Topomodels

David Álvarez Lombardi   Paulius Skaisgiris

Sunday 4th June, 2023

Abstract

Finish me.

Contents

1 Introduction

Finish me.

2 Topological Preliminaries

```
{-# LANGUAGE ScopedTypeVariables #-}

module Topology where

import Data.Set (Set, cartesianProduct, elemAt, intersection, isSubsetOf, union, unions,
    (\\), singleton)
import qualified Data.Set as S

import Test.QuickCheck
    ( Arbitrary(arbitrary), Gen, listOf1, elements, oneof, sublistOf, suchThat )
```

This section describes some topological preliminaries which will be necessary for defining TopoModels later on. The definitions are taken from the course slides of Topology, Logic, Learning given by Alexandru Baltag in Spring 2023.

A *topological space* is a pair (X, τ) where X is a nonempty set and $\tau \subseteq \wp(X)$ is a family of subsets of X such that (1) $\varnothing \in \tau$ and $X \in \tau$ (2) τ is closed under finite intersection: if $U, V \in \tau$ then $U \cap V \in \tau$ (3) τ is closed under arbitrary unions: for any subset $A \subseteq \tau$, the union $\bigcup A \in \tau$

Thus, let us first define closure under intersection and closure under unions.

```
unionize :: (Ord a) => Set (Set a) -> Set (Set a)
unionize sets = S.map (uncurry union) (cartesianProduct sets sets)

intersectionize :: (Ord a) => Set (Set a) -> Set (Set a)
intersectionize sets = S.map (uncurry intersection) (cartesianProduct sets sets)

-- The closure definitions defined below are finite, but it is sufficient for our purposes
-- since we will only work with finite models.

closeUnderUnion :: (Ord a) => Set (Set a) -> Set (Set a)
closeUnderUnion sets = do
    let oneUp = unionize sets
    if sets == oneUp
        then sets
        else closeUnderUnion oneUp

closeUnderIntersection :: (Ord a) => Set (Set a) -> Set (Set a)
closeUnderIntersection sets = do
    let oneUp = intersectionize sets
    if sets == oneUp
        then sets
        else closeUnderIntersection oneUp
```

Here we initialise a few sets to test our implementations going forward.

```
ghci> (s0 :: Set Int) = S.fromList [1]
ghci> (s1 :: Set Int) = S.fromList [2]
ghci> (s2 :: Set Int) = S.fromList [3, 4]
ghci> (s3 :: Set Int) = S.fromList [1, 2, 3]
ghci> (s4 :: Set Int) = S.fromList [2, 3]
ghci> (s5 :: Set Int) = S.fromList [3, 4]
ghci> (s6 :: Set Int) = S.fromList [1, 2]
ghci> (s7 :: Set Int) = S.fromList [1, 3]
```

Here we provide some examples of closure under intersections and unions.

```
ghci> closeUnderUnion $ S.fromList [s0, s1, s2]
fromList [fromList [1],fromList [1,2],fromList [1,2,3,4],fromList [1,3,4],fromList [2],
    fromList [2,3,4],fromList [3,4]]

ghci> closeUnderIntersection $ S.fromList [s0, s1, s2]
fromList [fromList [],fromList [1],fromList [2],fromList [3,4]]

ghci> closeUnderUnion $ S.fromList [s3, s4, s5]
fromList [fromList [1,2,3],fromList [1,2,3,4],fromList [2,3],fromList [2,3,4],fromList
    [3,4]]

ghci> closeUnderIntersection $ S.fromList [s3, s4, s5]
fromList [fromList [1,2,3],fromList [2,3],fromList [3],fromList [3,4]]

ghci> topology = (closeUnderUnion . closeUnderIntersection) $ S.fromList [s5, s6, s7]
ghci> topology
fromList [fromList [],fromList [1],fromList [1,2],fromList [1,2,3],fromList [1,2,3,4],
    fromList [1,3],fromList [1,3,4],fromList [3],fromList [3,4]]

$
```

Now, we can define a topological space in Haskell.

```
data TopoSpace a
    = TopoSpace
        (Set a) -- Carrier set
        (Set (Set a)) -- Topology
    deriving (Eq, Show)
```

