

# A Haskell implementation of translations between Distributive Lattices and Priestley Spaces

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

Sunday 16<sup>th</sup> March, 2025

## Abstract

We give a toy example of a report in *literate programming* style. The main advantage of this is that source code and documentation can be written and presented next to each other. We use the `listings` package to typeset Haskell source code nicely.

## Contents

1	Mappings	2
2	Partially ordered sets	2
3	Priestley Spaces	5
4	Distributive Lattices	8
5	Wrapping it up in an executable	10
6	Simple Tests	11
7	Conclusion	12
	Bibliography	12

# 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 evaluate 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 codomain 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

Note that most operations presume to have ‘Ord’ instances. This has to do with Set.Set implementation.

"Most operations require that e be an instance of the Ord class." <https://hackage.haskell.org/package/containers-0.8/docs/Data-Set.html>

We can potentially work around this by transferring to lists, doing the checking on those, and then back, with some Set.toList trickery, for now leaving it like this, if we need to avoid assuming instances of Ord we can change it.

But I see everyone else’s code also pretty much always assumes Ord.

```

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)

-- I have changed the Relation a from "newtype ... Set ..." to "type ... Set.Set ..." as
-- Relation a is a type synonym and it was giving me problems with the typechecking in
-- other files.

-- I have changed the data type of OrderedSet a, in order to have functions to retrieve the
-- underlying set and the underlying relation of the OrderedSet.
-- Giacomo

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

-- this relies on the fact that "Set.fromList" eliminates duplicates, as Set shouldn't care
-- about them
checkRelationWellDef :: Ord a => OrderedSet a -> Bool
checkRelationWellDef (OS s r) = tuplesUnfold r 'Set.isSubsetOf' s

unsharedElements :: Ord a => Set.Set a -> Set.Set a -> Set.Set a
unsharedElements x y = (x 'Set.union' y) 'Set.difference' (x 'Set.intersection' y)

-- leaving this in case i'm saying something stupid, but i realized I can just use the
-- closure for the check condition...
checkReflAlt :: Ord a => OrderedSet a -> Bool
checkReflAlt (OS s r) = all (\x -> (x, x) 'Set.member' r) s

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

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']

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

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

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

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

-- This only adds "one step" transitivity, needs to be recursed till the it becomes
-- idempotent or something like this
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)])

-- current hacky solution
closureTrans :: Ord a => OrderedSet a -> OrderedSet a
closureTrans currentSet =
    let recursedSet = transStep currentSet
    in if recursedSet == currentSet

```

```

        then currentSet
        else closureTrans recursedSet

-- transitive closure can break anti-symmetry, so case was added
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 loses anti-symmetry when
    transitively closed"
  | otherwise = closureTrans $ closureRefl os

-- 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 <- Set.toList $
    unsharedElements s (tuplesUnfold r), y <- Set.toList s] ++ [(x,y) | y <- Set.toList $
    unsharedElements s (tuplesUnfold r), x <- Set.toList s]))

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])

```

**Proposition:** ‘forceAntiSymm \$ transClosure’, where ‘forceAntiSym’ of a relation  $R$ , call it  $R^\dagger$  is defined by:

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

*Proof:* Suppose  $R$  is any relation. We know the transitive closure  $R^+$  transitive. Let  $R^\dagger$  be the antisymmetric "closure" of  $R^+$ .

Suppose  $xR^\dagger y$  and  $yR^\dagger z$ . Since  $R^\dagger$  is generated only by removing points from  $R^+$ , we must've also have  $xR^+y, yR^+z$ . So by transitivity  $xR^+z$ .

If  $x = y$  we're quickly done, since then  $xR^\dagger z$ . Likewise if  $y = z$ . So suppose they aren't equal to each other.

Now suppose for contradiction Again by how  $R^\dagger$  was defined, we must've had  $zR^+x$ . (If we didn't, then  $(x,z) \notin \{(x,y) \mid (x,y) \in R \wedge (y,x) \in R \wedge x \neq y\}$ , and so we'd have  $(x,z) \in R^+ \setminus \dots$ ).

But then by transitivity of  $R^+$  we'd have  $yR^+x$ . But then  $(x,y) \in \{(x,y) \mid (x,y) \in R \wedge (y,x) \in R \wedge x \neq y\}$ , so by definition  $(x,y) \notin R^\dagger$ , i.e.

```

forcePoSet :: Ord a => OrderedSet a -> OrderedSet a
forcePoSet os = closureRefl $ forceAntiSym $ closureTrans os

-- forcePoSet :: Ord a => OrderedSet a -> OrderedSet a
-- forcePoSet currentSet =
--   let recursedSet = forceAntiSym $ closureTrans currentSet
--   in if recursedSet == currentSet
--     then closureRefl currentSet
--     else forcePoSet recursedSet

```

Here's some GPT-generated test sets

```

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 Priestley Spaces

