lib/main/nat-times-less.ath

```
load "nat-times.ath"
load "nat-less.ath"

extend-module N {
extend-module Times {

define =-cancellation :=
  (forall y z x . zero < x & x * y = x * z ==> y = z)

by-induction =-cancellation {
  zero =>
    pick-any z x
      assume (zero < x & x * zero = x * z)
        conclude (zero = z)
          let {D := (!chain-last [(x * z)
                                  = (x * zero)     [(x * zero = x * z)]
                                  = zero           [right-zero]
                                  ==> (x = zero | z = zero)
                                  [no-zero-divisors]])}
            (!cases D
              assume (x = zero)
                (!from-complements (zero = z)
                   (x = zero)
                   (!chain-last
                    [(zero < x) ==> (x =/= zero) [Less=.not-equal1]]))
              assume (z = zero)
                (!sym (z = zero)))
| (S y) =>
    let {ind-hyp := (forall ?z ?x . zero < ?x & ?x * y = ?x * ?z ==> y = ?z)}
      datatype-cases (forall ?z ?x .
                          zero < ?x & ?x * (S y) = ?x * ?z ==> (S y) = ?z)
      {
        zero =>
          conclude (forall ?x .
                        zero < ?x & ?x * (S y) = ?x * zero ==> (S y) = zero)
            pick-any x
              assume (zero < x & x * (S y) = x * zero)
                let {C1 := (!chain-last
                             [(x * (S y))
                              = (x * zero) [(x * (S y) = x * zero)]
                              = zero       [right-zero]
                              ==> (x = zero | (S y) = zero)
                              [no-zero-divisors]])}
                  (!cases C1
                    assume (x = zero)
                      (!from-complements ((S y) = zero)
                        (x = zero)
                        (!chain-last
                         [(zero < x) ==> (x =/= zero) [Less=.not-equal1]]))
                    assume ((S y) = zero)
                      (!claim ((S y) = zero)))
      | (S z) =>
          conclude (forall ?x . zero < ?x & ?x * (S y) = ?x * (S z)
                                  ==> (S y) = (S z))
            pick-any x
              assume (zero < x & x * (S y) = x * (S z))
                (!chain-last
                 [(x * y + x)
                  = (x * (S y))  [right-nonzero]
                  = (x * (S z))  [(x * (S y) = x * (S z))]
                  = (x * z + x)
                  ==> (x * y = x * z)   [Plus.=-cancellation]
                  ==> (zero < x & x * y = x * z) [augment]
                  ==> (y = z)                    [ind-hyp]
                  ==> ((S y) = (S z))            [injective]])
      }
}
```

```
define <-cancellation :=
  (forall y z x . zero < x & x * y < x * z ==> y < z)

by-induction <-cancellation {
  zero =>
    pick-any z x
      assume (zero < x & x * zero < x * z)
        (!by-contradiction (zero < z)
          assume A := (~ zero < z)
            let {_ := (!chain-last [A ==> (z = zero) [Less.=zero]])}
              (!absurd
               (!chain-last
                [(x * zero < x * z)
                 ==> (zero < zero)  [right-zero (z = zero)]])
               (!chain-last
                [true ==> (~ zero < zero) [Less.irreflexive]])))
| (S y) =>
    let {ind-hyp := (forall ?z ?x . zero < ?x & ?x * y < ?x * ?z ==> y < ?z)}
      datatype-cases (forall ?z ?x . zero < ?x & ?x * (S y) < ?x * ?z
                                    ==> (S y) < ?z)
      {
        zero =>
          pick-any x
            assume (zero < x  &  x * (S y) < x * zero)
              (!from-complements ((S y) < zero)
                (!chain-last [(x * (S y) < x * zero)
                              ==> (x * (S y) < zero)  [right-zero]])
                (!chain-last [true ==> (~ x * (S y) < zero) [Less.not-zero]]))
      | (S z) =>
          pick-any x
            assume (zero < x  &  x * (S y) < x * (S z))
              conclude ((S y) < (S z))
                (!chain-last
                 [(x * (S y) < x * (S z))
                  ==> (x * y + x < x * z + x) [right-nonzero]
                  ==> (x * y < x * z)         [Less.Plus-cancellation]
                  ==> (y < z)                 [(zero < x) ind-hyp]
                  ==> ((S y) < (S z))         [Less.injective]])
      }
}

define <-cancellation-conv :=
  (forall x y z . zero < x & y < z ==> x * y < x * z)

conclude <-cancellation-conv
  pick-any x y z
    assume A1 := (zero < x & y < z)
      let {goal := (x * y < x * z)}
        (!by-contradiction goal
          assume (~ goal)
            let {D := (!chain-last
                       [(~ goal)
                        ==> (x * z <= x * y)   [Less=.trichotomy2]
                        ==> (x * z < x * y | x * z = x * y)
