A Haskell implementation of translations between Distributive Lattices and Priestley Spaces

Giacomo de Antonelllis, Estel Koole, Marco de Mayda,
Edoardo Menorello, Max Wehmeier

Tuesday 25th March, 2025

Abstract

The aim of this project is to provide a Haskell implementation for the Duality-theoretic translations between Distributive lattices and Priestley spaces.

In order to do so in Section 1 we define mappings. In section 2 we encode Partially ordered sets in Haskell, making use of the Data.set library. In section 2 we offer an implementation for Distributive lattices based on the on partially ordered set onstruction. In section 4 we define Priestley spaces in Haskell, making use of a prexisting Topology library. In section 5 we effectively implement translations between Priestley spaces and Distributive lattices.

The repository of the project can be found at https://github.com/maxwehmi/functional-duality.

Contents

1 Mappings

As in most of mathematics, maps (and more specifically isomorphisms) are of great importance to our project. As usual in mathematics, we implement maps as a set of pairs.

```
module Mapping where

import Data.Set (Set, map, size, elemAt, filter)

type Map a b = Set (a,b)
```

Of course, we also want to evalutate maps and get preimages. For images, we are given a map and an element x in the domain. Firstly, we calculate the set of second elements, such that the first element in the mapping is x and similarly for preimages.

```
getImages :: (Ord a, Ord b) => Map a b -> a -> Set b
getImages mapping x = Data.Set.map snd $ Data.Set.filter (\ (y,_) -> x == y) mapping

getPreimages :: (Ord a, Ord b) => Map a b -> b -> Set a
getPreimages mapping y = Data.Set.map fst $ Data.Set.filter (\ (_,z) -> z == y) mapping
```

Using these functions, we can check if a given set of pairs is actually a map, i.e. every element in its domain has exactly one image. Similarly, we can check bijectivity by confirming that the preimage of every element in the range is a singleton.

```
checkMapping :: (Ord a, Ord b) => Set a -> Map a b -> Bool
checkMapping sa mapping = all (\ x -> size (getImages mapping x) == 1) sa

checkBijective :: (Ord a, Ord b) => Set b -> Map a b -> Bool
checkBijective sb mapping = all (\ y -> size (getPreimages mapping y) == 1) sb
```

After confirming that the set of pairs is actually a map and bijective, we can evaluate it for a given point or calculate the preimage. To avoid errors, these functions should only be used after checking well-definedness and/or bijectivity.

```
getImage :: (Ord a, Ord b) => Map a b -> a -> b
getImage mapping x | size (getImages mapping x) == 1 = elemAt 0 (getImages mapping x)
                   | otherwise = error "Given Relation is not a mapping"

getPreimage :: (Ord a, Ord b) => Map a b -> b -> a
getPreimage mapping y | size (getPreimages mapping y) == 1 = elemAt 0 (getPreimages mapping
     y)
                      | otherwise = error "Either no or too many preimages"
```

2 Partially ordered sets

This section is devoted to the construction of posets. A poset $(P, \leq)$ is a structure such that P is a set and $\leq$ is a partial order, that is, $\leq$ is reflexive, transitive and antisymmetric.

We import the standard library for sets, Data.Set, in order to be able to work with objects that behave like sets and we start by defining the `OrderedSet` data type for sets equipped with a relation.

An object (P, R) of type `OrderedSet a`, is not necessarily a partially ordered set, therefore we need some helper functions in order to transform R in to a partial order.

```
module Poset where

import qualified Data.Set as Set
```

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

data OrderedSet a = OS {set :: Set.Set a, rel :: Relation a}
    deriving (Eq, Ord,Show)
```

2.1 Well-definedness

Firstly we need to check given an object of type `OrderedSet a` , the relation is indeed a subset of the cartesian product of the carrier set, as the implementation of `OrderedSet` does accept cases which are not of this sort.

We shall call a relation R "well-defined with respect to a set P" iff $R \subseteq P \times P$. We shall just say "well-defined" when the carrier set is clear form the context.

In order to check well-definedness of a relation we shall fisrt define the function `tuplesUnfold`, which "unfolds" the tuples of a set of tuples, i.e. a relation.

The implementation follows this idea: given a relation R, we get the list of the first elements in all the tuples by mapping `fst` to all the elements of R. We do the same with `snd` in order to get the list of all the second elements of the tuples. We then join these lists and turn the resulting list into a set, which then gives us the set of elements of of the tuples of R.

```
tuplesUnfold :: Ord a => Relation a -> Set.Set a
tuplesUnfold r = Set.fromList (Prelude.map fst (Set.toList r) ++ Prelude.map snd (Set.
    toList r))
```

Using `tuplesUnfold` we can now easily check for well-definedness.

