All kinds of lattices

Oleg Grenrus

July 23, 2019

Partial ordered set, or poset for short is well used example in category theory books. Yet, posets are not too familiar for an average CT-curious Haskeller, yet they are easy to visualise! Let's do exactly that, and mention elementery bits of category theory. In particular, I'll scan over the six first chapters of *Category Theory* by Steve Awodey [1], repeating related definitions and poset examples.[1]

1 Categories

Definition 1.1 (Awodey 1.1)**.** A *category* consist of the following data

- Objects: $A, B, C, \ldots$
- Arrows: $f, g, h, \ldots$
- For each arrow f, there are given objects
$$\mathrm{dom}(f), \qquad \mathrm{cod}(f)$$
called the *domain* and *codomain* of f. We write
$$f : A \to B$$
to indicate that $A = \mathrm{dom}(f)$ and $B = \mathrm{cod}(f)$.
- Given arrows $f : A \to B$ and $g : B \to C$, that is, with
$$\mathrm{cod}(f) = \mathrm{dom}(g)$$
there is given an arrow
$$g \circ f : A \to C$$
called the *composite* of f and g.
- For each object A, there is given an arrow
$$1_A : A \to A$$
called the *identity arrow* of A.
- Associativity:
$$h \circ (g \circ f) = (h \circ g) \circ f$$
for all $f : A \to B, g : B \to C, h : C \to D$.
- Unit:
$$f \circ 1_A = f = 1_B \circ f$$
for all $f : A \to B$.

[1] If you want to learn category theory, getting a book is a small investment. Your public or university library probably have a copy.

We'll see how `Category` type-class is related later, in Section 5.

A partially ordered set or *poset* is a set A equipped with a binary relation $a \leq_A b$ such that the following conditions hold for all $a, b, c \in A$:

- reflexivity: $a \leq_A a$,
- transitivity: if $a \leq_A b$ and $b \leq_A c$, then $a \leq_A c$,
- antisymmetry: if $a \leq_A b$ and $b \leq_A a$, then $a = b$.

In HASKELL, we can define a type-class for *decidable* partial orders. Decidable means, that we can *decide* whether the relation holds:

```
class Eq a ⇒ PartialOrd a where
  leq :: a → a → Bool
```

We'll see examples of posets and instances of `PartialOrd` in a moment.

Any partial order P can be regarded as a category by taking the objects to be the elements of P and taking an unique arrow,

$$a \to b \quad \text{if and only if} \quad a \leq b.$$

The reflexive and transitive conditions on $\leq$ ensure that this is indeed a category.

Finite poset-categories are easy to visualise. We'll draw a *graph* where vertices are objects, and edges are arrows between objects. To make pictures clearer, we can omit the implied composite arrows. We'll consider a subcategory of partial orders, namely *lattices*, i.e. partial orders with all meets and joins. We'll explain what that means later.

2 Domain-class

Before continuing to examples of poset-categories, we'll need to code up how to display them. We'll need a helper typeclass, `Domain`. The name will become clear later. All partial orders we'll work with, will be instances of the `Domain` typeclass.

We also require that types are `Ord`. The `Ord` instance doesn't need to be consistent with `PartialOrd`[2] (and often cannot be). We'll use it to put elements into a `Map` and `Set`, i.e. it's an exposed "implementation detail". All finite sets can be totally ordered, so it's not a problem to require.

As we work with finite sets, we can have a list of all elements, `elements`. We also require that the list is in a topological ordering, which is (hopefully) a small optimisation.

`ltPairs` method is used for visualisation. It's a list of pairs (x, y) such that $x \leq y$, but $x \neq y$, yet not all such pairs: transitive pairs can be removed.

```
data V2 a = V2 !a !a

class (PartialOrd a, Ord a) ⇒ Domain a where
  elements :: [a]
  ltPairs  :: [V2 a]
```

`ltPairs` has a default implementation. First we construct an adjacency matrix `am`, using `elements`. Here we use the fact that `elements` is topologically ordered, so we do a bit less work. Then we construct a DAG, compute its reduction, and return an adjacency list. Finally that list is flattened into a list of `V2` pairs. `V2 a` is a homogenous pair of `a`.

```
ltPairs
    = either (const []) (concatMap mk)
```

