Topomodels in Haskell

David Álvarez Lombardi   Paulius Skaisgiris

Sunday, 4 June 2023

Abstract

In this project, we provide a library for working with general topological spaces as well as topomodels for modal logic. We also implement a well-known construction for converting topomodels to **S4** Kripke models and back.

Contents

1 Introduction

Finish me.

2 Modal logic

In this section we define the syntax of a basic propositional modal logic.

```
module Syntax where

import Test.QuickCheck
```

2.1 Syntax

Our language will be the formulas of the following shape.

$$\varphi := \top \mid \bot \mid p_n \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \to \varphi \mid \neg\varphi \mid \Diamond\varphi \mid \Box\varphi$$

```
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)
```

2.2 Arbitrary modal formula generation

When testing, we can generate arbitrary `Form`'s via the following instance of `Arbitrary`.

```
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)
                ]
```

3 Set theory

In this section we define some set-theoretic helpers that will come in handy in the following sections.

```
module SetTheory where

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

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

3.1 Unions and intersections

A set of sets S is called *closed under unions* if $T, V \in S$ implies that $T \cup V \in S$. Similarly, S is called *closed under intersections* if $T, V \in S$ implies that $T \cap V \in S$.

The following functions close a passed set under unions and intersections.

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

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

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

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

We also include, for convenience, the following functions which correspond to $\bigcup$ and $\bigcap$ respectively.

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

```haskell
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
```

3.2 Relations

Below are a couple of simple helper functions for working with binary relations.

```haskell
type Relation a = Set (a, a)

field :: (Ord a) => Relation a -> Set a
field relation = domain `union` range
  where
    domain = S.map fst relation
    range = S.map snd relation

imageIn :: (Ord a) => a -> Relation a -> Set a
imageIn element relation = S.map snd $ S.filter (\(x, _) -> x == element) relation
```

Given a set X and a relation $R \subseteq X \times X$, we say that R is *transitive* if it satisfies, for all $x, y, z \in X$,

$$xRy \text{ and } yRz \text{ implies } xRz$$

Below is a function for making a passed relation transitive.

```haskell
onceMakeTransitive :: (Ord a) => Relation a -> Relation a
onceMakeTransitive relation = do
    let relField = field relation
    let fieldCubed = cartesianProduct (cartesianProduct relField relField) relField
    let relTriples = S.filter (\((x, y), z) -> (x, y) `member` relation && (y, z) `member`
        relation) fieldCubed
    let additions = S.map (\((x, _), z) -> (x, z)) relTriples
    relation `union` additions

makeTransitive :: (Ord a) => Relation a -> Relation a
makeTransitive relation = do
    let oneUp = onceMakeTransitive relation
    if relation == oneUp
        then relation
        else makeTransitive oneUp
```

3.3 Arbitrary set generation

Here we define functions that are useful in the (constrained) generation of arbitrary sets. These mirror their commonly-used `List`-counterparts, but must be adapted as we work with `Data.Set`. Inspiration for this implementation was taken from here.

```haskell
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

isOfSizeBetween :: Set a -> Int -> Int -> Bool
isOfSizeBetween set lower upper = lower <= S.size set && S.size set <= upper
```

4 Models

In this section we define some concepts that will be used in the subsequent sections on *Kripke models* and *topomodels*.

```haskell
module Models where

import Data.Set (Set, union)
import qualified Data.Set as S
import Test.QuickCheck (Arbitrary, Gen)

import SetTheory (subsetOf)
import Syntax (Form (P))
```

4.1 Arbitrary valuation generation

Given a set X, a *valuation* is a function $V : \mathbf{Prop} \to \wp(X)$ where **Prop** is the set of formulas of the shape p_n.

Note: The type of `Valuation` below suggests our valuations should be defined on all `Form` (instead of just on **Prop**), but this is merely an implementation detail and when generating arbitrary `Valuation`'s we only define them on **Prop**.

```haskell
type Valuation a = Set (Form, Set a)

-- Recursively make a valuation function. Go through each propositional variable
-- and assign a subset of points at which it is true
randomVal :: (Arbitrary a, Ord a) => Set a -> [Int] -> Gen (Valuation a)
randomVal _ [] = return S.empty
randomVal points (prop : props)
    | null points = return S.empty
    | otherwise = do
        x <- randomVal points props
        randSubset <- subsetOf points
        return $ S.singleton (P prop, randSubset) `union` x
```

5 Kripke models

In this section we define relational models of modal logic.

