lib/main/transitive-closure.ath

```
#......................................................................
#
# Transitive Closure

load "order"
load "nat-plus"

module Transitive-Closure {
open Irreflexive
open Strict-Partial-Order
overload + N.+
declare R+, R*: (S) [S S] -> Boolean
declare R**: (S) [N S S] -> Boolean
define [x y z y' m n] := [?x:'S ?y:'S ?z:'S ?y':'S ?m:N ?n:N]
define R**-zero :=
  (forall x y . (R** zero x y) <==> x = y)
define R**-nonzero :=
  (forall x n y .
     (R** (S n) x y) <==> (exists z . (R** n x z) & z R y))
define R+-definition :=
  (forall x y . x R+ y <==> (exists n . (R** (S n) x y)))
define R*-definition :=
  (forall x y . x R* y <==> (exists n . (R** n x y)))
define theory :=
  (make-theory
    [Irreflexive.theory
     (adapt-theory Strict-Partial-Order.theory |{R := R+}|)]
    [R**-zero R**-nonzero R+-definition R*-definition])
define R**-sum :=
  (forall n m x y z .
    (R** m x y) & (R** n y z) ==> (R** (m + n) x z))
define RR+-inclusion :=  (forall x y . x R y ==> x R+ y)
define R+R*-inclusion := (forall x y . x R+ y ==> x R* y)
define R+-lemma :=
  (forall x y .
     x R+ y <==> x R y |
     (exists y' . x R+ y' & y' R y))
define R*-lemma := (forall x y . x R* y <==> x = y | x R+ y)
define R*-Reflexive := (forall x . x R* x)
define TC-Transitivity :=
  (forall x y z . x R+ y & y R+ z  ==> x R+ z)
define TC-Transitivity1 :=
  (forall x y z . x R+ y & y R z ==> x R+ z)
define TC-Transitivity2 :=
  (forall x y z . x R y & y R z  ==> x R+ z)
define TC-Transitivity3 :=
  (forall x y z . x R* y & y R* z  ==> x R* z)
define theorems := [R**-sum TC-Transitivity RR+-inclusion R+R*-inclusion
                    R+-lemma R*-lemma R*-Reflexive TC-Transitivity1
                    TC-Transitivity2 TC-Transitivity3]
define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [R R+ R* R**] := (adapt [R R+ R* R**])}
    match theorem {
      (val-of R**-sum) =>
      by-induction (adapt theorem) {
        zero =>
        pick-any m x y z
          let {A1 := (R** m x y);
               A2 := (R** zero y z)}
          assume (A1 & A2)
            let {B := (!chain-> [A2 ==> (y = z)  [R**-zero]])}
            (!chain->
             [A1 ==> (R** m x z)          [B]
                 ==> (R** (m + zero) x z) [N.Plus.right-zero]])
      | (S n) =>
        let {ind-hyp := (forall ?m ?x ?y ?z .
```

```
                              (R** ?m ?x ?y) & (R** n ?y ?z) ==>
                              (R** (?m + n) ?x ?z))}
        pick-any m x y z
          let {A1 := (R** m x y);
               A2 := (R** (S n) y z)}
          assume (A1 & A2)
            let {B := (!chain->
                       [A2 ==> (exists ?y' . (R** n y ?y') & ?y' R z)
                                                   [R**-nonzero]])}
            pick-witness y' for B
              let {B-w1 := (R** n y y');
                   B-w2 := (y' R z)}
              (!chain->
               [(A1 & B-w1)
                ==> (R** (m + n) x y')             [ind-hyp]
                ==> ((R** (m + n) x y') & B-w2)    [augment]
                ==> (exists ?y' . (R** (m + n) x ?y') & ?y' R z)
                                                   [existence]
                ==> (R** (S (m + n)) x z)          [R**-nonzero]
                ==> (R** (m + (S n)) x z)          [N.Plus.right-nonzero]])
      }
   | (val-of R*-Reflexive) =>
     let {sort := (sort-of (first (qvars-of (adapt theorem))))}
     pick-any x:sort
       (!chain->
        [(x = x)
         ==> (R** zero x x)              [R**-zero]
         ==> (exists n . (R** n x x)) [existence]
         ==> (x R* x)                    [R*-definition]])
   | (val-of TC-Transitivity) =>
     pick-any x y z
       let {A1 := (x R+ y);
            A2 := (y R+ z)}
       assume (A1 & A2)
         let {B1 := (!chain->
                     [A1 ==> (exists m . (R** (S m) x y))
                         [R+-definition]]);
              B2 := (!chain->
                     [A2 ==> (exists n . (R** (S ?n) y z))