```
checkRelationWellDef :: Ord a => OrderedSet a -> Bool
checkRelationWellDef (OS s r) = tuplesUnfold r `Set.isSubsetOf` s
```

Moreover if the relation is not well-defined relation, we might want to force it to be. The following function takes care of this:

```
-- this maybe could've been done more simply, but idk it seems to work like this
forceRelation :: Ord a => OrderedSet a -> OrderedSet a
forceRelation (OS s r)
 | checkRelationWellDef (OS s r) = OS s r
 | otherwise = OS s ( r `Set.difference` Set.fromList (
                                                        [(x,y) | (x,y) <- Set.toList r, x `
                                                            Set.member` (tuplesUnfold r `
                                                            Set.difference` s)]
                                                        ++
                                                        [(x,y) | (x,y) <- Set.toList r, y
                                                            `Set.member` (tuplesUnfold r `
                                                            Set.difference` s)]
                                                      )
                    )
```

2.2 Closures

Secondly, given an object of type `OrderedSet a` , the relation need not to be reflexive and transitive, as, again, the implementation of `OrderedSet` does accept cases which are not of this sort. Therefore we shall define some functions in order to close the relation of any object of type `OrderedSet a` ,reflexivity and transitivity.

Moreover, we shall define some functions in order to check whether the relation of an object of type `OrderedSet a` is reflexive and transitive.

2.2.1 Closure under reflexivity

```
closureRefl :: Ord a => OrderedSet a -> OrderedSet a
closureRefl (OS s r) = OS s (r `Set.union` Set.fromList [(x,x)| x <- Set.toList s])
```

We now define the following two functions to check if given an object of type `OrderedSet a` , its relation is reflexive: the first makes use of the `closureRefl` function, the second works independently.

```
checkRefl :: Ord a =>  OrderedSet a -> Bool
checkRefl os = os == closureRefl os

checkReflAlt :: Ord a =>  OrderedSet a -> Bool
checkReflAlt (OS s r) = all (\x ->  (x, x) `Set.member` r) s
```

2.2.2 Closure under transitivity

The transitive closure requires more working: let (P, R) be an object of type `OrderedSet a` .

Firstly, we define the helper function `transPair`, to check if, given any x, z, there is a y such that $xRy \wedge yRz$.

```
transPair ::  Ord a => a -> a -> OrderedSet a -> Bool
transPair x z (OS s r)=  any (\y -> (x, y) `Set.member` r && (y,z) `Set.member` r) s
```

Now, we add to the relation any pair which satisfies `transPair`, so that we have "one-step" transitivity.

```
transStep :: Ord a => OrderedSet a -> OrderedSet a
transStep (OS s r) = OS s (r `Set.union` Set.fromList [(x,z) | x <- Set.toList s, z <- Set.
    toList s, transPair x z (OS s r)])
```

Now that with `transStep` we have enlarged our relation, new pairs which satisfy `transPair` might arise. Therefore in order to fully close the relation under transitivity we need to "recursively" apply the `transStep` function to our object (P, R) of type `OrderedSet a` until we reach a "tr"(P, R) such that `transStep tr(P,R) == tr(P,R)`.

`closureTrans` is the function that does exaxtly this:

```
closureTrans :: Ord a => OrderedSet a -> OrderedSet a
closureTrans  currentSet =
        let recursedSet = transStep currentSet
        in if recursedSet == currentSet
            then currentSet
            else closureTrans recursedSet
```

We now define the following two functions to check if given an object of type `OrderedSet a` , its relation is transitive: the first makes use of the `closureTrans` function, the second works independently.

```
checkTrans :: Ord a => OrderedSet a -> Bool
checkTrans os = os == closureTrans os

checkTransAlt :: Ord a => OrderedSet a -> Bool
checkTransAlt (OS _ r) = all (\(x, _, z) -> Set.member (x, z) r) [(x, y, z) | (x, y) <- Set
    .toList r, (y', z) <- Set.toList r, y == y']
```

2.3 Forcing Antisymmetry

For the same reason as before, given an object of type `OrderedSet a` , the relation need not to be antisymmetric. But it is not possible to just "close" a relation under antisymmetry, as all the symmetric couples whose elements are different from each other need to be somehow eliminated. Therefore we shall define some functions to do just that.

Moreover we shall define some functions that given an object of type `OrderedSet a` , will check whether the relation is antisymmetric.

There are two ways, among others, to force anti-symmetry on a relational structure (P, R): the first consists in eliminating all symmetric pairs from the relation R, the second instead creates a quotient of P based on the clusters of symmetric pairs of R.

We shall implement both as both have some advatages and some disadvantages to them and one may be preferable to the other depending on the situation at hand.

First implementation Given an object (P, R) of type `OrderedSet a`, we eliminate all the symmetric pairs in R. That is we construct a new relation $R^* \subseteq R$ such that if $x \neq y$ and $(x, y) \in R$ and $(y, x) \in R$, then $\neg((x, y), (y, x) \in R^*)$.