```

{-# 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 Poset
import Mapping

--import qualified Data.IntMap as Data.set

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
}

checkTopology :: Ord a => TopoSpace a -> Bool
--Checks topology condition, really assumes the input is finite
checkTopology (TS space top) = member space top && member empty top && unionClosure top ==
  top && intersectionClosure top == top
  -- all (\ y -> all (\ x -> member (union x y) top ) top) top &&
  -- all (\ y -> all (\ x -> member (intersection x y) top ) top) top

fixTopology :: Ord a => TopoSpace a -> TopoSpace a
--makes the input space into a topological space
fixTopology (TS space top) = TS space fixedTop where
  fixedTop = fromList [space, empty] 'union' unionClosure (intersectionClosure top)

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

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

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

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

```

```

-- since we are only working with finite spaces, they are always compact

checkPSA :: (Eq a, Ord a) => PriestleySpace a -> Bool
-- i'll write this in the most retarded way possible for now, also, I figured, this always
  holds in the finite case anyway
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 ]
-- extracts the set of all orderedpairs form the carrier set (could also be done
  differently)
implies :: Bool -> Bool -> Bool
implies x y = not x || y
--usual implication shorthand

clopUp :: Ord a => PriestleySpace a -> Topology a
-- In the finite case those are just the upsets, I think it's at least honest to implement
  a general checker anyway
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)

-- takes upset
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

inclusionOrder :: Ord a => Topology a -> Relation (Set a)
-- Constructs (maybe) an order out of the clopen upsets of a given PS
inclusionOrder x = fromList [ (z ,y) | z <- toList x, y <- toList x, isSubsetOf z y ]
--This may give problems if we convert too many times from spaces to the clopup Dual, we
  could Use Data.Set.Monad and have a monad instance to avoid nesting sets
--into sets multiple times

{-
This goes commented since for whatever reason there VsCode won't allow me to import the DL
  file

clopMap :: PriestleySpace a -> Lattice a
clopMap = if {checkDBLattice $ makeLattice $ (\ x -> (\ y -> OS y inclusionOrder y) clopUp
  x) == True}
  then {makeLattice $ (\ x -> (\ y -> OS y (inclusionOrder y)) clopUp x) }
  | 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))
```

## 4 Disributive Lattices

```
module DL where

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

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

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)

-- TODO TEST THIS! -- xx Es
checkDistributivity :: Eq a => Lattice a -> Bool
-- check for any interpretation, that
-- law 1: (a v (b n c)) == (a v b) n (a v c))
-- law 2: and (a n (b n c)) == (a n b) v (a n c))
-- this function takes any tuple of three points in the lattice,
-- and checks whether law 1 & law 2 hold.
checkDistributivity (L (OS s _) m v) = and
  [(a 'm' (b 'v' c)) == (a 'm' b) 'v' (a 'm' c))
   && (a 'v' (b 'm' c)) == (a 'v' b) 'm' (a 'v' c))
  | a <- Set.toList s, b <- Set.toList s, c <- Set.toList s]

-- function now checks whether join and meet under function are in lattice
-- should still check whether coincides with actual meet and join in lattice
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)

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

-- checks whether an element is in the underlying set of the lattice
inLattice :: Ord a => a -> Lattice a -> Bool
inLattice x l = x 'Set.member' set (carrier l)

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

-- 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
-- needs top to be a maybe function
-- findMeet (L (OS set rel) _ _) x y = findGreatest (OS upperBounds (filter (\(v,w) -> v )
  rel))
  -- where upperBounds = filter (\z -> (z, x) 'Set.member' rel && (z, y)
    'Set.member' rel) (Set.toList set)
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]

-- test ordered Set
myos :: OrderedSet Int
-- <<<<<<< Poset-anti-symm-force
myos = Poset.closurePoSet $ OS (Set.fromList [1,2,3,4, 5]) (Set.fromList [(1,2), (2,4),
(1,3),(3,4),(4,5)])
--
-- =====
-- myos = Poset.closurePS $ OS (Set.fromList [1,2,3,4, 5]) (Set.fromList [(1,3), (1,2),
(2,4), (3,4),(4,5)])
-- >>>>>>> main

mylat1 :: Lattice Int
mylat1 = L myos (-) (+)

mylat :: Lattice Int
mylat = L myos (\x y -> fromJust $ findMeet mylat1 x y) (\x y -> fromJust $ findJoin mylat1
x y)

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

-- uses meet & join function inside lattice, for arb meets & joins
-- only works on finite lattices.
-- Boundedness of l is required for this function
arbMeet :: Lattice a -> a -> a -> a
arbJoin :: Lattice a -> a -> a -> a
--arbJoin l a1 a2 = rfold (\x y -> meet l $ x y) (fromJust $ top l) upperBs
-- where upperBs = [c | c <- (set carrier l), (c, a1) 'Set.member' (rel
carrier l),
--               (c, a2) 'Set.member' (rel carrier l)] -- all
--               elements above both a1 and a2

arbJoin = undefined
arbMeet = undefined

-- 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

-- I expect that the poset is already closed under meets 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

```

## 5 Wrapping it up in an executable

We will now use the library from 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](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.