# Abstract-level order concepts and theorems

# Strict Partial Order

module Binary-Relation {
    declare R, R': (T) [T T] -> Boolean
    define [x y z] := [?x ?y ?z]
    define inverse-def := (forall x y . x R' y <==> y R x)
    define theory := (make-theory [] [inverse-def])
}

module Irreflexive {
    open Binary-Relation
    define irreflexive := (forall x . ~ x R x)
    define theory := (make-theory ['Binary-Relation'] [irreflexive])
    define inverse := (forall x y z . x R y & y R z ==> x R z)
    define proof :=
        method (theorem adapt)
            let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
                [R R'] := (adapt [R R'])}
            match theorem {
                (val-of inverse) =>
                    pick-any x
                    (!chain-> [true ==> (~ x R x) [irreflexive]
                        ==> (~ x R' x) [inverse-def]])
            }
        (add-theorems theory [{inverse := proof}])
}

module Transitive {
    open Binary-Relation
    define transitive := (forall x y z . x R y & y R z ==> x R z)
    define theory := (make-theory ['Binary-Relation'] [transitive])
    define inverse := (forall x y z . x R y & y R z ==> x R z)
    define proof :=
        method (theorem adapt)
            let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
                [R R'] := (adapt [R R'])}
            match theorem {
                (val-of inverse) =>
                    pick-any x y z
                    (!chain [x R' y & y R' z]
                        ==> [y R x & z R y] [inverse-def]
                        ==> [z R y & y R x] [and-comm]
                        ==> [z R x] [transitive]
                        ==> [x R' z] [inverse-def]])
            }
        (add-theorems theory [{inverse := proof}])
}

module Strict-Partial-Order {
    open Irreflexive, Transitive
    define theory := (make-theory ['Irreflexive 'Transitive'] [])
    define asymmetric := (forall x y . x R y ==> ~ y R x)
    define implies-not-equal := (forall x y . x R y ==> x /= y)
    define proofs :=
        method (theorem adapt)
            let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
                [R R'] := (adapt [R R'])}
            match theorem {
                (val-of asymmetric) =>
                    pick-any x y
                    assume (x R y)
                    (!by-contradiction (~ y R x))
assume (y R x)
!absurd
!chain-> [(y R x) [augment]
  ==> (x R x) [transitive]]
| (val-of implies-not-equal) =>
pick-any x y
assume (x R y)
||by-contradiction (x /= y)
assume (x = y)
let {xRx := (!chain-> [(x R y) [transitive]])
  -xRx := (!chain-> [true [irreflexive]])
}| (!absurd xRx -xRx)}
(add-theorems theory |{asymmetric implies-not-equal} := proofs|)
#
module Reflexive {
  open Binary-Relation
  define reflexive := (forall x . x R x)
  define theory := (make-theory ['Binary-Relation] [reflexive])
  define inverse := (forall x y . x R y & y R x ==> x = y)
  define proof :=
    method (theorem adapt)
      let { [get prove chain chain-> chain<->] := (proof-tools adapt theory);
        [R R'] := (adapt [R R'])}
      match theorem {
      (val-of inverse) =>
pick-any x
      | (chain-> [true ==> (x R x) [reflexive]]
        ==> (x R' x) [inverse-def])
      (!absurd xRx -xRx)}
      (add-theorems theory |{inverse := proof}|)
    }
}
module Preorder {
  open Transitive, Reflexive
  define theory := (make-theory ['Transitive 'Reflexive] [])
}
#
module Antisymmetric {
  open Binary-Relation
  define antisymmetric := (forall x y . x R y & y R x ==> x = y)
  define theory := (make-theory ['Binary-Relation] [antisymmetric])
  define inverse := (forall x y . x R' y & y R' x ==> x = y)
  define proof :=
    method (theorem adapt)
      let { [get prove chain chain-> chain<->] := (proof-tools adapt theory);
        [R R'] := (adapt [R R'])}
      match theorem {
      (val-of inverse) =>
pick-any x y
      | (chain-> [(x R' y & y R' x)
        ==> (y R x & x R y) [inverse-def]]
        ==> (x R y & y R x) [and-comm]
        ==> (x = y) [antisymmetric])
      } (add-theorems theory |{inverse := proof}|)
    }
module Partial-Order {
  open Preorder, Antisymmetric
  define theory := (make-theory ['Preorder 'Antisymmetric] [])
}

