn b1 b2)) +' (nn c1 c2))
          --> ((nn a1 + b1 a2 + b2) +' (nn c1 c2)) [definition]
          --> (nn (a1 + b1) + c1 (a2 + b2) + c2)   [definition]
          --> (nn a1 + (b1 + c1) a2 + (b2 + c2))   [N.Plus.associative]
          <-- ((nn a1 a2) +' (nn b1 + c1 b2 + c2)) [definition]
          <-- ((nn a1 a2) +' ((nn b1 b2) +' (nn c1 c2)))
                                                   [definition]])
        }
    }
}
} # close module Plus
} # close module NN

declare Z->NN: [Z] -> NN
declare NN->Z: [NN] -> Z

module Z-NN {
overload (+ N.+) (- N.-)
define [x y] := [?x:N ?y:N]
assert to-pos := (forall x . Z->NN pos x = nn x Top.zero)
assert to-neg := (forall x . Z->NN neg x = nn Top.zero x)
assert from-pos :=
  (forall x y . ~ x < y ==> NN->Z (nn x y) = pos (x - y))
assert from-neg :=
  (forall x y . x < y ==> NN->Z (nn x y) = neg (y - x))

define inverse := (forall a . NN->Z Z->NN a = a)

datatype-cases inverse {
  (pos x) => {
    (!chain-> [true ==> (~ x < Top.zero) [N.Less.not-zero]]);
    (!chain [(NN->Z Z->NN pos x)
         --> (NN->Z (nn x Top.zero)) [to-pos]
         --> (pos (x - Top.zero))    [from-pos]
```

```
         --> (pos x)                  [N.Minus.zero-right]])
  }
| (neg x) =>
   (!two-cases
     assume (x = Top.zero)
       let {_ := (!chain-> [true ==> (~ Top.zero < Top.zero)
                                              [N.Less.irreflexive]])}
       (!chain [(NN->Z Z->NN neg x)
            --> (NN->Z (nn Top.zero x))       [to-neg]
            --> (NN->Z (nn Top.zero Top.zero)) [(x = Top.zero)]
            --> (pos (Top.zero - Top.zero))   [from-pos]
            --> (pos Top.zero)                [N.Minus.zero-right]
            <-- zero                          [zero-definition]
            --> (neg Top.zero)                [zero-property]
            <-- (neg x)                       [(x = Top.zero)]])
     assume (~ x = Top.zero)
       let {A := (!chain->
                  [true ==> (Top.zero <= x)   [N.Less=.zero<=]
                        ==> (Top.zero < x | Top.zero = x)
                                         [N.Less=.definition]])}
       (!cases A
         assume (Top.zero < x)
           (!chain [(NN->Z Z->NN neg x)
                --> (NN->Z (nn Top.zero x))   [to-neg]
                --> (neg (x - Top.zero))      [from-neg]
                --> (neg x)                   [N.Minus.zero-right]])
         assume (Top.zero = x)
           (!from-complements (NN->Z Z->NN neg x = neg x)
                              (!sym (Top.zero = x)) (x =/= Top.zero))))
} # datatype-cases
} # close module Z-NN

module NN-equivalence {
overload - N.-
define [x y] := [?x:N ?y:N]
assert case1 :=
  (forall x y . x < y ==> (nn x y) = nn Top.zero (y - x))
assert case2 :=
  (forall x y . ~ x < y ==> (nn x y) = nn (x - y) Top.zero)
} # close module NN-equivalence

extend-module Z-NN {
define +' := NN.+'

define additive-homomorphism :=
   (forall a b . Z->NN (a + b) = (Z->NN a) +' (Z->NN b))

## Proof sketch (uses force):

datatype-cases additive-homomorphism {
  (pos x) =>
    datatype-cases
       (forall ?b .
          (Z->NN (pos x + ?b)) = (Z->NN (pos x)) +' (Z->NN ?b)) {
      (pos y) =>
        (!combine-equations
         (!chain [(Z->NN (pos x + pos y))
              --> (Z->NN (pos (x + y)))               [Plus.pos-pos]
              --> (nn (x + y) Top.zero)               [to-pos]])
         (!chain [((Z->NN pos x) +' (Z->NN pos y))
              --> ((nn x Top.zero) NN.+' (nn y Top.zero)) [to-pos]
              --> (nn (x + y) (Top.zero + Top.zero)) [NN.Plus.definition]
              --> (nn (x + y) Top.zero)               [N.Plus.right-zero]]))
    | (neg y) =>
      (!two-cases
        assume (x < y)
          (!force (Z->NN ((pos x) + (neg y)) =
                  (Z->NN pos x) +' (Z->NN neg y)))
        assume (~ x < y)
          (!force (Z->NN ((pos x) + (neg y)) =
```