```
forceAntiSym :: Ord a => OrderedSet a -> OrderedSet a
forceAntiSym (OS s r)
 | checkAntiSym (OS s r) = OS s r
 | otherwise = OS s (r `Set.difference` Set.fromList [(x,y)| x <- Set.toList s,
                                                            y <- Set.toList s,
                                                            x/= y && (y,x) `Set.member` r
                                                                && (x,y) `Set.member` r]
                    )
```

- Advantages: it does not modify the carrier set (eg `Set.size`, the cardinality, will remain the same after the procedure).
- Disadvantages: it could significantly reduce the numbers of pairs in the relation: in particular every (non-reflexive) cycle is eliminated from the relation.

Second Implementation Given any (P, R) of type `OrderedSet a`, we can quotient the set P on the symmetric points, i.e. merge the *vertex* that see each other into a cluster. That is, for any $x \in P$ we define the equivalence class $[x]_s$ as the set $\{y \in P | y \neq x \wedge xRy \wedge yRx\}$.

```
quotientAntiSym :: Ord a => Set.Set a -> Relation a -> Set.Set a
```

```
quotientAntiSym s r = s `Set.difference` Set.fromList [x| (x,y) <- Set.toList r, (y,x) `Set
    .member` r, x /= y, y < x]

forceAntiSymAlt :: Ord a => OrderedSet a -> OrderedSet a
forceAntiSymAlt (OS s r) = forceRelation $  OS (quotientAntiSym s r) r
```

- Advantages: this does preserve logical properties
- Disadvantages:
  - this does change the carrier set: from P we go to P/s, the quotient of P under the equivalence relation based on symmetry.
  - this operation may shrink significatly the size of the carrier set,in particular if done after taking its transitive closure.

Preservation of properties We need to make sure that forcing anti-symmetry in our two implementation does not make us loose an existing property of the relations. While it is immediate that the second implementation does so, this is not the case for the first implementation.

Of course the first implementation preserves reflexivity: only couples (x, y) such that $x \neq y$ are removed. The non trivial preservation concerns transitivity: we shall therefore prove the following proposition.

Proposition: Let P be any set and $R \subseteq P \times P$ any relation defined on that set. For any relation R, we define the antisymmetric forcing of R, $R^\dagger$ as:

$$R^\dagger = \begin{cases} R & \text{if } R \text{ is anti-symmetric} \\ R \setminus \{(x,y) \mid (x,y) \in R \wedge (y,x) \in R \wedge x \neq y\} & \text{otherwise} \end{cases}$$

Then, if R is transitive also $R^\dagger$ is transitive.

- *Proof*: Let that $R \subseteq P \times P$ be any transitive relation on some fixed but arbitrary set P.

  Let $R^\dagger$ be as in the above definition. Take now any $x, y, z \in P$ such that $xR^\dagger y \wedge yR^\dagger z$. We need to show that $xR^\dagger z$.

  Now since $R^\dagger \subseteq R$ by definition, $xRy \wedge yR^\dagger z$. Therefore xRz. Suppose now towards contradiction that $(x, z) \not\in R^\dagger$. Therefore zRx and $z \neq x$. But then, since yRz, by transitivity of R, yRx.

  Clearly, since we assumed $yR^\dagger z$ and $(x, z) \not\in R^\dagger$, $y \neq x$. But then, by definition of $R^\dagger$, $(x, y) \not\in R^\dagger$, which is a contradiction to our assumpion. Therefore $(x, z) \in R^\dagger$, which is what we had to show.

Now since the definition of `forceAntiSym` corresponds to that of $R^\dagger$, we can conclude that, given any `OS s r` of type `OrderedSet a`, if we denote by `OS s' r'` the result of `forceAntiSym transClosure (OS s r)`, `r'` is transitive.

Finally, we define the following function in order to chek for any given an object of type `OrderedSet a`, whether its relation is antisymmetric.

```
checkAntiSym :: Ord a => OrderedSet a -> Bool
checkAntiSym  (OS _ r) = not (any (\(x,y) -> x /= y && (y, x) `Set.member` r) r)
```

2.4 From ordered sets to posets

Finally, given all the passages we have gone through in this section, we are able to define functions that given any object of type `OrderedSet a`, will transform it into a poset and check whether it is indeed a poset.

For the first task we take two approaches:

closurePoset

```
closurePoSet :: Ord a => OrderedSet a -> OrderedSet a
closurePoSet os
 | not (checkRelationWellDef os) = error "relation isn't well-defined"
 | not (checkAntiSym os)  = error "relation isn't anti-symmetric"
 | not (checkAntiSym $ closureTrans os) = error "relation looses anti-symmetry when
     transitively closed"
 | otherwise = closureTrans $ closureRefl os
```

Forcing the poset For an object of type `OrderedSet a` to be a poset it suffices to first take the reflexive closure of its relation, then its transitive closure and then force antisymmetry on this new relation. That is it suffices to apply to such object, the following function:

```
forcePoSet :: Ord a => OrderedSet a -> OrderedSet a
forcePoSet  = closureRefl .  forceAntiSym .  closureTrans . forceRelation

-- forceRleation is reduntant here since it is inside forceAntiSymAlt
forcePosetAlt :: Ord a => OrderedSet a -> OrderedSet a
forcePosetAlt = closureRefl .  forceAntiSymAlt .  closureTrans
```

Checking the poset In order to check if an object of type `OrderedSet a` is indeed a poset, we define the following function:

```
checkPoset :: Ord a => OrderedSet a -> Bool
checkPoset x = checkRefl x && checkTrans x && checkAntiSym x && checkRelationWellDef x
```

2.5 Examples

Here are some toy examples to work with.