# SPO: Strict Partial Order with < instead of R, > instead of R'
module SPO {
  declare <, >: (T) [T T] -> Boolean
  define sm := |{Binary-Relation.R := <, Binary-Relation.R' := >}| |
  define renaming := (renaming sm)
  define theory := (adapt-theory 'Strict-Partial-Order sm)
}

# PO: Partial Order with <= instead of R, >= instead of R'
module PO {
  declare <=, >=: (T) [T T] -> Boolean
  define sm := |{Binary-Relation.R := <=, Binary-Relation.R' := =>}| |
  define renaming := (renaming sm)
  define theory := (adapt-theory 'Partial-Order sm)
}

# Show that if we start with SPO.theory and add a definition of <=, we
# can derive the axioms of PO.theory as theorems of SPO.theory.
module PO-from-SPO {
  define \[x y z\] := [?x ?y ?z]
  define \[< > <= >=\] := [SPO.< SPO.> PO.<= PO.>=]
  define <=-definition := (forall x y . x <= y <==> x < y | x = y)
  define >=-definition := (forall x y . x >= y <==> x > y | x = y)
  (add-axioms 'SPO [<=-definition >=-definition])
  define implied-by-less := (forall x y . x <= y ==> x <= y)
  define implied-by-equal := (forall x y . x = y ==> x <= y)
  define implies-not-reverse := (forall x y . x <= y ==> ~ y < x)
  define PO-inverse := (forall x y . x <= y ==> y <= x)
  define PO-reflexive := (forall x . x <= x)
  define PO-transitive := (forall x y z . x <= y & y <= z ==> x <= z)
  define PO-antisymmetric := (forall x y . x <= y & y <= x ==> x = y)
  define theorems := [<=-definition implied-by-less implied-by-equal
    implies-not-reverse PO-inverse PO-reflexive
    PO-antisymmetric PO-transitive]
  define proofs :=
    method (theorem adapt)
      let {adapt := (o adapt SPO.renaming);
        [get prove chain chain<-] := (proof-tools adapt SPO.theory);
        [< > <= >] := (adapt [< > <= >]);
        inverse-def := Strict-Partial-Order.inverse-def;
        irreflexive := Strict-Partial-Order.irreflexive;
        transitive := Strict-Partial-Order.transitive;
        asymmetric := ([prove Strict-Partial-Order.asymmetric])}
      match theorem {
        (val-of implied-by-less) =>
          pick-any x y
            (\(chain \{ (x < y) ==> (x < y | x = y) \}) alternate
            => (x <= y) [<=-definition])
        (val-of implied-by-equal) =>
          pick-any x y
\begin{verbatim}
/*chain [(x = y) ==> (x < y | x = y) [alternate]
  ==> (x <= y) [<=-definition]]

| (val-of implies-not-reverse) =>

pick-any x y
  assume A := (x <= y)
  let \{B := \{!chain- \[A \Rightarrow (x < y | x = y) [\leq-definition]\]\}\}
  \{cases B
  \{!chain [(x < y) \Rightarrow (y < x) [asymmetric]]
  assume \(x = y\)
  \{by-contradiction (y < x)
   assume \(y < x\)
   let \(is := \{!chain- \[(y < x) \Rightarrow (y < x) [\sim-\text{asymmetric}]\]\}\)
   is-not := \{!chain-> \{true \Rightarrow (\sim y < y) [\text{irreflexive}]\]\}\)
   (!absurd is is-not))
  | (val-of PO-inverse) =>
    pick-any x y
    \{!chain \[(x >= y) \iff (x > y | x = y) [\geq-definition]
    \iff (y < x | y = x) [\text{inverse-def sym]}
    \iff (y <= x) [\leq-definition]\}\}
  | (val-of PO-reflexive) =>
    pick-any x
    \{!chain \[(x = x) \Rightarrow (x <= x) [\text{IBE}]\]\}
  | (val-of PO-antisymmetric) => (!force (adapt theorem))
  | (val-of PO-transitive) => (!force (adapt theorem))
  }
  (add-theorems SPO.theory \{[theorems := proofs]\})