                         [R+-definition]]);
              _ := (!prove R**-sum)}
         pick-witness m for B1 B1-w
         pick-witness n for B2 B2-w
           (!chain->
            [(B1-w & B2-w)
             ==> (R** ((S m) + (S n)) x z)      [R**-sum]
             ==> (R** (S (m + (S n))) x z)      [N.Plus.left-nonzero]
             ==> (exists ?k . (R** (S ?k) x z)) [existence]
             ==> (x R+ z)                       [R+-definition]])
  | (val-of RR+-inclusion) =>
    pick-any x y
      (!chain
       [(x R y)
        ==> (x = x & x R y)                   [augment]
        ==> ((R** zero x x) & x R y)          [R**-zero]
        ==> (exists ?x' . (R** zero x ?x') & ?x' R y) [existence]
        ==> (R** (S zero) x y)                [R**-nonzero]
        ==> (exists ?k . (R** (S ?k) x y))    [existence]
        ==> (x R+ y)                          [R+-definition]])
  | (val-of R+-lemma) =>
    pick-any x y
     (!equiv
      assume A := (x R+ y)
        let {B := (!chain->
                   [A ==> (exists ?k . (R** (S ?k) x y)) [R+-definition]])}
        pick-witness k for B B-w
          let {C := (!chain->
                     [B-w ==> (exists ?x' . (R** k x ?x') & ?x' R y)
                          [R**-nonzero]])}
          pick-witness x' for C C-w
```

```
            (!two-cases
             assume D := (k = zero)
               let {E :=
                    (!chain->
                     [C-w ==> ((R** zero x x') & x' R y)     [D]
                          ==> (R** zero x x')                [left-and]
                          ==> (x = x')                       [R**-zero]])}
               (!chain->
                [C-w ==> (x' R y)                             [right-and]
                     ==> (x R y)                              [(x = x')]
                     ==> (x R y | (exists ?y' . x R+ ?y' & ?y' R y))
                     [alternate]])
             assume D := (k =/= zero)
               let {E :=
                    (!chain->
                     [D ==> (exists ?k' . k = (S ?k'))  [N.nonzero-S]])}
               pick-witness k' for E E-w
                 let {F := (!chain-> [C-w ==> (x' R y)  [right-and]])}
                 (!chain->
                  [C-w ==> ((R** (S k') x x') & x' R y) [E-w]
                       ==> (R** (S k') x x')            [left-and]
                       ==> (exists ?k' . (R** (S ?k') x x'))
                       [existence]
                       ==> (x R+ x')                    [R+-definition]
                       ==> (x R+ x' & F)                [augment]
                       ==> (exists ?x' . x R+ ?x' & ?x' R y) [existence]
                       ==> (x R y | (exists ?x' . x R+ ?x' & ?x' R y))
                       [alternate]]))
    assume A := (x R y | (exists ?y' . x R+ ?y' & ?y' R y))
      let {RRI := (!prove RR+-inclusion)}
      (!cases A
       (!chain [(x R y) ==> (x R+ y)  [RRI]])
       assume B := (exists ?y' . x R+ ?y' & ?y' R y)
         pick-witness y' for B B-w
           let {C :=
            (!chain->
             [B-w ==> (x R+ y')       [left-and]
                  ==> (exists ?k . (R** (S ?k) x y'))  [R+-definition]])}
           pick-witness k for C C-w
             (!chain->
              [C-w ==> ((R** (S k) x y') & y' R y) [augment]
                   ==> (exists ?y' . (R** (S k) x ?y') & ?y' R y)
                   [existence]
                   ==> (R** (S (S k)) x y)         [R**-nonzero]
                   ==> (exists ?k . (R** (S ?k) x y))  [existence]
                   ==> (x R+ y)                    [R+-definition]])))
  | (val-of R*-lemma) =>
    let {sort := (sort-of (first (qvars-of (adapt theorem))))}
    pick-any x:sort y:sort
      (!equiv
        assume A := (x R* y)
          let {B := (!chain->
                     [A ==> (exists ?n . (R** ?n x y))  [R*-definition]])}
          pick-witness n for B B-w
            (!two-cases
             assume C1 := (n = zero)