Now, let us implement an instance for `Arbitrary` for it. To do so, we will reimplement some functions from ‘QuickCheck‘ for Sets.

```
-- Inspired by https://stackoverflow.com/a/35529208

setOneOf :: Set (Gen a) -> Gen a
setOneOf = oneof . S.toList

subsetOf :: (Arbitrary a, Ord a) =>  Set a -> Gen (Set a)
subsetOf =  fmap S.fromList . sublistOf .  S.toList

setOf1 :: (Arbitrary a, Ord a) => Gen a -> Gen (Set a)
setOf1 = fmap S.fromList . listOf1

setElements :: Set a -> Gen a
setElements = elements . S.toList

isOfSizeAtMost :: Set a -> Int -> Bool
isOfSizeAtMost set s = S.size set <= s

instance (Arbitrary a, Ord a) => Arbitrary (TopoSpace a) where
  arbitrary = do
    (x'::Set a) <- arbitrary `suchThat` (`isOfSizeAtMost` 10)
    (randElem:: a) <- arbitrary
    -- Make sure x is not empty, otherwise we get an error because of `setElements`
    let x = x' `S.union` S.singleton randElem
    -- Put an artificial bound on the size of the set, otherwise it takes too long to "fix"
         the topology
    subbasis <- let basis = setOf1 (setElements x) `suchThat` (`isOfSizeAtMost` 3)
                  in setOf1 basis `suchThat` (`isOfSizeAtMost` 3)
    let someTopoSpace = TopoSpace x subbasis
    return (fixTopoSpace someTopoSpace)
```

Let’s implement some convenience functions. The first one simply checks if the input `TopoSpace` object respects all the topology axioms. The second one will fixed any given (potentially broken)

`TopoSpace` to have the necessary axioms.

```
isTopoSpace :: (Ord a) => TopoSpace a -> Bool
isTopoSpace (TopoSpace sp topo) | S.empty `notElem` topo = False
                                | sp `notElem` topo = False
                                | not (unions topo `isSubsetOf` sp) = False
                                | otherwise = topo == (closeUnderUnion .
                                  closeUnderIntersection $ topo)

fixTopoSpace :: (Ord a) => TopoSpace a -> TopoSpace a
fixTopoSpace (TopoSpace sp topo)
  -- Throw an error since we don't know how the topology should look like
  | not (S.unions topo `isSubsetOf` sp) = error "topology not a subset of the powerset of
      the space"
  | S.empty `notElem` topo = fixTopoSpace (TopoSpace sp (topo `union` S.singleton S.empty))
  | sp `notElem` topo = fixTopoSpace (TopoSpace sp (topo `union` singleton sp))
  | otherwise = let verifTopo = closeUnderUnion . closeUnderIntersection $ topo
                in TopoSpace sp verifTopo
```

Examples of using the above:

```
ghci> isTopoSpace (TopoSpace (arbUnion $ S.fromList [s5, s6, s7]) topology)
ghci> True

ghci> badTS = TopoSpace (S.fromList [1,2,3]) (S.fromList [S.fromList [1,2], S.fromList
    [2,3]])
ghci> isTopoSpace badTS
ghci> False

ghci> goodTS = fixTopoSpace badTS
ghci> isTopoSpace goodTS
ghci> True

ghci> isTopoSpace (fixTopoSpace goodTS)
ghci> True

ghci> fixTopoSpace (TopoSpace (S.fromList [1,2,3]) topology)
ghci> error "topology not a subset of the powerset of the space"
$
```

The elements of τ are called *open sets* or *opens*. Given a point $x \in X$, we call the set of all opens containing x the *open neighbourhoods of* x.

```
isOpenIn :: (Eq a) => Set a -> TopoSpace a -> Bool
isOpenIn set (TopoSpace _ opens) = set `elem` opens

openNbds :: (Eq a) => a -> TopoSpace a -> Set (Set a)
openNbds x (TopoSpace _ opens) = S.filter (x `elem`) opens
```

A set $C \subseteq X$ is called a *closed set* if it is the complement of an open set, i.e., $C = X \setminus U$ for some $U \in \tau$.