```
os6 :: OrderedSet Integer
os6 = OS (Set.fromList [1, 2, 3])
         (Set.fromList [(1,1), (2,2), (3,3), (1,2), (2,3), (1,3)])

os7 :: OrderedSet Integer
os7 = OS (Set.fromList [1, 2, 3])
         (Set.fromList [(1,1), (2,2), (3,3), (1,2), (2,1), (1,3), (3,1)])

os8 :: OrderedSet Integer
os8 = OS (Set.fromList [1, 2, 3])
         (Set.fromList [(1,1), (2,2), (3,3), (1,2), (2,3), (3,2)])

os9 :: OrderedSet Integer
os9 = OS (Set.fromList [1, 2, 3])
         (Set.fromList [(1,1), (2,2), (3,3), (1,2), (2,3), (3,2), (1,3)])

os10 :: OrderedSet Integer
os10 = OS (Set.fromList [1, 2, 3])
          (Set.fromList [(1,1), (2,2), (3,3), (1,2), (2,1)])

os11 :: OrderedSet Integer
```

```
os11 = OS (Set.fromList [1, 2, 3])
          (Set.fromList [(2,2), (3,3), (1,2), (2,3), (1,3)])
os12 :: OrderedSet Integer
os12 = OS (Set.fromList [1, 2, 3])
          (Set.fromList [(2,2), (3,3), (1,2), (2,3)])

os13 :: OrderedSet Integer
os13 = OS (Set.fromList [1, 2, 3])
          (Set.fromList [(2,2), (3,3), (1,2), (2,3), (3,1)])

os14 :: OrderedSet Integer
os14 = OS (Set.fromList [1, 2, 3, 4, 5])
          (Set.fromList [(1,1), (2,2), (3,3), (4,4), (5,5),
                         (1,2), (2,3), (1,3), (4,5), (1,4), (2,5)])

os15 :: OrderedSet Integer
os15 = OS (Set.fromList [1, 2, 3, 4, 5, 6])
          (Set.fromList [(1,1), (2,2), (3,3), (4,4), (5,5), (6,6),
                         (1,2), (2,3), (3,4), (4,5), (5,6), (1,3), (1,4), (1,5), (1,6)])

os16 :: OrderedSet Integer
os16 = OS (Set.fromList [1, 2, 3, 4, 5])
          (Set.fromList [(1,1), (2,2), (3,3), (4,4), (5,5),
                         (1,2), (2,3), (1,3), (4,5), (5,4)])

os17 :: OrderedSet Integer
os17 = OS (Set.fromList [1, 2, 3])
          (Set.fromList [(1,1), (2,2), (3,3), (1,2), (2,1), (1,3), (3,1)])

os18 :: OrderedSet Integer
os18 = OS (Set.fromList [1, 2, 3])
          (Set.fromList [(1,1), (2,2), (3,3), (1,2), (2,1)])

os19 :: OrderedSet Integer
os19 = OS Set.empty Set.empty


myOS :: OrderedSet Integer
myOS = OS (Set.fromList [1..4]) (Set.fromList [(1,4), (4,5), (5,4),(4,1),(2,1),(2,2),(3,3)
    ,(3,1),(1,1),(4,4)])

emptyRelOS :: OrderedSet Integer
emptyRelOS = OS (Set.fromList[1..4]) (Set.fromList [])

myCircle :: OrderedSet Integer
myCircle = OS (Set.fromList [1,2,3]) (Set.fromList [(1,2),(2,3),(3,1)])
```

3 Distributive Lattices

```
module DL where

import Poset
import qualified Data.Set as Set
import qualified Data.Maybe as M
```

This section is dedicated to Distributive Lattices. A lattice is a poset P such that for every $a, b \in P$ the greatest lower bound of $\{a, b\}$ (the meet of $a, b : a \wedge b$) is in P and least upper bound of $\{a, b\}$ (the join of $a, b : a \vee b$) is in P.

On top of this, a distributive lattice is a lattice whose meet and join satisfiy the two distributive laws: if $(L, \wedge, \vee)$ is a lattice, then:

1. $\forall a, b, c \in L, a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
2. $\forall a, b, c \in L, a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

We define the data type of lattices in the following manner:

```
data Lattice a = L {
    carrier :: OrderedSet a,
    meet :: a -> a -> a,
    join :: a -> a -> a
    }
```

Not every object of type lattice is an actual lattice in the mathematical sense: in particular three conditions have to be met for an object "l" of type "Lattice a", to be an actual lattice.

- Since we are working with finite structures, each lattice is a bound lattice.Therefore given an object l of type Lattice a, the first thing to check is whether the object "carrier l" has a maximal and a least element.
- The object "meet l" has to be defined on every two elements of the underlying set of "carrier l" and for every such two elements it has to return their greatest lower bound.
- The object "join l" has to be defined on every two elements of the underlying set of "carrier l" and for every such two elements it has to return their least upper bound.

The aim of the following functions is to ensure that the objects of type "Lattice a" behave as desired.

The 'top' and 'bottom' functions will give the top and bottom elements of a lattice, and 'isTop' and 'isBottom' checks whether some element in the lattice is actually the top or bottom element. Furthermore, the 'checkBoundedness' function will check the existence of a top and bottom element in a lattice.