```
                    (Z->NN pos x) +' (Z->NN neg y))))
    }
| (neg x) =>
    datatype-cases
       (forall ?b . Z->NN (neg x + ?b) =
                   (Z->NN neg x) +' (Z->NN ?b)) {
      (pos y) =>
        (!two-cases
          assume (x < y)
           let {_ := (!chain-> [(x < y) ==> (~ y < x)
                                   [N.Less.asymmetric]])}
           (!force (Z->NN (neg x + pos y) =
                   (Z->NN neg x) +' (Z->NN pos y)))
          assume (~ x < y)
           let {A := (!chain->
                      [(~ x < y)
                       ==> (y <= x)     [N.Less=.trichotomy2]
                       ==> (y < x | y = x)
                                        [N.Less=.definition]])}
           (!cases A
             assume (y < x)
               (!force (Z->NN (neg x + pos y) =
                       (Z->NN neg x)  +' (Z->NN pos y)))
             assume (y = x)
               let {_ := (!chain-> [true ==> (~ x < x)
                                        [N.Less.irreflexive]])}
               (!force (Z->NN (neg x + pos y) =
                       (Z->NN neg x) +' (Z->NN pos y)))))
    | (neg y) =>
        (!force (Z->NN (neg x + neg y) =
                (Z->NN neg x) +' (Z->NN neg y)))
    }
}

#......................................................................
## The complete proof:

datatype-cases additive-homomorphism {
  (pos x) =>
    datatype-cases
       (forall ?b .
          (Z->NN (pos x + ?b)) = (Z->NN (pos x)) +' (Z->NN ?b)) {
      (pos y) =>
        (!combine-equations
         (!chain [(Z->NN (pos x + pos y))
              --> (Z->NN (pos (x + y)))              [Plus.pos-pos]
              --> (nn (x + y) Top.zero)              [to-pos]])
         (!chain [((Z->NN pos x) +' (Z->NN pos y))
              --> ((nn x Top.zero) +' (nn y Top.zero))           [to-pos]
              --> (nn (x + y) (Top.zero + Top.zero)) [NN.Plus.definition]
              --> (nn (x + y) Top.zero)            [N.Plus.right-zero]]))
    | (neg y) =>
      (!two-cases
        assume (x < y)
          (!combine-equations
           (!chain [(Z->NN (pos x + neg y))
                --> (Z->NN neg (y - x))       [Plus.pos-neg-case1]
                --> (nn Top.zero (y - x))     [to-neg]])
           (!chain [((Z->NN pos x) +' (Z->NN neg y))
                --> ((nn x Top.zero) +' (nn Top.zero y))    [to-pos to-neg]
                --> (nn (x + Top.zero) (Top.zero + y))[NN.Plus.definition]
                --> (nn x y)           [N.Plus.right-zero N.Plus.left-zero]
                --> (nn Top.zero (y - x))        [NN-equivalence.case1]]))
        assume (~ x < y)
          (!combine-equations
           (!chain [(Z->NN (pos x + neg y))
                --> (Z->NN pos (x - y))          [Plus.pos-neg-case2]
                --> (nn (x - y) Top.zero)        [to-pos]])
           (!chain [((Z->NN pos x) +' (Z->NN neg y))
                --> ((nn x Top.zero) +' (nn Top.zero y)) [to-pos to-neg]
```