[2] We use `lattices` package for lattice related functionality, and `topograph` to operate on directed acyclic graphs.

```
    $ runG am $ λg → adjacencyList (reduction g)
  where
    am :: Map a (Set a)
    am = Map.fromList $ go elements where
      go []     = []
      go (x:xs) = (x, Set.fromList [x' | x' ← xs, leq x x']) : go xs
    mk :: (a, [a]) → [V2 a]
    mk (x, xs) = map (V2 x) xs
```

The actual display code is in Appendix B. There we generate a *Graphviz* graph, and render it to *PNG*-image. The `Display` type class specifies how to render elements, we won't abuse `Show`.

```
class Display a where
  display :: a → String
```

3 Simple lattices

In this section, we'll visualise few simple lattices: two-, three- and four-element lattices.

3.1 Bool

One of the simplest lattices, is the[3] two element lattice. It's also known as the category **2**. It's handy to use `Bool` as it is two-element set. `Domain Bool` instance is simple to define. We sort $[minBound..maxBound]$ list, in case if `Enum` and `Bounded` instances don't agree with `PartialOrd`.

```
instance Domain Bool where
  elements = insertionSort leq [minBound..maxBound]
```

The `Display` instance is brevety, `"T"` and `"F"` for `True` and `False`.

```
instance Display Bool where
  display False = "F"
  display True  = "T"
```

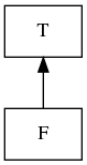


Figure 1: `Bool` lattice

With these two instances in place, we can visualise `Bool` lattice, on Figure 1.

```
outputBool :: IO ()
outputBool = outputGraph (Proxy :: Proxy Bool) "lattice-bool.png"
```

[3] It's actually *the* two element lattice. For example a set $\{a, b\}$ with a discrete partial order $a \leq a, b \leq b$ is not a lattice, as there are no $a \wedge b$ element, such that $a \wedge b \leq a$ and $a \wedge b \leq a$.

3.2 Zero-Half-One

Next lattice is so called *zero-half-one* lattice. It's the three element $(0, \frac{1}{2}, 1)$ totally ordered lattice.

$$0 \leq \frac{1}{2} \leq 1$$

It's the category **3**. The `Domain` and `Display` instances are defined similarly as in the `Bool` case.

```
type ZHO = ZeroHalfOne
instance Domain ZeroHalfOne where
  elements = insertionSort leq [minBound..maxBound]
instance Display ZeroHalfOne where
  display Zero = "0"
  display Half = "H"
  display One  = "1"
```

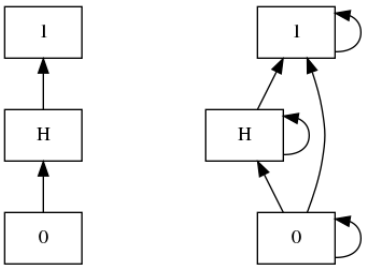


Figure 2: `ZHO` lattice, reduced and with all arrows explicitly drawn

The `ZHO` lattice is on Figure 2. Reduced version is a prettier, as redundant information is dropped.

```
outputZHO :: IO ()
outputZHO = outputGraph (Proxy :: Proxy ZHO) "lattice-zho.png"
```

3.3 M2

M_2 is one more "primitive" lattice. It is a four element lattice, $0, a, b, 1$, and the first one which partial order isn't total. Another four-element lattice is the totally ordered one.

$$0 \leq a \leq 1 \qquad 0 \leq b \leq 1 \qquad a, b \text{ are not related}$$

It's visualised on Figure 3, the graph has a nice diamond shape.