We let $\overline{\tau} := \{X \setminus U \mid U \in \tau\}$ denote the family of all closed sets of (X, τ).

```
closeds :: (Ord a) => TopoSpace a -> Set (Set a)
closeds (TopoSpace space opens) = S.map (space \\) opens

isClosedIn :: (Eq a, Ord a) => Set a -> TopoSpace a -> Bool
isClosedIn set topoSpace = set `elem` closeds topoSpace
```

A subset $S \subseteq X$ is called *clopen* if it is both closed and open, i.e. $A \in \tau$ and $A \in \overline{\tau}$.

```
isClopenIn :: (Eq a, Ord a) => Set a -> TopoSpace a -> Bool
isClopenIn set topoSpace = set `isOpenIn` topoSpace && set `isClosedIn` topoSpace
```

Examples of using the above:

```
ghci> topoSpace = TopoSpace (S.fromList [1, 2, 3, 4]) topology

ghci> closeds topoSpace
fromList [fromList [],fromList [1,2],fromList [1,2,3,4],fromList [1,2,4],fromList [2],
    fromList [2,3,4],fromList [2,4],fromList [3,4],fromList [4]]

ghci> openNbds 2 topoSpace
fromList [fromList [1,2],fromList [1,2,3],fromList [1,2,3,4]]

ghci> S.fromList [1] `isOpenIn` topoSpace
True
ghci> S.fromList [1] `isClosedIn` topoSpace
False
ghci> S.fromList [] `isClopenIn` topoSpace
True
```

Given some topological space $\mathbf{X} := (X, \tau)$, a *basis* for $\mathbf{X}$ is a subset $\beta \subseteq \tau$ such that τ is equal to the closure of β under arbitrary unions.

A *subbasis* for $\mathbf{X}$ is a subset $\sigma \subseteq \tau$ such that the closure of σ under finite intersections forms a basis for $\mathbf{X}$.

```
isBasisFor :: (Ord a) => Set (Set a) -> TopoSpace a -> Bool
isBasisFor sets (TopoSpace _ opens) = closeUnderUnion sets == opens

isSubbasisFor :: (Ord a) => Set (Set a) -> TopoSpace a -> Bool
isSubbasisFor sets topoSpace = closeUnderIntersection sets `isBasisFor` topoSpace
```

Given some topological space (X, τ) and a subset $S \subseteq X$, the *interior* of S, denoted by $\mathrm{int}(S)$, is the union of all open subsets of S, i.e.

$$\bigcup \{U \in \tau \mid U \subseteq S\}$$

The *closure* of S, denoted by $\overline{S}$, is the intersection of all closed supersets of S, i.e.

$$\bigcap \{C \in \overline{\tau} \mid S \subseteq C\}$$

Here we implement the union and intersection functions utilised above as well as the interior and closure operations.

```
arbUnion :: (Ord a) => Set (Set a) -> Set a
arbUnion = S.foldr union S.empty

arbIntersection :: (Eq a, Ord a) => Set (Set a) -> Set a
arbIntersection sets
    | sets == S.empty = error "Cannot take the intersection of the empty set."
    | length sets == 1 = firstSet
    | otherwise = firstSet `intersection` arbIntersection restOfSets
  where
    firstSet = elemAt 0 sets
    restOfSets = S.drop 1 sets

interior :: (Ord a) => Set a -> TopoSpace a -> Set a
interior set topoSpace = arbUnion opensBelowSet
  where
    TopoSpace _ opens = topoSpace
    opensBelowSet = S.filter (`isSubsetOf` set) opens
```

```haskell
closure :: (Ord a) => Set a -> TopoSpace a -> Set a
closure set topoSpace = arbIntersection closedsAboveSet
  where
    closedsAboveSet = S.filter (set `isSubsetOf`) (closeds topoSpace)
```

Examples of using the above:

```
ghci> interior (S.fromList [1]) topoSpace
fromList [1]

ghci> closure (S.fromList [1]) topoSpace
fromList [1,2]
```

3 Syntax

```haskell
module Syntax where

import Test.QuickCheck
```

```haskell
data Form
    = Top
    | Bot
    | P Int
    | Form `Dis` Form
    | Form `Con` Form
    | Form `Imp` Form
    | Neg Form
    | Dia Form
    | Box Form
    deriving (Eq, Show, Ord)
```

```haskell
instance Arbitrary Form where
  arbitrary = sized randomForm
   where
    randomForm :: Int -> Gen Form
    randomForm 0 = P <$> elements [1 .. 5] -- Fixed vocabulary
    randomForm n =
      oneof
        [
        P <$> elements [1 .. 5]
        , Dis
            <$> randomForm (n `div` 2)
            <*> randomForm (n `div` 2)
        , Con
            <$> randomForm (n `div` 2)
            <*> randomForm (n `div` 2)
        , Imp
            <$> randomForm (n `div` 2)
            <*> randomForm (n `div` 2)
        , Neg <$> randomForm (n `div` 2)
        , Dia <$> randomForm (n `div` 2)
        , Box <$> randomForm (n `div` 2)
        ]
```