```
isTop :: Ord a => Lattice a -> a -> Bool
isTop l x = all (\y -> (y, x) `elem` rel k) (set k)
    where
     k = carrier l

-- when lattice is a poset, this should return a singleton with the top,
-- or empty set with no top, so nothing
top :: Ord a =>Lattice a -> Maybe a
top l = Set.lookupMax (Set.filter (isTop l) (set $ carrier l))

isBot :: Ord a => Lattice a -> a -> Bool
isBot l x = all (\y -> (x,y) `elem` rel k) (set k)
    where
     k = carrier l

bot :: Ord a =>Lattice a -> Maybe a
bot l = Set.lookupMin (Set.filter (isBot l) (set $ carrier l))


-- The four above functions are used to check if a given element of a given lattice is its
    top/bottom element and to obtain the top/bottom element of a lattice if it exists

checkBoundedness :: Ord a => Lattice a -> Bool
checkBoundedness l = M.isJust (top l) && M.isJust (bot l)
```

We want to work with distributive lattices. A lattice L is distributive if for any $a, b, c \in L$ the following laws hold:

- Law 1: $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
- Law 2: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

The function 'checkDistributivity' checks whether a lattice is distributive. Furthermore, law 1 and 2 are equivalent and so the function will only check law 1, which is sufficient.

```
checkDistributivity :: Eq a => Lattice a -> Bool
checkDistributivity (L (OS s _) m v) = and
                        [(a `m` (b `v` c) == (a `m` b) `v` (a `m` c))
                        | a <- Set.toList s, b <- Set.toList s, c <- Set.toList s]
```

In the definition of our lattice, the lattice comes with functions called 'meet' and 'join'. We want a lattice to be closed under meet and join and thus, we use 'checkClosedMeetJoin' as a function to check this. Let L be a lattice. For two arbirtray elements $a, b \in L$, we want that (meet a b) $\in L$ and (join a b) $\in L$.

```
checkClosedMeetJoin :: Ord a => Lattice a -> Bool
checkClosedMeetJoin l = all (\x -> pairMeet x `elem` lSet ) j -- x is arb. pair in l
                        &&
                        all (\x -> pairJoin x `elem` lSet) j
    where
        lSet = set $ carrier l
        j = Set.cartesianProduct lSet lSet -- sets of pairs
        pairMeet = uncurry (meet l)
        pairJoin = uncurry (join l)
```

Furthermore, we desire a function that checks whether some lattice is well-defined, meaning that the function 'meet' and 'join' that come with our lattice correspond with the actual meet and join in the ordered set underlying the lattice. That is what the function 'checkMeetJoinMakeSense' does.

```
checkMeetJoinMakeSense :: Ord a => Lattice a -> Bool
checkMeetJoinMakeSense l = and [Just (meet l x y) == findMeet l x y
                                && Just (join l x y) == findJoin l x y
                                |x <- Set.toList (set (carrier l)), y <- Set.toList (set (
                                    carrier l))]
```

Besides using the functions the lattice comes with for finding meets and joins, namely 'meet' and 'join', we also need functions that will find the actual meet and join in the lattice by looking at the poset underlying the lattice. These functions are called 'findMeet' and 'findJoin'.Applicative

Let L be a lattice and let $a, b \in L$ be arbitrary elements. The meet $a \wedge b$ is defined as the greatest lower bound, and the join $a \vee b$ as the least upper bound. That is why we use helper functions to find the upper bounds and lower bounds of a and b, namely 'upperBounds' and 'lowerBounds'.

Subsequently, the functions 'findGreatest' and 'findLeast' will find the greatest or least element of a subset of some lattice L with respect to the ordering inside L.

Now suppose that our lattice L was not a lattice after all, meaning that L is not closed under meet and join. Then either the set of upper bounds or lower bounds will be empty, or there will be no greatest or least element in the set. In those cases 'findMeet' and 'findJoin' return 'Nothing'. In cases the functions are succesful, it will return 'Just x' where x is the meet or join of the two input elements.