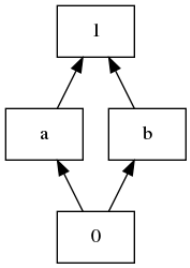


Figure 3: M_2 lattice

```
instance Domain M2 where
  elements = insertionSort leq [minBound..maxBound]
instance Display M2 where
  display M2i = "1"
  display M2a = "a"
  display M2b = "b"
  display M2o = "0"
outputM2 :: IO ()
outputM2 = outputGraph (Proxy :: Proxy M2) "lattice-m2.png"
```

4 Functor

Before proceeding, we'll answer a question: is there a category with posets as objects? Yes, it's called **Pos**! What are the arrows in this category? An arrow from a poset A to a poset B is a function

$$m : A \to B$$

that is *monotone*, in the sense that, for all $a, a' \in A$,

$$a \leq_A a' \qquad \text{implies} \qquad m(a) \leq_B m(a').$$

We need to know that $1_A : A \to A$ is monotone, and also that if $f : A \to B$ and $g : B \to C$ are monotone, then $g \circ f : A \to C$ is monotone. That holds, check!

Recall, posets are categories, so monotone functions are "mappings" between categories. A "homomorphism[4] of categories" is called a functor.

Definition 4.1 (Awodey 1.2)**.** A *functor*

$$F : \mathbf{C} \to \mathbf{D}$$

between categories **C** and **D** is a mapping of objects to objects and arrows to arrows, in such a way that

- $F(f : A \to B) = F(f) : F(A) \to F(B)$,
- $F(1_A) = 1_{F(A)}$,
- $F(g \circ f) = F(g) \circ F(f)$.

That is, F preserves domains and codomains, identity arrows, and composition. A functor $F : \mathbf{C} \to \mathbf{D}$ thus gives a sort of "picture" – perhaps distorted – of **C** in **D**.

[4] morphism preserving the structure

The HASKELL version looks quite different:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

There is a mismatch of a notation of category theory, and what can be written as code. In CT notation F acts both on objects and arrows. In HASKELL `f` acts on objects, and `fmap f` acts on arrows. Substituting above, `id` and `.` into definition of functor, will also give familiar laws

```
fmap id      = id
fmap (g . f) = fmap g . fmap f
```

However, HASKELL `Functor`-class is only for functors from a pseudo-category **Hask** to itself, where `f`, a mapping from types to types is a type-constructor, not arbitrary type family. `Functor` is a very special case of category theoretical variant.

With small posets, like `Bool` and `M2` we can visualise a monotone function, a functor. Let's consider a function

```
f :: Bool → M2
f True  = M2i
f False = M2o
```

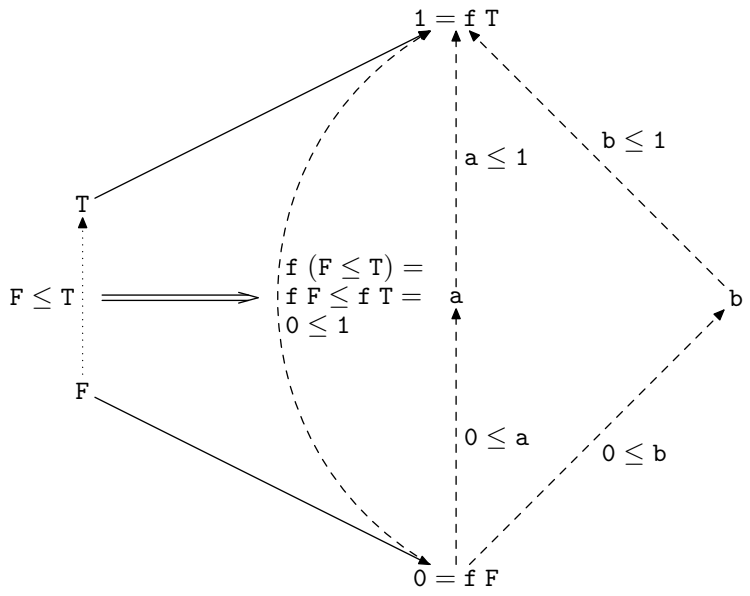


Figure 4: Graph of `f :: Bool → M2`

The graph of `f` is on Figure 4. Dotted and dashed lines are arrows in `Bool` and `M2` respectively. We can see on figure, that `f` indeed gives a picture of `Bool` in `M2`.

In HASKELL we have only written a mapping of objects, `True` and `False`. The mapping of arrows is something we need to check, to be able to claim that `f` is a functor, and therefore a monotone function. The other way around, there are mappings from `Bool` to `M2` which aren't monotone, and aren't functors.