4 Semantics

In this module we define the semantics for the formulas defined in `Syntax.lhs` on both TopoModels and **S4** Kripke models.

```
module Semantics where

import qualified Data.Set as S

import KripkeModels (PointedS4KripkeModel (PS4KM), S4KripkeFrame (S4KF), S4KripkeModel (
    S4KM))
import Syntax (Form (..))
import TopoModels (PointedTopoModel (..), TopoModel (..))
import Topology (TopoSpace (TopoSpace), openNbds)
```

```
class PointSemantics pointedModel where
    (|=) :: pointedModel -> Form -> Bool

class ModelSemantics model where
    (||=) :: model -> Form -> Bool
```

```
instance (Eq a) => PointSemantics (PointedTopoModel a) where
    (|=) _ Top = True
    (|=) _ Bot = False
    (|=) pointedModel (P n) = x `elem` worldsWherePnTrue
      where
        PointedTopoModel topoModel x = pointedModel
        TopoModel _ valuation = topoModel
        worldsWherePnTrue = snd . S.elemAt 0 $ S.filter (\(p, _) -> p == P n) valuation
    (|=) pointedModel (phi `Dis` psi) = (pointedModel |= phi) || (pointedModel |= psi)
    (|=) pointedModel (phi `Con` psi) = (pointedModel |= phi) && (pointedModel |= psi)
    (|=) pointedModel (phi `Imp` psi) = pointedModel |= (Neg phi `Dis` psi)
    (|=) pointedModel (Neg phi) = not $ pointedModel |= phi
    (|=) pointedModel (Dia phi) = pointedModel |= Neg (Box (Neg phi)) -- TODO - Implement
        this as a primitive
    (|=) pointedModel (Box phi) = not (null openNbdsSatisfyingFormula)
      where
        PointedTopoModel topoModel point = pointedModel
        TopoModel topoSpace _ = topoModel
        wholeSetSatisfiesForm set psi = all (\x -> PointedTopoModel topoModel x |= psi) set
        openNbdsOfPoint = openNbds point topoSpace
        openNbdsSatisfyingFormula = S.filter (`wholeSetSatisfiesForm` phi) openNbdsOfPoint

instance (Eq a) => ModelSemantics (TopoModel a) where
    topoModel ||= phi = wholeSetSatisfiesForm space phi
      where
        (TopoModel topoSpace _) = topoModel
        TopoSpace space _ = topoSpace
        wholeSetSatisfiesForm set psi = all (\x -> PointedTopoModel topoModel x |= psi) set
```

```
instance (Eq a, Ord a) => PointSemantics (PointedS4KripkeModel a) where
    (|=) _ Top = True
    (|=) _ Bot = False
    (|=) pointedModel (P n) = x `elem` worldsWherePnTrue
      where
        PS4KM kripkeModel x = pointedModel
        S4KM _ valuation = kripkeModel
        worldsWherePnTrue = snd . S.elemAt 0 $ S.filter (\(p, _) -> p == P n) valuation
    (|=) pointedModel (phi `Dis` psi) = (pointedModel |= phi) || (pointedModel |= psi)
    (|=) pointedModel (phi `Con` psi) = (pointedModel |= phi) && (pointedModel |= psi)
    (|=) pointedModel (phi `Imp` psi) = pointedModel |= (Neg phi `Dis` psi)
    (|=) pointedModel (Neg phi) = not $ pointedModel |= phi
    (|=) pointedModel (Dia phi) = pointedModel |= Neg (Box (Neg phi)) -- TODO - Implement
        this as a primitive
    (|=) pointedModel (Box phi) = all (\w' -> PS4KM kripkeModel w' |= phi) imageOfWorld
```

```haskell
      where
        (PS4KM kripkeModel world) = pointedModel
        S4KM kripkeFrame _ = kripkeModel
        S4KF _ relation = kripkeFrame
        imageOfWorld = S.map snd $ S.filter (\(x, _) -> x == world) relation

instance (Eq a, Ord a) => ModelSemantics (S4KripkeModel a) where
    kripkeModel ||= phi = wholeSetSatisfiesForm carrier phi
      where
        (S4KM frame _) = kripkeModel
        (S4KF carrier _) = frame
        wholeSetSatisfiesForm set psi = all (\x -> PS4KM kripkeModel x |= psi) set
```