```
-- Helper functions for checkClosedMeetJoin
-- finds meet & join in lattice, independant of
findMeet :: Ord a => Lattice a -> a -> a -> Maybe a
findJoin :: Ord a => Lattice a -> a -> a -> Maybe a
-- find all lower bounds, and take the maximum
```

```
findMeet l x y = findGreatest (carrier l) (lowerBounds (carrier l) x y)
findJoin l x y = findLeast (carrier l) (upperBounds (carrier l) x y)

-- For some ordered set (X, <=), find the greatest element of some subset S of X
findGreatest :: Ord a => OrderedSet a -> Set.Set a -> Maybe a
-- findGreatest (OS s r) s = if all (\x -> (x, greatest) `Set.member` r) (Set.toList s)
    then Just greatest else Nothing
                    -- where greatest = foldr (\new old -> (if (old, new) `Set.member` r
                        then new else old)) (head $ Set.toList s) s
findGreatest os s = Set.lookupMax $ Set.filter (\x -> all (\y -> (y, x) `Set.member` rel os
    ) s) s

findLeast :: Ord a => OrderedSet a -> Set.Set a -> Maybe a
findLeast os s = Set.lookupMax $ Set.filter (\x -> all (\y -> (x, y) `Set.member` rel os) s
    ) s

-- set of elements above a1 and a2
upperBounds :: Ord a => OrderedSet a -> a -> a -> Set.Set a
upperBounds os a1 a2 = Set.fromList [c | c <- Set.toList $ set os, (a1, c) `Set.member` rel
     os &&
                                                              (a2, c) `Set.member` rel os]

lowerBounds :: Ord a => OrderedSet a -> a -> a -> Set.Set a
lowerBounds os a1 a2 = Set.fromList [c | c <- Set.toList $ set os, (c, a1) `Set.member` rel
     os &&
                                                              (c, a2) `Set.member` rel os]
```

To check whether some object of type 'Lattice' is actually a lattice, we check whether it is well-defined with respect to 'meet' and 'join' and check whether it is closed under binary meets and joins.

```
-- check whether actual meet & join align with functions, check whether closed under meet
    and join
checkLattice :: Ord a => Lattice a -> Bool
checkLattice l = checkMeetJoinMakeSense l && checkClosedMeetJoin l
```

A distributive lattice L is a bounded lattice, which follows the distributivity laws. In our function 'checkDL', we check whether an object of type 'Lattice' is a lattice, bounded and distributive.

As we are working in the finite case, any lattice is bounded as the finite join of all the elements would be the top element, and the finite meet of all elements the bottom element.

```
checkDL :: Ord a => Lattice a -> Bool
checkDL l =         checkLattice l
                     &&
                    checkBoundedness l
                     &&
                    checkDistributivity l
```

Lastly, we want to be able to go from the type 'OrderedSet' to the type of 'Lattice'. In our function makeLattice the ordered set given as input is used as the structure of the lattice and the functions for 'meet' and 'join' are added.

Still to implement is to add a check that makes sure the input ordered set is closed under meet and joins.

```
fromJust :: Maybe a -> a
fromJust (Just x) = x
fromJust Nothing = error "Sorry, but your poset is not closed under meet and joins"

makeLattice :: Ord a => OrderedSet a -> Lattice a
makeLattice os = L os (\x y -> fromJust $ findMeet preLattice x y) (\x y -> fromJust $
    findJoin preLattice x y)
                where preLattice = L os const const -- give it two mock functions
```

Below are a few test cases. 'myos' is a poset. Furthermore, 'mylat1' is a non well-defined lattice, meaning that the functions for 'meet' and 'join' do not coincide with 'findMeet' and 'findJoin'. Lastly, mylat is a lattice.

```
myos :: OrderedSet Int
myos = Poset.closurePoSet $ OS (Set.fromList [1,2,3,4, 5]) (Set.fromList [(1,2), (2,4),
    (1,3),(3,4),(4,5)])

-- not well-defined lattice
mylat1 :: Lattice Int
mylat1 = L myos (-) (+)

mylat :: Lattice Int
mylat = makeLattice myos

myos2 :: OrderedSet Int
myos2 = Poset.closurePoSet $ OS (Set.fromList [1,2,3,4,5]) (Set.fromList [(1,2), (2,4),
    (1,3),(3,4)])

mylat2 :: Lattice Int
mylat2 = makeLattice myos2
```

```
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use infix" #-}
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use infix" #-}
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use infix" #-}
module Priestley where
import Data.Set (Set, toList, fromList, intersection, union, difference, filter, map, size,
     elemAt, isSubsetOf, member, empty, cartesianProduct)
import Data.Bifunctor (bimap)
import DL
import Poset
import Mapping
```

4 Priestley Spaces

We introduce the main data types of this section.
In the definition of the types, we keep it as close as possible to their mathematical counterparts:

1. a Topological Space is a set X endowed with a collection of subsets of X τ.
   In particular it is required that $X, \emptyset$ are elements of τ, and τ is closed under finitary intersections and arbitrary unions.
   Notice that, since we are working with finite cases, finitary and arbitrary unions (and intersections) coincide.

2. a Priestley space is a Topological Space endowed with a partial order $\leq$ on its carrier set. Moreover, it satisfies the following Priestley Separation Axiom:

   $$\text{PSA:} x \not\leq y \rightarrow \exists C \subseteq X((C = \uparrow C) \& (C \in \tau) \& ((X \setminus C) \in \tau) \& (x \in C) \& (y \notin C))$$

   Intuitively, for any x, y that are not related by $\leq$, there exists a upwards-closed set in the topology that separates them. Moreover, the complement of such set should also be in the topology.
   Elements of τ such that their complement in X also is in the topology are called "Clopen Sets".

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

data TopoSpace a = TS {
    setTS :: Set a,
    topologyTS :: Topology a
}

data PriestleySpace a = PS {
    setPS :: Set a,
    topologyPS :: Topology a,
    relationPS :: Relation a
}
```