In this section we went backwards. More principal approach would been to first consider functors between poset categories. The monotonicity requirement is implied by first functor requirement. This is a power of category theory. When you look for something, category theory tells you which properties it should have. Once you find something which satisfies the requirements, you know that it's the right one (up to an isomorphism).

5 Product

Next, we are going to see the categorical definition of a product of two objects in a category.

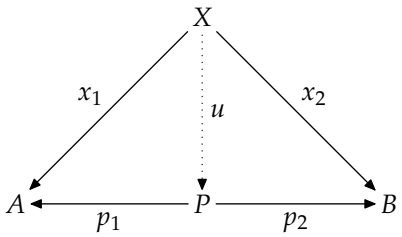
Definition 5.1 (Awodey 2.15)**.** In any category **C**, a *product diagram* for the objects A and B consists of an object P and arrows (*projections*)

$$A \xleftarrow{p_1} P \xrightarrow{p_2} B$$

satisfying the following universal mapping property:
Given any diagram of the form

$$A \xleftarrow{x_1} X \xrightarrow{x_2} B$$

there exists a unique $u : X \to P$, making the diagram



commute, that is, such that $x_1 = p_1 u$ and $x_2 = p_2 u$.

Example 5.2 (Awodey 2.5, 4th example)**.** Let P be a poset and consider a product of elements $p, q \in P$. We must have projections

$$p \times q \leq p$$
$$p \times q \leq q$$

and if for any element x,

$$x \leq p, \qquad \text{and} \qquad x \leq q$$

then we need

$$x \leq p \times q.$$

This operation $p \times q$ is usually called the *greatest lower bound* or *meet*: $p \times q = p \wedge q$. A poset with all finite meets is a *meet-semilattice*.

Example 5.3 (Awodey 2.4, 2nd example)**.** Products of "structured sets" like monoids or groups or lattices can be often constructed as products of the underlying sets with *componentwise* operations: If P and Q are meet-semilattices, for instance, $P \times Q$ can constructed by taking the underlying set of $P \times Q$ to be the set $\{\langle p, q\rangle \mid p \in P, q \in Q\}$. It can be partially ordered by

$$\langle p, q\rangle \leq \langle p', q'\rangle \qquad \text{if and only if} \qquad p \leq p' \quad \text{and} \quad q \leq q'$$

And we can define meet as

$$\langle p, q\rangle \wedge \langle p', q'\rangle = \langle p \wedge p', q \wedge q'\rangle$$

We must check that

$$\langle p, q\rangle \wedge \langle p', q'\rangle = \langle p \wedge p', q \wedge q'\rangle \leq \langle p, q\rangle$$
$$\langle p, q\rangle \wedge \langle p', q'\rangle = \langle p \wedge p', q \wedge q'\rangle \leq \langle p', q'\rangle$$

and that projection functions $p_1 : P \times Q \to P$ and $p_2 : P \times Q \to Q$ are monotone, as the pairing $\langle f, g\rangle : X \to P \times Q$, if $f : X \to P$ and $g : X \to Q$. A lot to check, but it all holds.

The category **Pos** of posets has products, as well as category **Latt** of lattices[5].

[5] In this article we examine the subcategories with *finite* underlying sets. More correctly, we should speak about **FinPos** and **FinLatt**.

At this point you should try to make it clear to yourself: **Latt** is a category of categories, and it has products. In other words products of categories with products. Abstract construction of abstract constructions.

The product in **Pos** is easy to encode in HASKELL. Pairs are products, so it's enough to write instances for `(a, b)`.

```
instance (PartialOrd a, PartialOrd b) ⇒ PartialOrd (a, b) where
  leq (a, a′) (b, b′) = leq a a′ && leq b b′
```

And that's all. The underlying theory says that this is a correct instance.