}|}

extend-module PO-from-SPO {

define proofs :=
  method (theorem adapt)
    let \{adapt := (o adapt PO.renaming);
    \{get prove chain chain-> chain<\} := (proof-tools adapt SPO.theory);
    \(< \leq \} := (adapt [< \leq \});
    \text{irreflexive} := \text{Strict-Partial-Order.irreflexive};
    \text{transitive} := \text{Strict-Partial-Order.transitive};
    \text{asymmetric} := (!prove \text{Strict-Partial-Order.asymmetric})

match theorem {
  \{val-of PO-antisymmetric\} =>
    pick-any x y
    assume \(x <= y \& y <= x\)
    let \{disj1 := \{!chain- \[(x <= y) \Rightarrow (x < y | x = y) [\leq-definition]\]\}\}
    disj2 := \{!chain- \[(y <= x) \Rightarrow (y < x | y = x) [\leq-definition]\]\}\)
    \{cases disj1
    assume \(x < y\)
    \{cases disj2
    assume \(y < x\)
    \{from-complements \(x = y\)
    \(y < x\)
    \{chain- \[(x < y) \Rightarrow (\sim y < x) [\text{asymmetric}]\]\}\)
    assume \(y = x\)
    \{sym \(y = x\)\}
    assume \(x = y\)
    \{claim \(x = y\)\}
  | (val-of PO-transitive) =>
    pick-any x y z
    assume \(x <= y \& y <= z\)
    let \{disj1 := \{!chain- \[(x <= y) \Rightarrow (x < y | x = y) [\leq-definition]\]\}\)
    disj2 := \{!chain- \[(y <= z) \Rightarrow (y < z | y = z) [\leq-definition]\]\}\)
    \text{by-less} := (!prove \text{implied-by-less});
    \text{by-equal} := (!prove \text{implied-by-equal})
    \{cases disj1
\end{verbatim}
assume (x < y)
  (!cases disj2
    assume i := (y < z)
    (!chain->
      [i ===> (x < y & y < z) [augment]
      ===> (x < z) [transitive]
      ===> (x <= z) [by-less]])
    assume ii := (y = z)
    (!chain->
      [(x < y) ===> (x < z) [ii]
      ===> (x <= z) [by-less]])
  )
assume (x = y)
  (!cases disj2
    assume i := (y < z)
    (!chain->
      [i ==> (x < z) [(x = y)]
      ===> (x <= z) [by-less]])
    assume ii := (y = z)
    (!chain->
      [x --> y [i] --> z [ii]
      ===> (x <= z) [by-equal]])
  )
}

{add-theorems SPO.theory |{[PO-antisymmetric PO-transitive] := proofs}}}

# .........................................................................
# SWO: Strict Weak Order, a refinement of SPO

extend-module SPO {
declare E: (T) [T T] -> Boolean [100]
define E-definition := (forall x y . x E y <=> ~ x < y & ~ y < x)
  (add-axioms theory [E-definition])
}