```
                --> (nn (x + Top.zero) (Top.zero + y))[NN.Plus.definition]
                --> (nn x y)          [N.Plus.right-zero N.Plus.left-zero]
                --> (nn (x - y) Top.zero)       [NN-equivalence.case2]])))
    }
| (neg x) =>
    datatype-cases
       (forall ?b . Z->NN (neg x + ?b) =
                   (Z->NN neg x) +' (Z->NN ?b)) {
      (pos y) =>
        (!two-cases
          assume (x < y)
           let {_ := (!chain-> [(x < y) ==> (~ y < x)
                                   [N.Less.asymmetric])}
           (!combine-equations
            (!chain [(Z->NN (neg x + pos y))
                 --> (Z->NN pos (y - x))          [Plus.neg-pos-case1]
                 --> (nn (y - x) Top.zero)        [to-pos]])
            (!chain [((Z->NN neg x) +' (Z->NN pos y))
                 --> ((nn Top.zero x) +' (nn y Top.zero)) [to-neg to-pos]
                 --> (nn (Top.zero + y) (x + Top.zero))
                                                     [NN.Plus.definition]
                 --> (nn y x)        [N.Plus.right-zero N.Plus.left-zero]
                 --> (nn (y - x) Top.zero)      [NN-equivalence.case2]]))
          assume (~ x < y)
           let {A := (!chain->
                      [(~ x < y)
                       ==> (y <= x)    [N.Less=.trichotomy2]
                       ==> (y < x | y = x)
                                       [N.Less=.definition]])}
           (!cases A
             assume (y < x)
              (!combine-equations
               (!chain [(Z->NN (neg x + pos y))
                    --> (Z->NN neg (x - y))     [Plus.neg-pos-case2]
                    --> (nn Top.zero (x - y))   [to-neg]])
               (!chain [((Z->NN neg x) +' (Z->NN pos y))
                    --> ((nn Top.zero x) +' (nn y Top.zero))
                                                [to-neg to-pos]
                    --> (nn (Top.zero + y) (x + Top.zero))
                                                [NN.Plus.definition]
                    --> (nn y x)                [N.Plus.left-zero
                                                 N.Plus.right-zero]
                    --> (nn Top.zero (x - y))  [NN-equivalence.case1]]))
             assume (y = x)
               let {_ := (!chain-> [true ==> (~ x < x)
                                            [N.Less.irreflexive])}
               (!combine-equations
                (!chain [(Z->NN (neg x + pos y))
                    --> (Z->NN neg (x - y))     [Plus.neg-pos-case2]
                    --> (nn Top.zero (x - y))   [to-neg]
                    --> (nn Top.zero (x - x))   [(y = x)]
                    --> (nn Top.zero Top.zero) [N.Minus.second-equal]])
                (!chain [((Z->NN neg x) +' (Z->NN pos y))
                     --> ((nn Top.zero x) +' (nn y Top.zero))
                                               [to-neg to-pos]
                     --> (nn (Top.zero + y) (x + Top.zero))
                                               [NN.Plus.definition]
                     --> (nn y x)              [N.Plus.left-zero
                                                N.Plus.right-zero]
                     --> (nn x x)              [(y = x)]
                     --> (nn (x - x) Top.zero) [NN-equivalence.case2]
                     --> (nn Top.zero Top.zero)[N.Minus.second-equal]]))))
    | (neg y) =>
        (!combine-equations
         (!chain [(Z->NN (neg x + neg y))
              --> (Z->NN neg (x + y))              [Plus.neg-neg]
              --> (nn Top.zero (x + y))            [to-neg]])
         (!chain [((Z->NN neg x) +' (Z->NN neg y))
              --> ((nn Top.zero x) +' (nn Top.zero y))           [to-neg]
              --> (nn (Top.zero + Top.zero) (x + y)) [NN.Plus.definition]
```

```
              --> (nn Top.zero (x + y))            [N.Plus.right-zero]]))
    }
} # datatype-cases
} # close module Z-NN

# Finally:

extend-module Plus {
define +' := NN.+'

conclude associative
  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 + (b + c))))
            (!chain
             [(f ((a + b) + c))
          --> ((f (a + b)) +' (f c))     [Z-NN.additive-homomorphism]
          --> ((f a +' f b) +' f c)      [Z-NN.additive-homomorphism]
          --> (f a +' (f b +' f c))      [NN.Plus.associative]
          <-- (f a +' (f (b + c)))       [Z-NN.additive-homomorphism]
          <-- (f (a + (b + c)))          [Z-NN.additive-homomorphism]])}
    conclude (((a + b) + c) = a + (b + c))
      (!chain [((a + b) + c)
           <-- (g f ((a + b) + c)) [Z-NN.inverse]
           --> (g f (a + (b + c))) [f-application]
           --> (a + (b + c))       [Z-NN.inverse]])

# Next, the commutative property:

define commutative := (forall a b . a + b = b + a)

# Prove it by transforming to NN representation, as with associativity.

extend-module NN {
extend-module Plus {
define commutative := (forall a b . a +' b = b +' a)

datatype-cases commutative {
  (nn a1 a2) =>
    datatype-cases
       (forall ?b . (nn a1 a2) +' ?b = ?b +' (nn a1 a2)) {
      (nn b1 b2) =>
        (!chain [((nn a1 a2) +' (nn b1 b2))
             --> (nn (a1 + b1) (a2 + b2))     [definition]
             --> (nn (b1 + a1) (b2 + a2))     [N.Plus.commutative]
             <-- ((nn b1 b2) +' (nn a1 a2))   [definition]])
    }
}
} # close module Plus
} # close module NN

conclude commutative
  pick-any a:Z b:Z
    let {f:(OP 1) := Z->NN;
         g:(OP 1) := NN->Z;
         f-application :=
          conclude (f (a + b) = f (b + a))
            (!chain [(f (a + b))
                 --> (f a +' f b)   [Z-NN.additive-homomorphism]
                 --> (f b +' f a)   [NN.Plus.commutative]
                 <-- (f (b + a))    [Z-NN.additive-homomorphism]])}
    conclude (a + b = b + a)
      (!chain [(a + b)
           <-- (g f (a + b))    [Z-NN.inverse]
           --> (g f (b + a))    [f-application]
           --> (b + a)          [Z-NN.inverse]])
```

```
} # close module Plus

} # close module Z
```