5 Executables

Finish me.

```haskell
module Main where

main :: IO ()
main = undefined
```

6 Tests

```haskell
module Main where

import Topology
import TopoModels
import Syntax
import Semantics
import TestHelpers

import Test.Hspec
    ( hspec, describe, it, shouldBe, shouldThrow, anyException )
import Test.Hspec.QuickCheck ( prop)
import Test.QuickCheck
import Control.Exception (evaluate)

import Data.Set (Set, isSubsetOf)
import qualified Data.Set as S
```

```haskell
main :: IO ()
main = hspec $ do
  describe "TopoSpace generation" $ do
    prop "Arbitrary TopoSpace satisfies the open set definition of a topo space" $ do
      \ts -> isTopoSpace (ts :: TopoSpace Int)
    prop "The subset in arbitrary SubsetTopoSpace is indeed a subset of the space" $ do
      \(STS setA (TopoSpace space _)) -> (setA :: Set Int) `isSubsetOf` space
    prop "The two subsets in arbitrary SSubsetTopoSpace are indeed subsets of the space" $
        do
      \(SSTS setA setB (TopoSpace space _)) -> (setA :: Set Int) `isSubsetOf` space && (
          setB :: Set Int) `isSubsetOf` space
  describe "Kuratowski Axioms for the closure operator" $ do
    prop "Preserves the empty set" $ do
        \x -> closure S.empty (x :: TopoSpace Int) `shouldBe` S.empty
    prop "Is extensive for all A \\subseteq X" $ do
        \ (STS setA ts) -> (setA :: Set Int) `isSubsetOf` closure setA ts
    prop "Is idempotent for all A \\subseteq X" $ do
        \(STS setA ts) -> closure (setA :: Set Int) ts `shouldBe` closure (closure setA ts)
             ts
    prop "Distributes over binary unions" $ do
```

```
      \(SSTS setA setB ts) ->
        closure ((setA :: Set Int) `S.union` setB) ts `shouldBe`
        closure setA ts `S.union` closure setB ts
describe "Kuratowski Axioms for the interior operator" $ do
  prop "Preserves the whole space" $ do
      \(TopoSpace space topo) -> interior (space :: Set Int) (TopoSpace space topo) `
          shouldBe` space
  prop "Is intensive for all A \\subseteq X" $ do
      \ (STS setA ts) -> interior (setA :: Set Int) ts `isSubsetOf` setA
  prop "Is idempotent for all A \\subseteq X" $ do
       \(STS setA ts) -> interior (setA :: Set Int) ts `shouldBe` interior (interior setA
            ts) ts
  prop "Distributes over binary intersections" $ do
      \(SSTS setA setB ts) ->
        interior ((setA :: Set Int) `S.intersection` setB) ts `shouldBe`
        interior setA ts `S.intersection` interior setB ts
describe "Examples from the Topology module" $ do
  it "closeUnderUnion $ Set.fromList [s0, s1, s2]" $ do
    let result = S.fromList [S.fromList [1], S.fromList [1,2], S.fromList [1,2,3,4], S.
        fromList [1,3,4],S.fromList [2],S.fromList [2,3,4],S.fromList [3,4]]
    closeUnderUnion (S.fromList [s0, s1, s2]) `shouldBe` result
  it "closeUnderIntersection $ Set.fromList [s0, s1, s2]" $ do
    let result = S.fromList [S.fromList [], S.fromList [1], S.fromList [2], S.fromList
        [3,4]]
    closeUnderIntersection (S.fromList [s0, s1, s2]) `shouldBe` result
  it "closeUnderUnion $ Set.fromList [s3, s4, s5]" $ do
    let result = S.fromList [S.fromList [1,2,3], S.fromList [1,2,3,4], S.fromList [2,3],
        S.fromList [2,3,4], S.fromList [3,4]]
    closeUnderUnion (S.fromList [s3, s4, s5]) `shouldBe` result
  it "closeUnderIntersection $ Set.fromList [s3, s4, s5]" $ do
    let result = S.fromList [S.fromList [1,2,3], S.fromList [2,3], S.fromList [3], S.
        fromList [3,4]]
    closeUnderIntersection (S.fromList [s3, s4, s5]) `shouldBe` result
  it "(closeUnderUnion . closeUnderIntersection) $ Set.fromList [s5, s6, s7]" $ do