Let's define `Display` and `Domain` instances, to visualise few lattice products.

```
instance (Display a, Display b) ⇒ Display (a, b) where
  display (a, b) = "(" ++ display a ++ "," ++ display b ++ ")"
instance (Domain a, Domain b) ⇒ Domain (a, b) where
  elements
    = insertionSort leq
    $ [(a, b) | a ← elements, b ← elements]
```

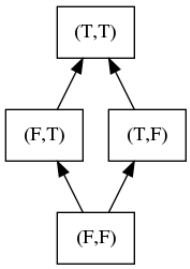


Figure 5: `Bool` × `Bool` lattice

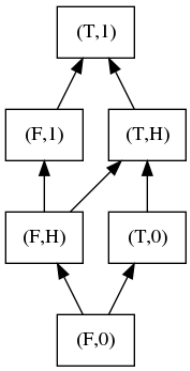


Figure 6: `Bool` × `ZHO` lattice

If we then visualise the `Bool` × `Bool` lattice, Figure 5, we'll see that it's M_2. So M_2 isn't "primitive", it's the product of `Bool` and `Bool` lattices! The `Bool` × `ZHO` lattices on Figure 6 is pretty and new, we haven't seen such lattice yet.

```
outputBoolBool :: IO ()
outputBoolBool = outputGraph (Proxy :: Proxy (Bool, Bool)) "lattice-boolbool.png"
outputBoolZHO :: IO ()
outputBoolZHO = outputGraph (Proxy :: Proxy (Bool, ZHO)) "lattice-boolzho.png"
```

6 Duals: Coproduct

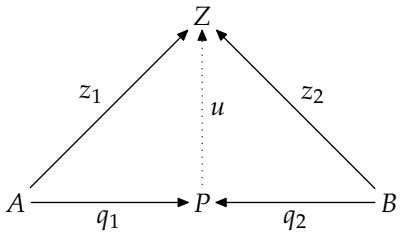
We can form a "dual statement" in the elementary language of category theory by making the following replacements:

$$\begin{array}{rcl} f \circ g & \text{for} & g \circ f \\ \text{cod} & \text{for} & \text{dom} \\ \text{dom} & \text{for} & \text{cod.} \end{array}$$

i.e. "reversing the arrows".

Let us consider the example of products and see what the dual notion must be.

Definition 6.1 (Awodey 3.3)**.** A diagram $A \xrightarrow{q_1} Q \xleftarrow{q_2} B$ is a "dual-product" of A and B if for any Z and $A \xrightarrow{z_1} Z \xleftarrow{z_2} B$ there is an unique $u : Q \to Z$ with $u \circ q_1 = z_1$ and $u \circ q_2 = z_2$ all as indicated in



Actually, these are called *coproducts*; the convention is to use the prefix "co-" to indicate the dual notion.

Example 6.2 (Awodey 3.7)**.** In a fixed poset P, what is a coproduct of two elemnents $p, q \in P$? We have

$$p \leq p + q \qquad \text{and} \qquad q \leq p + q$$

and if

$$p \leq z \qquad \text{and} \qquad q \leq z$$

then

$$p + q \leq z.$$

So $p + q = p \vee q$ is the *join*, or *least upper bound*, of p and q. A poset with all finite joins is a *join-semilattice*. A poset with all finite meets and joins is a *lattice*.

Similarly as product, we can try to define a coproduct in **Pos**. Given posets P and Q, the poset $P + Q$ has tagged elements L p and R q, and is partially order by

$$\begin{array}{lll} \text{L } p \leq \text{L } p' & \text{if and only if} & p \leq p' \\ \text{R } q \leq \text{R } q' & \text{if and only if} & q \leq q' \end{array}$$

The injections are *monotone*, as the pairing $[f, g] : P + Q \to Z$, if $f : P \to Z$ and $Q : P \to Z$ are monotone.

Let's encode coproducts in HASKELL. Coproduct or *sum* of sets is `Either`. Instances for `Either`:

```
instance (PartialOrd a, PartialOrd b) ⇒ PartialOrd (Either a b) where
  leq (Left a)  (Left a')  = leq a a'
  leq (Right b) (Right b') = leq b b'
  leq _         _          = False
instance (Display a, Display b) ⇒ Display (Either a b) where
  display (Left a)  = "L " ++ display a
```

```
  display (Right b) = "R " ++ display b
instance (Domain a, Domain b) => Domain (Either a b) where
  elements
    = insertionSort leq
    $ map Left elements ++ map Right elements
```