```
{-# LANGUAGE ScopedTypeVariables #-}

module KripkeModels where

import Data.Set (Set, cartesianProduct, union)
import qualified Data.Set as S
import Test.QuickCheck (Arbitrary (arbitrary), suchThat)

import Models (Valuation, randomVal)
import SetTheory (Relation, isOfSizeBetween, makeTransitive, setElements, subsetOf)
```

5.1 Kripke models

An **S4** *Kripke frame* is a tuple (X, R) where X is a set and $R \subseteq X \times X$ and the following are true for all $x, y, z \in X$.

- xRx (Reflexivity)
- xRy and yRz implies xRz (Transitivity)

An **S4** *Kripke model* is a triple (X, R, V) where (X, R) is an **S4** Kripke frame and V is a valuation on X.

A *pointed* **S4** *Kripke model* is a 4-tuple (X, R, V, x) where (X, R, V) is an **S4** Kripke model and $x \in X$.

```
data S4KripkeFrame a = S4KF (Set a) (Relation a)
    deriving (Eq, Show)

data S4KripkeModel a = S4KM (S4KripkeFrame a) (Valuation a)
    deriving (Eq, Show)

data PointedS4KripkeModel a = PS4KM (S4KripkeModel a) a
    deriving (Eq, Show)
```

5.2 Arbitrary Kripke model generation

Below we define a method for generating arbitrary Kripke models. This presented something of an interesting challenge as we cannot simply take *any* relation on *any* carrier set; we must ensure that the generated frame is, indeed, **S4** (i.e. reflexive and transitive).

To accomplish this, we generate an arbitrary carrier set and an arbitrary subset of its cartesian product. We then add to this random relation all of the reflexive pairs and close it under transitive triples.

This closure process grows the relation significantly so, in order to avoid ending up with a complete graph, we choose a starting relation that is quite small. Given a carrier set of cardinality n, instead of allowing any subset of the cross product (which could be as large as n^2), we capped the random relation at cardinality $2n$, which ensures that we get interesting frames.

```haskell
instance (Arbitrary a, Ord a) => Arbitrary (S4KripkeFrame a) where
    arbitrary = do
        (carrier :: Set a) <- arbitrary `suchThat` (\set -> isOfSizeBetween set 1 10)
        let carrierSquared = cartesianProduct carrier carrier
        let relationIsNotTooBig rel = isOfSizeBetween rel 1 (S.size carrier * 2)
        {-
            If no cap is put on this then the resulting frame (after being made
            reflexive and transitive) will almost always be a complete graph,
            which is uninteresting.
        -}
        (randomRelation :: Relation a) <- subsetOf carrierSquared `suchThat`
            relationIsNotTooBig
        let diagonal = S.filter (uncurry (==)) carrierSquared
        let reflexiveRelation = randomRelation `union` diagonal
        let s4Relation = makeTransitive reflexiveRelation
        return (S4KF carrier s4Relation)

instance (Arbitrary a, Ord a) => Arbitrary (S4KripkeModel a) where
    arbitrary = do
        (randomFrame :: S4KripkeFrame a) <- arbitrary
        let (S4KF carrier _) = randomFrame
        (randomValuation :: Valuation a) <- randomVal carrier [1 .. 10]
        return (S4KM randomFrame randomValuation)

instance (Arbitrary a, Ord a) => Arbitrary (PointedS4KripkeModel a) where
    arbitrary = do
        (randomModel :: S4KripkeModel a) <- arbitrary
        let (S4KM frame _) = randomModel
        let (S4KF carrier _) = frame
        point <- setElements carrier
        return (PS4KM randomModel point)
```

6 Topological preliminaries

In this section we define basic topological concepts that will form the foundation for our subsequent definition of topomodels and toposemantics for modal logic.

