# 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 $(X, \tau)$ 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. $\tau$ is closed under finite intersection: if $U, V \in \tau$ then $U \cap V \in \tau$ 3. $\tau$ 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.Set (Set, cartesianProduct, union, intersection, (\\), elemAt, isSubsetOf)
import qualified Data.Set as Set

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

intersectionize :: Ord a => Set (Set a) -> Set (Set a)
intersectionize sets = Set.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 closeUnderUnion 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 (Set a) (Set (Set a))
    deriving (Eq, Show)
```

The elements of $\tau$ 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 $(X, \tau)$.

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

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

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

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

isClopenIn :: (Eq a , Ord a) => Set a -> TopoSpace a -> Bool
isClopenIn x ts =  x `isOpenIn` ts && x `isClosedIn` 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 $S$.

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

arbIntersection :: (Eq a, Ord a) => Set (Set a) -> Set a
arbIntersection sets | sets == Set.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 = undefined

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

closure :: Ord a => Set a -> TopoSpace a -> Set a
closure set topoSpace = arbIntersection closedsAboveSet
    where
```

```haskell
        closedsAboveSet = Set.filter (\c -> set `isSubsetOf` c) (closeds topoSpace)
```

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

```haskell
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

```haskell
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