lib/main/nat-plus.ath

```
##################################################
#
# Natural number datatype and Plus function
#

datatype N := zero | (S N)
set-precedence S 350
assert (datatype-axioms "N")

# Procedures for transforming an Athena int to a ground-term N and vice-versa

define (int->nat n) :=
  (check ((integer-numeral? n)
          (check ((n less? 1) zero)
                 (else (S (int->nat (n minus 1))))))
        (else n))

define (nat->int n) :=
  match n {
    zero => 0
  | (S k) => (plus (nat->int k) 1)
  | _ => n
  }

define (nat->int n) :=
  match n {
    zero => 0
  | (S k) => (plus (nat->int k) 1)
  | ((some-symbol f) (some-list args)) => try { (make-term f (map nat->int args)) | n }
  | (list-of h rest) => (add (nat->int h) (map nat->int rest))
  | _ => n
  }

module N {

define [zero S] := [zero S]

declare one, two: N

define [k m n p x y z] := [?k:N ?m:N ?n:N ?p:N ?x:N ?y:N ?z:N]

assert one-definition := (one = (S zero))
assert two-definition := (two = (S one))

define S-not-zero     := (forall n . (S n) =/= zero)
define one-not-zero   := (one =/= zero)
define S-injective      := (forall m n . (S m) = (S n) <==> m = n)

# S-not-zero is essentially the same as one of the propositions
# returned by (datatype-axioms "N"):

conclude S-not-zero
  pick-any n
    (!sym (!instance (first (datatype-axioms "N")) n))

conclude S-not-zero
  pick-any n
    (!chain->
     [true ==> (zero =/= (S n)) [(datatype-axioms "N")]
           ==> ((S n) =/= zero) [sym]])

# Next we use S-not-zero to prove one-not-zero.

(!by-contradiction one-not-zero
   assume (one = zero)
     let {is := conclude ((S zero) = zero)
                  (!chain
```

```
                   [(S zero)
                    <-- one          [one-definition]
                    --> zero         [(one = zero)]]);
          is-not := (!chain-> [true ==> ((S zero) =/= zero)
                                        [S-not-zero]])}
       (!absurd is is-not))

# One direction of S-injective is the second proposition
# returned by (datatype-axioms "N")

conclude S-injective
  pick-any m:N n:N
    let {right := (!chain [((S m) = (S n)) ==> (m = n)
                              [(second (datatype-axioms "N"))]]);
         left := assume (m = n)
                   (!chain [(S m) --> (S n) [(m = n)]])}
      (!equiv right left)

# The following is equivalent to another of the propositions
# returned by (datatype-axioms "N"), but here we show
# it is a theorem.

define nonzero-S :=
  (forall n . n =/= zero ==> (exists m . n = (S m)))

define S-not-same := (forall n . (S n) =/= n)

by-induction nonzero-S {
  zero => assume (zero =/= zero)
           (!from-complements (exists ?m (zero = (S ?m)))
                              (!reflex zero)
                              (zero =/= zero))
| (S m) => assume ((S m) =/= zero)
    let {_ := (!reflex (S m))}
      (!egen (exists ?m . (S m) = (S ?m)) m)
}

by-induction S-not-same {
  zero => conclude ((S zero) =/= zero)
            (!instance S-not-zero zero)
| (S n) =>
  let {induction-hypothesis := ((S n) =/= n)}
    (!chain-> [induction-hypothesis
           ==> ((S (S n)) =/= (S n)) [S-injective]])
}

########################################
#
# Addition operator, Plus
#

declare +: [N N] -> N [200]

module Plus {

# Axioms:
assert* Plus-def := [(n + zero = n)
                     (n + S m = S (n + m))]

define [right-zero right-nonzero] := Plus-def
#assert right-zero    := (forall n . n + zero = n)
#assert right-nonzero := (forall m n . n + (S m) = (S (n + m)))

# Theorems:

define left-zero    := (forall n . zero + n = n)
define left-nonzero := (forall n m . (S m) + n = (S (m + n)))

by-induction left-zero {
  zero => conclude (zero + zero = zero)
```

```
            (!chain [(zero + zero) --> zero [right-zero]])
| (S n) => conclude (zero + (S n) = (S n))
             let {induction-hypothesis := (zero + n = n)}
             (!chain [(zero + (S n))
                      --> (S (zero + n)) [right-nonzero]