```haskell
{-# LANGUAGE ScopedTypeVariables #-}

module Topology where

import Data.Set (Set, isSubsetOf, singleton, union, (\\))
import qualified Data.Set as S
import Test.QuickCheck (Arbitrary (arbitrary), suchThat)

import SetTheory (arbIntersection, arbUnion, closeUnderIntersection, closeUnderUnion,
    isOfSizeBetween, setElements, setOf1)
```

6.1 Topological spaces

A *topological space* (or *topospace*) is a tuple (X, τ) where X is a non-empty set and $\tau \subseteq \wp(X)$ is a family of subsets of X such that

1. $\varnothing, X \in \tau$
2. $S \subseteq \tau$ and $|S| < \omega$ implies $\bigcap S \in \tau$
3. $S \subseteq \tau$ implies $\bigcup S \in \tau$

```
type Topology a = Set (Set a)

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

The elements of τ are referred to as *open sets*, so we say a subset $S \subseteq X$ is *open* in τ if $S \in \tau$. Given a point $x \in X$, we call the set of all open sets containing x the *open neighbourhoods of* x.

Additionally, we say that S is *closed* (in τ) if $X - A \in \tau$ (i.e. S is the complement of an open set). The set of closed sets of (X, τ) is denoted by $\overline{\tau}$.

Finally, we say that S is *clopen* if it is both open and closed.

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

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

isClosedIn :: (Eq a, Ord a) => Set a -> TopoSpace a -> Bool
isClosedIn set (TopoSpace space topology) = space \\ set `elem` topology

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

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

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

$$\text{int}(S) := \bigcup \{U \in \tau \mid U \subseteq S\}$$

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

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

```
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
```

```haskell
closure set topoSpace = arbIntersection closedsAboveSet
  where
    closedsAboveSet = S.filter (set `isSubsetOf`) (closeds topoSpace)
```

6.2 Bases and Subbases

Given a 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 $\mathbf{X}$.

```haskell
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
```

6.3 Arbitrary topological spaces

First we include a couple of helper functions.

The first of them checks if the passed `TopoSpace` is, indeed, a topological space (i.e. respects all of the axioms).

The second actually *fixes* a passed `TopoSpace` in the case that it is not *truly* a topological space (i.e. fails to satisfy one of the axioms). This is necessary for the generation of arbitrary topospaces later on.

```haskell
isTopoSpace :: (Ord a) => TopoSpace a -> Bool
isTopoSpace (TopoSpace space topology)
    -- Passed space is empty
    | space == S.empty = False
    -- Passed topology is not a subset of the power set of passed space
    | not (arbUnion topology `isSubsetOf` space) = False
    -- Passed topology is missing the empty set or the full space
    | S.empty `notElem` topology || space `notElem` topology = False
    -- Passed topology should be closed under intersections and unions
    | otherwise = topology == (closeUnderUnion . closeUnderIntersection) topology

fixTopoSpace :: (Ord a) => TopoSpace a -> TopoSpace a
fixTopoSpace (TopoSpace space topology)
    -- Throw an error since we don't know how the topology should look like
    | not (S.unions topology `isSubsetOf` space) = error "Points in topology are not all
      members of the space"
    | S.empty `notElem` topology = fixTopoSpace (TopoSpace space (topology `union`
      singleton S.empty))
    | space `notElem` topology = fixTopoSpace (TopoSpace space (topology `union` singleton
      space))
    | otherwise = TopoSpace space closedTopology
  where
    closedTopology = closeUnderUnion . closeUnderIntersection $ topology
```

Now we can define a method for generating arbitrary topospaces.

```
instance (Arbitrary a, Ord a) => Arbitrary (TopoSpace a) where
    arbitrary = do
        (x :: Set a) <- arbitrary `suchThat` (\set -> isOfSizeBetween set 1 10)
        -- 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` (\set -> isOfSizeBetween set 0 3)
             in setOf1 basis `suchThat` (\set -> isOfSizeBetween set 0 3)
        let someTopoSpace = TopoSpace x subbasis
        return (fixTopoSpace someTopoSpace)
```

7 Topomodels

In this section we define topological models of modal logic.

```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE ScopedTypeVariables #-}

module TopoModels where

import Test.QuickCheck (Arbitrary (arbitrary))

import Models (Valuation, randomVal)
import SetTheory (setElements)
import Topology (TopoSpace (TopoSpace))
```

7.1 Topomodels

A *topomodel* is a triple (X, τ, V) where (X, τ) is a topospace and V is a valuation on X.