               (!chain->
                [B-w ==> (R** zero x y)     [C1]
                     ==> (x = y)            [R**-zero]
                     ==> (x = y | x R+ y)   [alternate]])
             assume C2 := (n =/= zero)
               let {D := (!chain-> [C2 ==> (exists ?m . n = S ?m)
                                              [N.nonzero-S]])}
               pick-witness m for D D-w
                 (!chain->
                  [B-w ==> (R** (S m) x y) [D-w]
                       ==> (exists ?m . (R** (S ?m) x y)) [existence]
                       ==> (x R+ y)          [R+-definition]
                       ==> (x = y | x R+ y) [alternate]]))
         assume A := (x = y | x R+ y)
```

```
            (!cases A
              assume A1 := (x = y)
                (!chain->
                 [A1 ==> (R** zero x y)               [R**-zero]
                     ==> (exists ?n . (R** ?n x y))  [existence]
                     ==> (x R* y)                     [R*-definition]])
              assume A2 := (x R+ y)
                let {B :=
                       (!chain->
                        [A2 ==> (exists ?n . (R** (S ?n) x y))
                                                      [R+-definition]])}
                pick-witness n for B B-w
                  (!chain->
                   [B-w ==> (exists ?k . (R** ?k x y)) [existence]
                        ==> (x R* y)                   [R*-definition]])))
  | (val-of R+R*-inclusion) =>
    let {R*L := (!prove R*-lemma)}
    pick-any x y
      (!chain
       [(x R+ y)
        ==> (x = y | x R+ y)      [alternate]
        ==> (x R* y)              [R*L]])
  | (val-of TC-Transitivity1) =>
    pick-any x y z
      let {A1 := (x R+ y);
           A2 := (y R z);
           R+L := (!prove R+-lemma)}
      assume (A1 & A2)
        (!chain->
         [(A1 & A2)
          ==> (exists ?y . x R+ ?y & ?y R z) [existence]
          ==> (x R z | (exists ?y . x R+ ?y & ?y R z))
                                      [alternate]
          ==> (x R+ z)                [R+L]])
  | (val-of TC-Transitivity2) =>
    pick-any x y z
      let {A1 := (x R y);
           A2 := (y R z);
           R+-transitive := ((renaming |{R := R+}|) transitive);
           RR+I := (!prove RR+-inclusion)}
      assume (A1 & A2)
        (!chain->
         [A1 ==> (x R+ y)              [RR+I]
             ==> (x R+ y & A2)         [augment]
             ==> (x R+ y & y R+ z)     [RR+I]
             ==> (x R+ z)              [R+-transitive]])
  | (val-of TC-Transitivity3) =>
    let {sort := (sort-of (first (qvars-of (adapt theorem))))}
    pick-any x:sort y:sort z:sort
      let {A1 := (x R* y);
           A2 := (y R* z);
           RRI := (!prove R+R*-inclusion);
           R*L := (!prove R*-lemma)}
      assume (A1 & A2)
        let {B1 := (!chain->
                      [A1 ==> (x = y | x R+ y) [R*L]]);
             B2 := (!chain->
                      [A2 ==> (y = z | y R+ z) [R*L]])}
        (!cases B1
          assume C1 := (x = y)
            (!cases B2
             assume D1 := (y = z)
               (!chain->
                [x = y [C1] = z [D1]
                   ==> (x = z | x R+ z) [alternate]
                   ==> (x R* z)         [R*L]])
             assume D2 := (y R+ z)
               (!chain->
                [D2 ==> (x R+ z)     [C1]
                    ==> (x R* z)     [RRI]]))
```

```
            assume C2 := (x R+ y)
              (!cases B2
                assume D1 := (y = z)
                  (!chain->
                   [C2 ==> (x R+ z)      [D1]
                       ==> (x R* z)      [RRI]])
                assume D2 := (y R+ z)
                  (!chain->
                   [D2 ==> (C2 & D2)    [augment]
                       ==> (x R+ z)      [TC-Transitivity]
                       ==> (x = z | x R+ z)   [alternate]
                       ==> (x R* z)      [R*L]])))

   }
(add-theorems theory |{theorems := proofs}|)
} # close module Transitive-Closure
```