4.1 Set-theoretic preliminaries

In order to deal effectively with topological spaces, we first define some Set-theoretic preliminary notions. In addition to the standard functions drawn from "Data.set" library we define new functions to compute the closure of a given set under arbitrary unions and intersections.
In both cases, we first define functions to perform a one-step intersection (resp. union) of a set with itself, and then iterate the function until the resulting set is identical to its own one-step intersection (resp. union) closure.

```
unionStep :: (Ord a) => Topology a -> Topology a
unionStep x = Data.Set.map (uncurry union) (cartesianProduct x x)
intersectionStep :: (Ord a) => Topology a -> Topology a
intersectionStep x = Data.Set.map (uncurry intersection) (cartesianProduct x x)

unionClosure :: (Eq a, Ord a) => Topology a -> Topology a
unionClosure y = do
                let cycle1 = unionStep y
                if y == cycle1
                then y
                else unionStep cycle1
intersectionClosure :: (Eq a, Ord a) => Topology a -> Topology a
intersectionClosure z = do
                let cycle1 = intersectionStep z
                if z == cycle1
                then z
                else intersectionStep cycle1
```

4.2 Topology basics

We now introduce some machinery to work on topological spaces, and in particular, to ensure our spaces respect the Priestley separation axiom.

The following function check whether a Topological space given as input respects the requirements spelled above.
It is assumed that the input is finite. In case the input does not respect the conditions, the second function adjusts the space so that it meets the requirements.

```
checkTopology :: Ord a => TopoSpace a -> Bool
checkTopology (TS space top) = member space top && member empty top && unionClosure top ==
    top && intersectionClosure top == top
```

```
fixTopology :: Ord a => TopoSpace a -> TopoSpace a
fixTopology (TS space top) = TS space fixedTop  where
    fixedTop  = fromList [space, empty] `union` unionClosure (intersectionClosure top)
```

The next functions allow us to extract from a given Priestley space its underlying set together with the topology, and its underlying carrier set together with its order.

```
getTopoSpace :: PriestleySpace a -> TopoSpace a
getTopoSpace p = TS (setPS p) (topologyPS p)

getOrderedSet :: PriestleySpace a -> OrderedSet a
getOrderedSet p = OS (setPS p) (relationPS p)
```

Next, we define a function to check whether a given Space really is a Priestley space.
We make use of some secondary helper functions:

1. the implementation for "implies" is routine,
2. the "allPairs" function extracts the totality of order pairs,

   from the carrier set X (this is required for the antecedent of the PSA axiom above),
3. the "clopUpset" function extracts all the elements from the topology which are both upward-closed and clopen by checking that their complement with respect to the space also is in the topology, and by checking that their are identical to their upwards-closure. Recall that a subset S of an ordered set is upward-closed if and only if whenever $x \in S$ and $x \leq y$ implies $y \in S$.
   This function makes use of the "upClosure" function, which computes the upwards-closure of any given set with respect to the given order.

The output of those is then fed to the "checkPSA" function, which then ensures the validity of the Priestley separation axiom for all points in X not related by $\leq$.

```
checkPriestley :: (Eq a, Ord a) => PriestleySpace a -> Bool
checkPriestley p = checkTopology (getTopoSpace p) && checkPoset (getOrderedSet p) &&
    checkPSA p

checkPSA :: (Eq a, Ord a) => PriestleySpace a -> Bool

checkPSA (PS space top order) = all (\ pair ->
 implies (pair `notElem` order) (any (\ open -> elem (fst pair) open
   && notElem (snd pair) open) (clopUp (PS space top order)) )) $ allPairs space

allPairs :: Set a -> [(a,a)]
allPairs space = [(x,y) | x <- toList space ,y <- toList space ]

implies :: Bool -> Bool -> Bool
implies x y = not x || y

clopUp :: Ord a => PriestleySpace a -> Topology a
clopUp (PS space top ord) = intersection (clopens top) (upsets top) where
        clopens = Data.Set.filter (\ x -> difference space x `elem` top)
        upsets = Data.Set.filter (\ y -> y == upClosure y ord)

upClosure :: (Eq a, Ord a) => Set a -> Relation a -> Set a
upClosure set1 relation = Data.Set.map snd (Data.Set.filter (\ x -> fst x `elem` set1 )
    relation) `union` set1
```

