Topomodels

David Álvarez Lombardi   Paulius Skaisgiris

Sunday 4th June, 2023

Abstract

Finish me.

Contents

1 Introduction **2**

2 Topological Preliminaries **2**

3 Syntax **4**

4 Semantics **4**

5 Executables **5**

6 Tests **5**

7 Conclusion **6**

Bibliography **6**

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

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
```

Some examples of applying the closure functions:

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

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

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

ghci> topology = (closeUnderUnion . closeUnderIntersection) \$ Set.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) (Set (Set a))
    deriving (Eq, 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.

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

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

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 set topoSpace = set `elem` closeds topoSpace

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 (Set.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> (Set.fromList [1]) `isOpenIn` topoSpace
True
ghci> (Set.fromList [1]) `isClosedIn` topoSpace
False
ghci> (Set.fromList []) `isClopenIn` topoSpace
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 = 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

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 (Set.fromList [1]) topoSpace
fromList [1]

ghci> closure (Set.fromList [1]) topoSpace
fromList [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 Data.Set qualified as S
```

```
import Syntax
import TopoModels
import Topology

satisfies :: (Eq a) => PointedTopoModel a -> Form -> Bool
satisfies _ Top = True
satisfies _ Bot = False
satisfies 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
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 openNbdsSatisfyingFormula)
  where
    PointedTopoModel topoModel point = pointedModel
    TopoModel topoSpace _ = topoModel
    wholeSetSatisfiesForm set psi = all (\x -> PointedTopoModel topoModel x `satisfies` psi
        ) set
    openNbdsOfPoint = openNbds point topoSpace
    openNbdsSatisfyingFormula = S.filter (`wholeSetSatisfiesForm` phi) openNbdsOfPoint

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

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

5 Executables

Finish me.

```
module Main where

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

6 Tests

Finish me.

7 Conclusion

Finish me.

References