                                               [Less=.definition]])}
              (!cases D
                assume A2 := (x * z < x * y)
                  (!chain-last
                   [A2 ==> (z < y)             [<-cancellation]
                       ==> (~ y < z)           [Less.asymmetric]
                       ==> (y < z & ~ y < z)   [augment]
                       ==> false               [prop-taut]])
                assume A3 := (x * z = x * y)
                  (!absurd
                   (!chain-last
                    [(zero < x & A3) ==> (z = y) [=-cancellation]])
                   (!chain-last [(y < z)
                                 ==> (~ z = y)   [Less.not-equal1]])))))

define <=-cancellation-conv :=
```

```
  (forall x y z . y <= z ==> x * y <= x * z)


conclude <=-cancellation-conv
  pick-any x y z
    assume A := (y <= z)
      let {goal := (x * y <= x * z)}
        (!two-cases
          assume A1 := (zero < x)
            (!by-contradiction goal
              assume (~ goal)
                (!chain-last
                 [(~ goal)
                  ==> (x * z < x * y)  [Less=.trichotomy1]
                  ==> (z < y)          [A1 <-cancellation]
                  ==> (~ y <= z)       [Less=.trichotomy4]
                  ==> (A & ~ A)        [augment]
                  ==> false            [prop-taut]]))
          assume A2 := (~ zero < x)
            let {C := (!chain-last
                        [true ==> (~ x < zero)       [Less.not-zero]
                              ==> (~ x < zero & A2) [augment]
                              ==> (x = zero)         [Less.trichotomy1]])}
              (!chain-first [goal <== (zero * y <= zero * z)   [C]
                                  <== (zero <= zero)   [left-zero]
                                  <== true             [Less=.reflexive]]))

define identity-lemma1 :=
  (forall x y . zero < x & x * y = x ==> y = one)
define identity-lemma2 :=
  (forall x y . x * y = one ==> x = one)

conclude identity-lemma1
  pick-any x y
    assume (zero < x & x * y = x)
      (!chain-last
       [(x * y = x)
        ==> (x * y = x * one)    [right-one]
        ==> (y = one)            [(zero < x) =-cancellation]])

conclude identity-lemma2
  pick-any x y
    assume A := (x * y = one)
      let {C1 := (!by-contradiction (x =/= zero)
                   assume (x = zero)
                     (!absurd
                      (!chain-last
                       [true ==> (zero * y = zero)  [left-zero]
                             ==> (x * y = zero)     [(x = zero)]
                             ==> (one = zero)       [A]])
                      (!chain-last
                       [true ==> (one =/= zero)     [one-not-zero]])));
           C2 := (!by-contradiction (y =/= zero)
                   assume (y = zero)
                     (!absurd
                      (!chain-last
                       [true ==> (x * zero = zero)  [right-zero]
                             ==> (x * y = zero)     [(y = zero)]
                             ==> (one = zero)       [A]])
                      (!chain-last
                       [true ==> (one =/= zero)     [one-not-zero]])));
           C3 := (!by-contradiction (~ one < x)
                   assume (one < x)
                     let {_ := (!chain-last
                                [C2 ==> (zero < y)   [Less.zero<]])}
                       (!absurd
                        (!chain-last
                         [(one < x)
                     ==> (zero < y & one < x)       [augment]
                     ==> (y * one < y * x)  [<-cancellation-conv]
```

```
                       ==> (one * y < x * y) [commutative]
                       ==> (one * y < one)   [A]
                       ==> (y < (S zero))    [left-one one-definition]
                       ==> (y < (S zero) & y =/= zero) [augment]
                       ==> (y < zero)        [Less.S-step]])
                         (!chain-last
                          [true ==> (~ y < zero) [Less.not-zero]])));
           C4 := (!chain-last
                   [C3 ==> (~ (S zero) < x)  [one-definition]
                       ==> (x <= (S zero))   [Less=.trichotomy2]
                       ==> (x < (S zero) | x = (S zero))  [Less=.definition]])}
         (!by-contradiction (x = one)
           assume A := (x =/= one)
             (!absurd
              (!chain-last
               [A ==> (C4 & A)              [augment]
                  ==> (C4 & x =/= (S zero))  [one-definition]
                  ==> (x < (S zero))         [prop-taut]
                  ==> (x =/= zero & x < (S zero))  [augment]
                  ==> (x < zero)             [Less.S-step]])
              (!chain-last
               [true ==> (~ x < zero)        [Less.not-zero]])))

} # Times
} # N
```