module SWO {
  open SPO
  define E-transitive := (forall x y z . x E y & y E z ==> x E z)
  define theory := (make-theory [SPO] [E-transitive])
  define E-reflexive := (forall x . x E x)
  define E-symmetric := (forall x y . x E y ==> y E x)
  define <E-transitive-1 := (forall x y z . x < y & y E z ==> x < z)
  define <E-transitive-2 := (forall x y z . x < y & x E z ==> z < y)
  define not-<property := (forall x y . ~ x < y ==> y < x | y E x)
  define <E-transitive-not-1 := (forall x y z . x < y & ~ z < y ==> x < z)
  define <E-transitive-not-2 := (forall x y z . x < y & ~ x < z ==> z < y)
  define <E-transitive-not-3 := (forall x y z . ~ y < x & y < z ==> x < z)
  define not-<is-transitive :=
    (forall x y z . ~ x < y & ~ y < z ==> ~ x < z)
  define <E-theorems :=
    [E-reflexive E-symmetric <E-transitive-1 <E-transitive-2
     not-<property <E-transitive-not-1 <E-transitive-not-2
     <E-transitive-not-3 not-<is-transitive]
  define ren := (get-renaming 'SPO)
  define <E-proofs :=
    method (theorem adapt)
      let [adapt := (o adapt SPO.renaming);
        (get prove chain chain-> chain<-) := (proof-tools adapt theory);
        E := lambda (x y) (adapt (x E y));
        < := lambda (x y) (adapt (x < y));
        irreflexive := Strict-Partial-Order.irreflexive;
        transitive := Strict-Partial-Order.transitive;
        asymmetric := Strict-Partial-Order.asymmetric]
      match theorem {
        (val-of E-reflexive) =>
          pick-any x
      }
\begin{verbatim}
(!chain-> [true
  => (\sim x < x) [irreflexive]
  => (\sim x < x & \sim x < x) [augment]
  => (x <E x) [E-definition]]
| (val-of E-symmetric) =>
  pick-any x y
  assume (x E y)
  (!chain-> [(x E y)
    => (\sim x < y & \sim y < x) [E-definition]
    => (\sim y < x & \sim x < y) [and-comm]
    => (y <E x) [E-definition]])
  | _ => (!force (adapt theorem))
}

(add-theorems theory |{<-E-theorems := <-E-proofs}|
}
# close module SWO

extend-module SWO {
  declare <E: (T) [T T] -> Boolean
  define <E-definition := (forall x y . x <E y <==>
    \sim y < x)
  (add-axioms theory [<E-definition])
}

# Show that <E is a preorder:

extend-module SWO {
  define <E-reflexive := (forall x . x <E x)
  define <E-transitive := (forall x y z . x <E y & y <E z ==> x <E z)
  define theorems := [<E-reflexive <E-transitive]

  define proofs :=
    method (theorem adapt)
    let (adapt := (o adapt SWO.renaming);
      [get prove chain chain-> chain<-] := (proof-tools adapt theory);
      < := lambda (x y) (adapt (x < y));
      <E := lambda (x y) (adapt (x <E y));
      irreflexive := Strict-Partial-Order.irreflexive;
      transitive := Strict-Partial-Order.transitive)
    match theorem {
      (val-of <E-reflexive) =>
        pick-any x
        (!chain-> [true => (\sim x < x) [irreflexive]
          => (x <E x) [E-definition]]
| (val-of <E-transitive) =>
        let (transitive := (!prove not<-is-transitive))
        pick-any x y z
        (!chain [(x <E y & y <E z)
          => (\sim y < x & \sim z < y) [E-definition]
          => (\sim z < x) [transitive]
          => (x <E z) [E-definition]])
    }

  (add-theorems theory |{theorems := proofs}|
}
# close module SWO

#........................................................................
# STO: Strict Total Order theory

module STO {
  open SWO

  define strict-trichotomy := (forall x y . \sim y < x & \sim x < y => x = y)
  define theory := (make-theory [SWO] [strict-trichotomy])

  define E-iff-equal := (forall x y . x E y <=> x = y)
}
# close module STO

extend-module STO {
\end{verbatim}
define proof :=
method (theorem adapt)
let \{adapt := \langle o adapt SPO.renaming \rangle ; \}
\{get proof chain chain-> chain<-> \} := \langle proof-tools adapt theory \rangle ;
E := \lambda (x \ y) \langle adapt \langle x \ E \ y \rangle \rangle ;
< := \lambda (x \ y) \langle adapt \langle x < y \rangle \rangle 
match theorem |
\{\val-of E-iff-equal \} =>
pick-any x y
(!equiv

(!chain [(x E y)]

=> \langle \sim x < y \& \sim y < x \rangle) \ [E-definition]

=> \langle x = y \rangle \ [strict-trichotomy]]

assume \langle x = y \rangle

(!chain-> [true

=> \langle \sim x < x \rangle) \ [Strict-Partial-Order.irreflexive]

=> \langle \sim x < x \& \sim x < x \rangle) \ [augment]

=> \langle x E x \rangle \ [E-definition]

=> \langle x E y \rangle \ [(x = y)]\}

\}

(add-theorems theory |{E-iff-equal := proof}|)

| # close module STO