### Natural number datatype and Plus function

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

```plaintext
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
   }
```

```plaintext
module N {
   declare one, two: 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
   (!by-contradiction one-not-zero
      assume (one = zero)
      let [is := conclude ((S zero) = zero)]
   )
   conclude S-not-zero
   pick-any n
   (!by-contradiction one-not-zero
      assume (one = zero)
      let [is := conclude ((S zero) = zero)]
   )
```

lib/main/nat-plus.ath

66  [[S zero])
67       <-- one       [one-definition]
68       --> zero    [[one = zero]]);
69       is-not := (!chain-> [true ==> ((S zero) =/= zero)
70         ((S not-zero))])
71       (labsurd is is-not))
72
73 # One direction of S-injective is the second proposition
74 # returned by (datatype-axioms "N")
75 conclude S-injective
76 pick-any m:N n:N
77 let {right := (!chain [{(S m) = (S n)} => (m = n)
78         [(second (datatype-axioms "N"))])];
79     left := assume (m = n)
80     ((chain [{(S m) --> {n} [(m = n)]})
81     (equiv right left))
82
83 # The following is equivalent to another of the propositions
84 # returned by (datatype-axioms "N"), but here we show
85 # it is a theorem.
86 define nonzero-S :=
87    (forall n . n =/= zero ==> (exists m . n = (S m)))
88 define S-not-same := (forall n . (S n) =/= n)
89 by-induction nonzero-S {
90    zero =>
91    assume (zero =/= zero)
92    (!from-complements (exists ?m (zero = (S ?m))))
93    (!reflex zero)
94    (zero =/= zero))
95    | (S m) =>
96    assume ((S m) =/= zero)
97    let _ := (!reflex (S m))
98    (!egen (exists ?m . (S m) = (S ?m)) m)
99 }
100
101 by-induction S-not-same {
102    zero =>
103    conclude ((S zero) =/= zero)
104    (!instance S-not-zero zero)
105    | (S n) =>
106    let {induction-hypothesis := ((S n) =/= n)}
107    (!chain-> {induction-hypothesis
108      => ((S (S n)) =/= (S n) [S-injective])
109    })
110 }
111
112###############################################################################
113#
114# Addition operator, Plus
115#
116# declare +: [N N] -> N [200]
117
118module Plus {
119
120    # Axioms:
121    assert Plus-def := [(n + zero = n)
122      (n + S m = S (n + m))]
123    define [right-zero right-nonzero] := Plus-def
124    #assert right-zero := (forall n . n + zero = n)
125    #assert right-nonzero := (forall m n . n + (S m) = (S (n + m)))
126
127    # Theorems:
128
129    define left-zero := (forall n . zero + n = n)
130    define left-nonzero := (forall n m . (S m) + n = (S (m + n)))
131
132    by-induction left-zero {
133    zero =>
134    conclude (zero + zero = zero)
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 m] + n]] [right-nonzero]
      --> [[[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]
    --> [[[S n] + 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 + zero]]] [right-zero]
| (S n) =>
| pick-any m
let (induction-hypothesis := (forall ?m . ?m + n = n + ?m))
| (!chain [m + (S n)]) [right-nonzero]
| --> (S (m + n)) [right-nonzero]
| --> ((S n) + m) [left-zero]
| --> ((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-zero]
| --> (n + S k) [assumption]
| --> (S (n + k)) [right-zero]
| ==> (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 := (!chain->
| [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 NPlus
| # module N