# Subtraction of natural numbers.

load "nat-less"

extend-module N {
  declare -: [N N] -> N [200]
}

module Minus {

  assert* axioms :=
      [(zero - x = zero)
      (x - zero = x)
      (S x - S y = x - y)]

  define [zero-left zero-right both-nonzero] := axioms

  define Plus-Cancel := (forall y x . y <= x ==> x = (x - y) + y)

  by-induction Plus-Cancel {
    zero =>
      pick-any x
      assume [zero <= x]
      (!sym (!chain ![((x - zero) + zero)
        = (x + zero) [zero-right]
        = x [Plus.right-zero]])
    | (S y) =>
      let [ind-hyp := (forall ?x . y <= ?x ==> ?x = (?x - y) + y)]
      data-type-cases {
        zero =>
          conclude
        | (S x) =>
          let {ind-hyp := (forall ?x . y <= ?x ==> ?x = (?x - y) + y)}
          datatype-cases {
            zero =>
              conclude
            | (S x) =>
              assume [S y <= S x]
              let [C := (!chain ![A => (y <= x) [Less=.injective]])]
              (!sym ![chain ![((S x - S y) + S y)
                = ((x - y) + S y) [both-nonzero]
                = (S ((x - y) + y)) [Plus.right-nonzero]
                = (S x) [C ind-hyp]])
          }
      }

  define second-equal := (forall x . x - x = zero)

  by-induction second-equal {
    zero =>
      !chain ![zero - zero = zero [zero-left]]
    | (S x) =>
      let [ind-hyp := (x - x = zero)]
      !chain ![S x - S x = (x - x) [both-nonzero]
        = zero [ind-hyp]]

  }

  # Or, without using induction:
  conclude second-equal
  pick-any x:N
  !chain-> [true
define second-greater := (forall x y . x < y ==> x - y = zero)

by-induction second-greater {
  zero =>
  conclude (forall ?y . zero < ?y ==> zero - ?y = zero)
  pick-any y
  assume (zero < y)
  (!chain [(zero - y) = zero [zero-left]])
  | (S x) =>
  let (ind-hyp := (forall ?y . x < ?y ==> x - ?y = zero))
  datatype-cases (forall ?y . S x < ?y ==> S x - ?y = zero)
  |
  | zero =>
  assume A := (S x < zero)
  (!from-complements (S x - zero = zero)
  | (S y) =>
  assume A := (S x < S y)
  let C := (!chain-> [A ==> (x < y) [Less.injective]])
  (!chain [(S x - S y) = (x - y) [both-nonzero] = zero [C ind-hyp]])
  |
  |
  define second-greater-or-equal :=
  (forall x y . x <= y ==> x - y = zero)

conclude second-greater-or-equal
pick-any x:N y
assume A := (x <= y)
let {goal := ((x - y) < x)}
| !by-contradiction goal
  assume (~ goal)
  (!absurd [Less.Plus-k])
define Plus-Minus-property :=
(forall x y z . x = y + z ==> x - y = z)

conclude Plus-Minus-property

pick-any x y z
assume A := (x = y + z)
let (C1 :=

(!chain->
[A ==> (y <= x) [Less=.k-Less=]]

===> (x = (x - y) + y) [Plus-Cancel];

C2 := (!chain-> [A ==> (x = z + y) [Plus.commutative]]))

(!chain->
[((x - y) + y) = x [C1]

= (z + y) [C2]

==> ((x - y) = z) [Plus.=-cancellation]]

conclude Plus-Minus-property-1 :=
(forall x y z . x = y + z ==> x - z = y)

pick-any x:N y:N z:N
(!chain [(x = y + z)

==> (x = z + y) [Plus.commutative]

==> (x - z = y) [Plus-Minus-property]])

conclude Plus-Minus-property-2 :=
(forall x y z . x + y = z ==> x = z - y)

pick-any x:N y:N z:N
(!chain [(x + y = z)

==> (z = x + y) [sym]

==> (z - y = x) [Plus-Minus-property-1]

==> (x = z - y) [sym]])

conclude Plus-Minus-property-3 :=
(forall x y z . x + y = z ==> y = z - x)

pick-any x:N y:N z:N
(!chain [(x + y = z)

==> (z = x + y) [sym]

==> (z - x = y) [Plus-Minus-property]

==> (y = z - x) [sym]])

define Plus-Minus-properties :=
[Plus-Minus-property Plus-Minus-property-1

Plus-Minus-property-2 Plus-Minus-property-3]

define cancellation := (forall x y . (x + y) - x = y)

conclude cancellation

pick-any x y

![chain->
[(x + y = x + y) ==> ((x + y) - x = y) [Plus-Minus-property]]]

define comparison :=
(forall x y z . z < y & y <= x ==> x - y < x - z)

conclude comparison

pick-any x y z

let (A1 := (z < y); A2 := (y <= x))

assume (A1 & A2)

let (u := (x - y);

v := (x - z);

B1 := (!chain->

[A2 ==> (x = u + y) [Plus-Cancel]]);
B2 := (!chain->
  [(A1 & A2)
    ==> (z < x) [Less=.transitive]
    ==> (z <= x) [Less=.Implied-by-<]
    ==> (x = v + z) [Plus-Cancel]
    ==> (x = z + v) [Plus.commutative]
    ==> (u + y = z + v) [B1]]))

("by-contradiction (u < v)
  assume (* u < v)
  let [C1 := (!chain->
    [(~ u < v) ==> (v <= u) [Less=.trichotomy2]]);
    C2 := (!chain->
      [(z < y) ==> (z + v < y + v) [Less.Plus-k]
       ==> (z + v < v + y) [Plus.commutative]]);
    C3 := (!chain->
      [(v <= u)
       ==> (v + y <= u + y) [Less.Plus]
       ==> (z + v < v + y & v + y <= u + y) [augment]
       ==> (z + v < u + y) [Less=.transitive]
       ==> (u + y /= z + v) [Less.not-equal]]))

  (!absurd B2 C3))

define Times-Distributivity :=
  (forall x y z . x * y - x * z = x * (y - z))

conclude Times-Distributivity

pick-any x y z

("two-cases
  assume A := (z <= y)
  (!chain->
    [(x * y)
     = (x * ((y - z) + z)) [Plus-cancel]
     = (x * (y - z) + x * z) [Times.left-distributive]
     ==> (x * z + x * (y - z)) [Plus.commutative]
     ==> (x * y - x * z = x * (y - z))
     [Plus-Minus-property]]))

  assume A := (~ z <= y)
  let [C := (!chain-> [A ==> (y < z) [Less=.trichotomy]])]

  (!combine-equations
   (!chain->
    [C ==> (C | y = z) [alternate]
     ==> (y <= z) [Less=.definition]
     ==> (x * y <= x * z) [Times.<-cancellation-conv]
     ==> (x * y - x * z = zero)
     [second-greater-or-equal]])

  (!chain
   [(x * (y - z))
    = (x * zero) [second-greater]
    = zero [Times.right-zero]]))
}
"