A *pointed topomodel* is a 4-tuple (X, τ, V, x) where (X, τ, V) is an topomodel and $x \in X$.

```
data TopoModel a = TopoModel (TopoSpace a) (Valuation a)
    deriving (Eq, Show)

data PointedTopoModel a = PointedTopoModel (TopoModel a) a
    deriving (Eq, Show)
```

7.2 Arbitrary topomodel generation

Below we define a method for generating arbitrary topomodels.

```
instance (Arbitrary a, Ord a) => Arbitrary (TopoModel a) where
  arbitrary = do
    (TopoSpace space topo) <- arbitrary
```

```
    -- Random Valuation depending on the points of the space
    -- Fix the number of propositional variables
    val <- randomVal space [1..10]
    return (TopoModel (TopoSpace space topo) val)

instance (Arbitrary a, Ord a) => Arbitrary (PointedTopoModel a) where
  arbitrary = do
   (TopoModel (TopoSpace space topo) val) <- arbitrary
   (x :: a) <- setElements space
   return (PointedTopoModel (TopoModel (TopoSpace space topo) val) x)
```

8 Semantics

In this section we define the semantics for the formulas defined in `Syntax.lhs` on both `TopoModel`'s and `S4KripkeModel`'s.

```
module Semantics where

import qualified Data.Set as S

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

Given a formula φ in our language along with a model $\mathfrak{M}$, a *semantics* is a definition of when $\mathfrak{M}$ makes φ *true*. The relation 'makes true' is often written using '$\models$', so we abbreviate the statement '$\mathfrak{M}$ makes φ true' as '$\mathfrak{M} \models \varphi$'.

```
class Semantics m where
    (|=) :: m -> Form -> Bool
```

In both of the below-defined instances of `Semantics`, the Boolean cases are defined in the same, standard way, so we will only comment on the key modal cases.

8.1 Kripke semantics

Given a *pointed* **S4** Kripke model (X, R, V, x), we define the following for all formulas φ in our modal language.

$$\begin{aligned}(X,R,V,x) \models \Box\varphi \;&:\Longleftrightarrow\; \forall y \in X(xRy \Rightarrow (X,R,V,y) \models \varphi)\\ (X,R,V,x) \models \Diamond\varphi \;&:\Longleftrightarrow\; (X,R,V,x) \models \neg\Box\neg\varphi\end{aligned}$$

Given an **S4** Kripke model (X, R, V) (without a point), we also define the following for all formulas φ in our modal language.

$$(X,R,V) \models \varphi \;:\Longleftrightarrow\; \forall x \in X((X,R,V,x) \models \varphi)$$

```haskell
instance (Eq a, Ord a) => Semantics (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))
    (|=) pointedModel (Box phi) = all (\w' -> PS4KM kripkeModel w' |= phi) imageOfWorld
      where
        (PS4KM kripkeModel world) = pointedModel
        S4KM kripkeFrame _ = kripkeModel
        S4KF _ relation = kripkeFrame
        imageOfWorld = world `imageIn` relation

instance (Eq a, Ord a) => Semantics (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
```

8.2 Topo-semantics

Given a *pointed* topomodel (X, τ, V, x), we define the following for all formulas φ in our modal language.

$$(X, R, V, x) \models \Box\varphi \;:\Longleftrightarrow\; \exists U \in \tau(x \in U \text{ and } \forall y \in U((X, \tau, V, y) \models \varphi))$$
$$(X, \tau, V, x) \models \Diamond\varphi \;:\Longleftrightarrow\; (X, \tau, V, x) \models \neg\Box\neg\varphi$$

Given a topomodel (X, τ, V) (without a point), we also define the following for all formulas φ in our modal language.

$$(X, \tau, V) \models \varphi \;:\Longleftrightarrow\; \forall x \in X((X, \tau, V, x) \models \varphi)$$

```haskell
instance (Eq a) => Semantics (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))
    (|=) 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) => Semantics (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
```

9 Model conversion

In this sections, we implement a method for converting `S4KripkeModel`'s to `TopoModel`'s and vice-versa. We follow the construction described in [Pac17, 22-23].