4.3 Dual maps and Isomorphisms

We present some functions to check basic properties of topological spaces. In particular, we want to be able to decide whether two spaces are isomorphic. this is going to come in handy

when exploring the duality with algebras.
We also present the first step towards implementing the algebra duality: keeping things brief, the set of Clopen Upset of a Priestley space is going to form a distributive lattice under the order induced by set-theoretic inclusion.
To this extent, we implement a function to extract an order based on set-theoretic inclusion between sets, which we canlater apply to the Clopen Upsets of our topology.
Next, we construct a lattice using the Clopen Upsets of our topological space and endowing this set with the desired inclusion-order. We make use of functions from the "DL" section to both construct the lattice and check it is distributive.

```
inclusionOrder :: Ord a => Topology a -> Relation (Set a)
inclusionOrder x = fromList [ (z ,y) |  z <- toList x, y <- toList x, isSubsetOf z y ]

clopMap :: Ord a => PriestleySpace a -> Lattice (Set a)
clopMap  ps = do
              let result = makeLattice $  OS (clopUp ps) (inclusionOrder (clopUp ps))
              if checkDL result then result else error "104!"
```

When working with Priestley Space, we want to be able to check if two given ones are "similar enough", i.e. isomorphic. This will become important when we want to confirm that a Priestley Space is isomorphic to the dual of its dual.
To check isomorphism, we have to be given two Priestley Spaces and a map between them. The map is an isomorphism, if it is actually a map, bijective, a homoemorphism on the topological spaces and an order isomorphism on the relations. If the map is an isomorphism, the spaces are isomorphic.

```
checkIso :: (Eq a, Ord a) => PriestleySpace a -> PriestleySpace a -> Map a a -> Bool
checkIso (PS sa ta ra) (PS sb tb rb) mapping = checkMapping sa mapping
    && checkBijective sb mapping
    && checkHomoemorphism ta tb mapping
    && checkOrderIso ra rb mapping
```

Assuming bijectivity (by laziness of &&), to check that the given map is a homeomorphism, we have to check that it is an open and continuous map, i.e. it maps opens to opens and the preimages of opens are also open. This means that applying the map to an open set in the topology of the domain should yield an element of the topology of the codomain, so applying it to the set of opens of the domain (its topology) should yield a subset of the opens of the codomain (its topology). Similarly, we check that the preimage of the topology of the codomain is a subset of the topology of the domain.

```
checkHomoemorphism :: (Ord a, Ord b) => Topology a -> Topology b -> Map a b -> Bool
checkHomoemorphism ta tb mapping =
    mapTop mapping ta `isSubsetOf` tb
    && premapTop mapping tb `isSubsetOf` ta
```

To apply the map to every open and thus every element of every open, we have to nest `Data.Set.map` twice. Again, we deal similarly with the preimages.

```
mapTop :: (Ord a, Ord b) => Map a b -> Topology a -> Topology b
mapTop mapping = Data.Set.map (Data.Set.map (getImage mapping))

premapTop :: (Ord a, Ord b) => Map a b -> Topology b -> Topology a
premapTop mapping = Data.Set.map (Data.Set.map (getPreimage mapping))
```

Lastly, it remains the check that the map is an order isomorphism, which means that two elements x, y of the domain satisfy $x \leq y$ in the domain iff $f(x) \leq f(y)$ in the codomain (here f is the map). This means that applying the map component wise to every pair of the relation

in the domain should yield the relation of the codomain and vice versa.

```
checkOrderIso :: (Ord a, Ord b) => Relation a -> Relation b -> Map a b -> Bool
checkOrderIso ra rb mapping = mapRel mapping ra == rb && premapRel mapping rb == ra
```

Similar to above, we have to nest `Data.Set.map` with `Data.Bifunctor.bimap` to apply the map to both components of all pairs in the relation.

```
mapRel :: (Ord a, Ord b) => Map a b -> Relation a -> Relation b
mapRel mapping = Data.Set.map (Data.Bifunctor.bimap (getImage mapping) (getImage mapping))

premapRel :: (Ord a, Ord b) => Map a b -> Relation b -> Relation a
premapRel mapping = Data.Set.map (bimap (getPreimage mapping) (getPreimage mapping))
```

5 Wrapping it up in an exectuable

We will now use the library form Section ?? in a program.

```
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
  print somenumbers
  print (map funnyfunction somenumbers)
  myrandomnumbers <- randomnumbers
  print myrandomnumbers
  print (map funnyfunction myrandomnumbers)
  putStrLn "GoodBye"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

6 Simple Tests

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where

import Basics
```

```
import Test.Hspec
import Test.QuickCheck
```

The following uses the HSpec library to define different tests. Note that the first test is a specific test with fixed inputs. The second and third test use QuickCheck.

```
main :: IO ()
main = hspec $ do
  describe "Basics" $ do
    it "somenumbers should be the same as [1..10]" $
      somenumbers `shouldBe` [1..10]
    it "if n > - then funnyfunction n > 0" $
      property (\n -> n > 0 ==> funnyfunction n > 0)
    it "myreverse: using it twice gives back the same list" $
      property $ \str -> myreverse (myreverse str) == (str::String)
```

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

Finally, we can see that [LW13] is a nice paper.

References

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.