                      --> (S n)          [induction-hypothesis]])
}

by-induction left-nonzero {
  zero =>
  pick-any m
    (!chain [((S m) + zero)
             --> (S m)            [right-zero]
             <-- (S (m + zero))   [right-zero]])
| (S n) =>
  let {induction-hypothesis := (forall ?m . (S ?m) + n = (S (?m + n)))}
  pick-any m
    (!chain [((S m) + (S n))
             --> (S ((S m) + n)) [right-nonzero]
             --> (S (S (m + n))) [induction-hypothesis]
             <-- (S (m + (S n))) [right-nonzero]])
}

# Adding one is the same as applying S

define right-one := (forall n . n + one = (S n))
define left-one  := (forall n . one + n = (S n))

conclude right-one
  pick-any n
    (!chain [(n + one)
             --> (n + (S zero))    [one-definition]
             --> (S (n + zero))    [right-nonzero]
             --> (S n)             [right-zero]])

conclude left-one
  pick-any n
    (!chain [(one + n)
             --> ((S zero) + n)    [one-definition]
             --> (S (zero + n))    [left-nonzero]
             --> (S n)             [left-zero]])

# Associativity and commutativity:

define associative := (forall m p n . (m + p) + n = m + (p + n))
define commutative := (forall n m . m + n = n + m)

by-induction associative {
  zero =>
    pick-any p n
      (!chain [((zero + p) + n)
               --> (p + n)          [left-zero]
               <-- (zero + (p + n)) [left-zero]])
| (S m) =>
    let {induction-hypothesis :=
           (forall ?p ?n . (m + ?p) + ?n = m + (?p + ?n))}
      pick-any p n
        (!chain
         [(((S m) + p) + n)
          --> ((S (m + p)) + n) [left-nonzero]
          --> (S ((m + p) + n)) [left-nonzero]
          --> (S (m + (p + n))) [induction-hypothesis]
          <-- ((S m) + (p + n)) [left-nonzero]])
}

by-induction commutative {
  zero =>
    pick-any m
      (!chain [(m + zero)
               --> m             [right-zero]
```

```
               <-- (zero + m)    [left-zero]])
| (S n) =>
    pick-any m
      let {induction-hypothesis := (forall ?m . ?m + n = n + ?m)}
      (!chain [(m + (S n))
               --> (S (m + n)) [right-nonzero]
               --> (S (n + m)) [induction-hypothesis]
               <-- ((S n) + m) [left-nonzero]])
}

# A cancellation property

define =-cancellation :=
  (forall k m n . m + k = n + k ==> m = n)

by-induction =-cancellation {
  zero =>
    pick-any m n
      assume assumption := (m + zero = n + zero)
        (!chain [m <-- (m + zero) [right-zero]
                   --> (n + zero) [assumption]
                   --> n          [right-zero]])
| (S k) =>
    let {induction-hypothesis :=
           (forall ?m ?n . ?m + k = ?n + k ==> ?m = ?n)}
    pick-any m n
      assume assumption := (m + S k = n + S k)
        (!chain->
         [(S (m + k))
          <-- (m + S k)            [right-nonzero]
          --> (n + S k)            [assumption]
          --> (S (n + k))          [right-nonzero]
          ==> (m + k = n + k)      [S-injective]
          ==> (m = n)              [induction-hypothesis]])
}

# If a sum of two natural numbers is zero, each is zero. (Here we only show
# the first is zero.)

define squeeze-property := (forall m n . m + n = zero ==> m = zero)

conclude squeeze-property
  pick-any m n
    assume A := (m + n = zero)
      (!by-contradiction (m = zero)
         assume (m =/= zero)
           let {C := (!chain->
                       [(m =/= zero)
                        ==> (exists ?k . m = (S ?k)) [nonzero-S]])}
           pick-witness k for C witnessed
             let {is := conclude ((S (k + n)) = zero)
                          (!chain [(S (k + n))
                                   <-- ((S k) + n)  [left-nonzero]
                                   <-- (m + n)      [witnessed]
                                   --> zero         [A]]);
                  is-not := (!chain-> [true ==> ((S (k + n)) =/= zero)
                                                [S-not-zero]])}
             (!absurd is is-not))
} # module N.Plus
} # module N
```