Topomodels

David Álvarez Lombardi   Paulius Skaisgiris

Sunday 4th June, 2023

Abstract

Finish me.

Contents

1 Introduction

Finish me.

2 Topological Preliminaries

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

Note: In our Haskell implementation we will use lists instead of sets as they seem easier to work with.

A *topological space* is a pair (X, τ) where X is a nonempty set and $\tau \subseteq \mathcal{P}(X)$ is a family of subsets of X such that 1. $\emptyset \in \tau$ 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.

```
module Topology where

import Data.List

-- TODO: workaround nub somehow?
-- Notion of set equality on lists
eq :: Ord a => [a] -> [a] -> Bool
eq x y = sort (nub x) == sort (nub y)

unionize :: (Eq a, Ord a) => [[a]] -> [[a]]
unionize [] = []
unionize xs = sort . nub $ [sort (x `union` y) | x <- xs, y <- xs, x /= y]

intersectionize :: (Eq a, Ord a) => [[a]] -> [[a]]
intersectionize [] = []
intersectionize xs = sort . nub $ [sort (x `intersect` y) | x <- xs, y <- xs, x /= y]

-- The closure definitions defined below are finite, but it is sufficient for our purposes
-- since we will only work with finite models.
closeUnderUnion :: (Eq a, Ord a) => [[a]] -> [[a]]
closeUnderUnion [] = []
closeUnderUnion xs = do
 let oneUp = nub (xs ++ unionize xs)
 if xs == oneUp  then xs
 else closeUnderUnion oneUp

closeUnderIntersection :: (Eq a, Ord a) => [[a]] -> [[a]]
closeUnderIntersection [] = []
closeUnderIntersection xs = do
 let oneUp = nub (xs ++ intersectionize xs)
 if xs == oneUp  then xs
 else closeUnderIntersection oneUp
```

Some examples of applying the closure functions:

```
ghci> closeUnderUnion [[1], [2], [3, 4]]
ghci> [[1],[2],[3,4],[1,2],[1,3,4],[2,3,4],[1,2,3,4]]

ghci> closeUnderIntersection [[1,2,3], [2,3], [3,4]]
ghci> [[1,2,3],[2,3],[3,4],[3]]
ghci> let t = closeUnderIntersection . closeUnderUnion \$ [[1, 2], [1,3], [3, 4]]
ghci> t
ghci> [[1,2],[1,3],[3,4],[1,2,3],[1,2,3,4],[1,3,4],[],[1],[3]]
```

Now, we can define a Topological space in Haskell.

```
data  TopoSpace a = TopoSpace {
  space :: [a]
, top :: [[a]]
} deriving (Show)
```

The elements of τ are called *open sets* or *opens*. A set $C \subseteq X$ is called a *closed set* if it is the complement of an open set, i.e., it is of the form $X \setminus U$ for some $U \in \tau$.

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

A set $A \subseteq X$ is called *clopen* if it is both closed and open.

```
opens :: TopoSpace a -> [[a]]
opens = top

closeds :: (Eq a) => TopoSpace a -> [[a]]
closeds ts = [space ts \\ open | open <- top ts]

isOpen :: (Eq a) => [a] -> TopoSpace a -> Bool
isOpen x ts = x `elem` opens ts

isClosed :: (Eq a) => [a] -> TopoSpace a -> Bool
isClosed x ts = x `elem` closeds ts

isClopen :: (Eq a) => [a] -> TopoSpace a -> Bool
isClopen x ts = isOpen x ts && isClosed x ts
```

Examples of using the above:

```
ghci> let ts = TopoSpace {space = [1,2,3,4], top = t}
ghci> opens ts
ghci> [[1,2],[1,3],[3,4],[1,2,3],[1,2,3,4],[1,3,4],[],[1],[3]]
ghci> closeds ts
ghci> [[3,4],[2,4],[1,2],[4],[],[2],[1,2,3,4],[2,3,4],[1,2,4]]
ghci> isOpen [1] ts
ghci> True
ghci> isClosed [1] ts
ghci> False
ghci> isClopen [] ts
ghci> True
```

The *interior* of a subset S of a topological space X is the union of all open subsets of S.

The *closure* of a subset S of a topological space X is the intersection of all closed subsets containing S.

```
powerset :: [a] -> [[a]]
powerset [] = [[]]
powerset (x:xs) = [x:ps | ps <- powerset xs] ++ powerset xs

-- Equivalent property taken from https://en.wikipedia.org/wiki/Subset#Properties
isSubsetEq :: (Eq a) => [a] -> [a] -> Bool
isSubsetEq xs ys = (xs `intersect` ys) == xs

interior :: (Eq a, Ord a) => [a] -> TopoSpace a -> [a]
interior xs ts = sort . nub . concat $ [ u | u <- top ts, isSubsetEq u xs]

arbIntersect :: (Eq a) => [[a]] -> [a]
arbIntersect xs = foldr intersect (concat xs) xs
```

```
closure :: (Eq a, Ord a) => [a] -> TopoSpace a -> [a]
closure xs ts = sort . nub . arbIntersect $ [ u | u <- closeds ts, isSubsetEq xs u]
```

Examples of using the above:

```
ghci> powerset [1,2,3]
ghci> [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]

ghci> isSubsetEq [] [1,2,3]
ghci> True

ghci> isSubsetEq [1,2] [1,2,3]
ghci> True

ghci> isSubsetEq [1,4] [1,2,3]
ghci> False

ghci> interior [1,2] ts
ghci> [1,2]

ghci> closure [1,2] ts
ghci> [1,2]
```

3 Syntax

```
module Syntax where

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

4 Semantics

```
module Semantics where

import Syntax
import Topology
import TopoModels


satisfies :: Eq a => PointedTopoModel a -> Form -> Bool
satisfies _ Top = True
satisfies _ Bot = False
satisfies pointedModel (P n) = x `elem` valuation (P n)
    where
        PointedTopoModel topoModel x = pointedModel
        TopoModel _ valuation = topoModel
satisfies pointedModel (phi `Dis` psi) = pointedModel `satisfies` Neg (Neg phi `Con` Neg
    psi)
satisfies pointedModel (phi `Con` psi) = (pointedModel `satisfies` phi) && (pointedModel `
    satisfies` psi)
```

```
satisfies pointedModel (phi `Imp` psi) = pointedModel `satisfies` (Neg phi `Dis` psi)
satisfies pointedModel (Neg phi) = not $ pointedModel `satisfies` phi
satisfies pointedModel (Dia phi) = pointedModel `satisfies` Neg (Box (Neg phi))
satisfies pointedModel (Box phi) = not (null openNeighbourhoodsContainedInTruthSet)
    where
        PointedTopoModel topoModel point = pointedModel
        TopoModel topoSpace _ = topoModel
        TopoSpace space topology = topoSpace
        truthSet = [x | x <- space, PointedTopoModel topoModel x `satisfies` phi]
        openNeighbourhoodsContainedInTruthSet = [openSet | openSet<-topology, point `elem`
            openSet, openSet `isSubsetEq` truthSet]

(|=) :: Eq a => PointedTopoModel a -> Form -> Bool
pointedModel |= phi = pointedModel `satisfies` phi

(||=) :: Eq a => TopoModel a -> Form -> Bool
topoModel ||= phi = all (\x -> PointedTopoModel topoModel x |= phi) space
    where
        (TopoModel topoSpace _) = topoModel
        TopoSpace space _ = topoSpace
```

5 Executables

Finish me.

```
module Main where

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

6 Tests

Finish me.

7 Conclusion

Finish me.

References