```
module ModelConversion where

import Data.Set (cartesianProduct, member, singleton)
import qualified Data.Set as S

import KripkeModels (PointedS4KripkeModel (PS4KM), S4KripkeFrame (S4KF), S4KripkeModel (
    S4KM))
import SetTheory (closeUnderUnion, imageIn)
import TopoModels (PointedTopoModel (PointedTopoModel), TopoModel (TopoModel))
import Topology (TopoSpace (TopoSpace), closure)
```

Given a set-relation pair $\mathbf{X} := (X, R)$, an *upset* is a subset $S \subseteq X$ that satisfies the following for all $x, y \in X$.

$$(x \in S \text{ and } xRy) \text{ implies } y \in S$$

The term 'upset' is used because orders are often depicted using Hasse diagrams where xRy is depicted by the point y being above x, connected by a line. We denote the set of all upsets of $\mathbf{X}$ by $\mathtt{Up}(\mathbf{X})$.

Given an **S4** Kripke frame $\mathbf{X} := (X, R)$, it is a well known fact that $(X, \mathtt{Up}(\mathbf{X}))$ is a topological space. What is more, for all modal formulas φ, all valuations V on X, and all points $x \in X$, we have

$$(X, R, V, x) \models \varphi \Leftrightarrow (X, \mathtt{Up}(\mathbf{X}), V, x) \models \varphi$$

Observe how the '$\models$' on the left-hand-side is a relational semantics while on the right-hand-side it is a topo-semantics.

Below we implement this conversion from `S4KripkeModel`'s to `TopoModel`'s. Since we are working with finite models, we can generate all upsets by closing all of the principle upsets under unions (along with the empty set).

```
toTopoSpace :: (Ord a) => S4KripkeFrame a -> TopoSpace a
toTopoSpace kripkeFrame = TopoSpace carrier opens
  where
    S4KF carrier relation = kripkeFrame
    nonEmptyUpsets = closeUnderUnion $ S.map (`imageIn` relation) carrier
    opens = S.insert S.empty nonEmptyUpsets
```

Now we turn to the other conversion.

Given a topospace ($\mathbf{X} := (X, \tau)$), the *specialisation order* $R_{\mathbf{X}}$ *on* $\mathbf{X}$ is defined as follows for all $x, y \in \mathbf{X}$.

$$xR_{\mathbf{X}}y \;:\Longleftrightarrow\; y \in \overline{\{x\}}$$

It follows quite easily that this relation is reflexive and transitive, implying that $(X, R_{\mathbf{X}})$ is an **S4** Kripke frame.

Similarly to the other conversion, we get the following for all modal formulas φ, all valuations V on X, and all points $x \in X$.

$$(X, \tau, V, x) \models \varphi \Leftrightarrow (X, R_{\mathbf{X}}, V, x) \models \varphi$$

```
toS4KripkeFrame :: (Ord a) => TopoSpace a -> S4KripkeFrame a
toS4KripkeFrame topoSpace = S4KF space relation
  where
    (TopoSpace space _) = topoSpace
    relation = S.filter (\(x, y) -> y `member` closure (singleton x) topoSpace) (
        cartesianProduct space space)
```

Since the carrier sets and valuations remain unchanged, we can extend these conversions to (pointed) models.

```
toTopoModel :: (Ord a) => S4KripkeModel a -> TopoModel a
toTopoModel (S4KM frame valuation) = TopoModel (toTopoSpace frame) valuation

toS4KripkeModel :: (Ord a) => TopoModel a -> S4KripkeModel a
toS4KripkeModel (TopoModel topoSpace valuation) = S4KM (toS4KripkeFrame topoSpace)
    valuation

toPointedTopoModel :: (Ord a) => PointedS4KripkeModel a -> PointedTopoModel a
toPointedTopoModel (PS4KM kripkeModel point) = PointedTopoModel (toTopoModel kripkeModel)
    point

toPointedS4KripkeModel :: (Ord a) => PointedTopoModel a -> PointedS4KripkeModel a
toPointedS4KripkeModel (PointedTopoModel topoModel point) = PS4KM (toS4KripkeModel
    topoModel) point
```

10 Testing

```
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))))
```

11 Conclusion

Finish me.

References

[Pac17] Eric Pacuit. *Neighborhood Semantics for Modal Logic*. Springer Cham, 2017.