    let result = S.fromList [S.fromList [], S.fromList [1], S.fromList [1,2], S.fromList
        [1,2,3], S.fromList [1,2,3,4], S.fromList [1,3], S.fromList [1,3,4], S.fromList
        [3], S.fromList [3,4]]
    (closeUnderUnion . closeUnderIntersection) (S.fromList [s5, s6, s7]) `shouldBe`
        result
  it "isTopoSpace (TopoSpace (arbUnion Set.fromList [s5, s6, s7]) topology)" $ do
    isTopoSpace (TopoSpace (arbUnion $ S.fromList [s5, s6, s7]) topology)
  it "isTopoSpace badTS" $ do
    not . isTopoSpace $ badTS
  it "isTopoSpace goodTS" $ do
    isTopoSpace goodTS
  it "isTopoSpace (fixTopoSpace goodTS)" $ do
    isTopoSpace (fixTopoSpace goodTS)
  it "closeds topoSpace" $ do
    let result = S.fromList [S.fromList [], S.fromList [1,2], S.fromList [1,2,3,4], S.
        fromList [1,2,4], S.fromList [2], S.fromList [2,3,4], S.fromList [2,4], S.
        fromList [3,4], S.fromList [4]]
    closeds topoSpace `shouldBe` result
  it "openNbds 2 topoSpace" $ do
    let result = S.fromList [S.fromList [1,2], S.fromList [1,2,3], S.fromList [1,2,3,4]]
    openNbds 2 topoSpace `shouldBe` result
  it "(S.fromList [1]) `isOpenIn` topoSpace" $ do
    S.fromList [1] `isOpenIn` topoSpace
  it "(S.fromList [1]) `isClosedIn` topoSpace" $ do
    not (S.fromList [1] `isClosedIn` topoSpace)
  it "(S.fromList []) `isClopenIn` topoSpace" $ do
    S.fromList [] `isClopenIn` topoSpace
  it "interior (Set.fromList [1]) topoSpace" $ do
    let result = S.fromList [1]
    interior (S.fromList [1]) topoSpace `shouldBe` result
  it "closure (Set.fromList [1]) topoSpace" $ do
    let result = S.fromList [1,2]
    closure (S.fromList [1]) topoSpace `shouldBe` result
  it "fixTopoSpace (TopoSpace (S.fromList [1,2,3]) topology)" $ do
     evaluate (fixTopoSpace (TopoSpace (S.fromList [1,2,3]) topology)) `shouldThrow`
         anyException
describe "TopoModel semantics" $ do
  prop "Validates the K axiom" $ do
    \ts -> (ts :: TopoModel Int) ||= kAxiom
    prop "Validates tautology: p or not p" $ do
      \ts -> (ts :: TopoModel Int) ||= (P 1 `Dis` Neg (P 1))
    prop "Validates tautology: p implies p" $ do
      \ts -> (ts :: TopoModel Int) ||= (P 1 `Imp` P 1)
    prop "Validates tautology: p implies (q implies (p and q))" $ do
      \ts -> (ts :: TopoModel Int) ||= (P 1 `Imp` (P 2 `Imp` (P 1 `Con` P 2)))
    prop "Validates modal tautology: Dia p or not Dia p"$ do
      \ts -> (ts :: TopoModel Int) ||= (Dia (P 1)`Dis` Neg (Dia (P 1)))
    prop "Validates modal tautology: Box p implies Dia p"$ do
      \ts -> (ts :: TopoModel Int) ||= (Box (P 1) `Imp` Dia (P 1))
    prop "Cannot satisfy contradiction p and not p" $ do
      \ts -> not ((ts :: PointedTopoModel Int) |= (P 1 `Con` Neg (P 1)))
    prop "Cannot satisfy contradiction ((P or Q) implies R) and not ((P or Q) implies R)" $
        do
      \ts -> not ((ts :: PointedTopoModel Int) |= (((P 1 `Dis` P 2) `Imp` P 3) `Con` Neg ((
         P 1 `Dis` P 2) `Imp` P 3)))
    prop "Cannot satisfy modal contradiction: Dia p or not Dia p" $ do
      \ts -> not ((ts :: PointedTopoModel Int) |= (Dia (P 1) `Con` Neg (Dia (P 1))))
    prop "Cannot satisfy modal contradiction: Box p and Dia not p" $ do
      \ts -> not ((ts :: PointedTopoModel Int) |= (Box (P 1) `Con` Dia (Neg (P 1))))
```

To run the tests, use `stack test`.

To also find out which part of your program is actually used for these tests, run `stack clean && stack test` Then look for "The coverage report for ... is available at ... .html" and open this file in your browser. See also: `https://wiki.haskell.org/Haskell_program_coverage`.

7 Conclusion

Finish me.

References