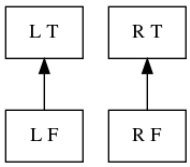


Figure 7: `Bool` + `Bool` poset

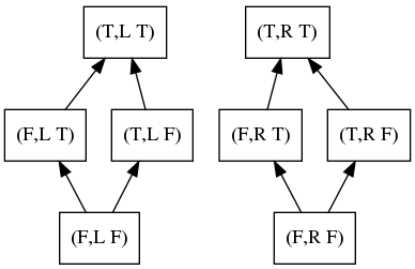


Figure 8: `Bool` × (`Bool` + `Bool`) poset

`Bool` + `Bool` partial order (Figure 7) is not a lattice, there are two disjoint parts. We can conclude that even **Pos** has coproducts, **Latt** doesn't. Sad truth, many categories don't have coproducts.

The graph of `Bool` × (`Bool` + `Bool`) on Figure 8 nicely show the distributivity of coproducts and products (the execise 8.9.3) there are two disjoint `Bool` × `Bool` graphs.

```
outputBoolOrBool ::  IO ()
outputBoolOrBool =
  outputGraph (Proxy :: Proxy (Either Bool Bool)) "lattice-boolObool.png"
outputBoolBoolOrBool :: IO ()
outputBoolBoolOrBool =
  outputGraph (Proxy :: Proxy (Bool, Either Bool Bool)) "lattice-boolboolObool.png"
```

7 Expontentials: Monotone functions

Next we are going to look at one more elementaty universal structure. This important structure is called an *exponential*, and it can be thought of as a categorical notion of a "function space".

Definition 7.1 (Awodey 6.1)**.** Let the category **C** have binary products. An *exponential* of objects B and C consists of an object

$$C^B$$

and an arrow (*evaluation*)

$$\epsilon : C^B \times B \to C$$

such that, for any object A and arrow

$$f : A \times B \to C$$

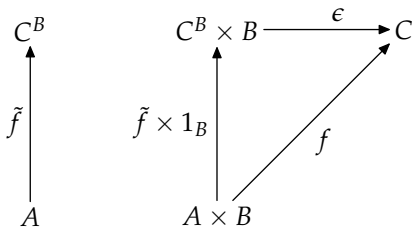
there is an a unique arrow (*transpose* of f)

$$\tilde{f} : A \to C^B$$

such that

$$\epsilon \circ (\tilde{f} \times 1_B) = f$$

all as in the diagram



Example 7.2 (Exponential in **Sets**)**.** In category **Sets** the exponential C^B is a set of function space between C and B. It's often important diffentiate between $B \to C$, which is an *arrow* in a category, and a C^B which is an *object* in that category. HASKELL "abuses" the notation by identifying $C^B = B \to C$.

Example 7.3 (Awodey 6.4)**.** The category **Pos** has exponential objects. For the exponential Q^P, we take the set of monotone functions,

$$Q^P = \{f : P \to Q \mid f \text{ monotone}\}$$

ordered *pointwise*, that is,

$$f \leq q \qquad \text{if and only if} \qquad f\,p \leq q\,p \text{ for all } p \in P.$$

The evaluation $\epsilon : Q^P \times P \to Q$ and transposition $\tilde{f} : X \to Q^P$ of a given arrow $f : X \times P \to Q$ are the usual ones of the underlying functions. Awodey shows that these are monotone.

Coding this up in HASKELL is a bit trickier than previously. We'll have a **newtype** `Monotone`, to be able to write our instances.

```
newtype Monotone a b = M { evalMonotone :: a → b }
```

As a short digression, we can define `Category Monotone` instance. In HASKELL `Category` type-class is defined for morphisms in that category, where usually in category theory text we speak about the objects: **Set**, **Pos**, **Latt**... This is one more case where category theory and HASKELL "differ in notation".

```
  -- Category of partial orders
instance Category Monotone where
  id      = M id
  M g . M f = M (g . f)
```

`Eq`, `Ord`, `PartialOrd` instances use `elements` provided by `Domain` type-class. The choice of the name should become obvious now.

```
instance (Domain a, Eq b) ⇒ Eq (Monotone a b) where
  M f ≡ M g           = all (λx → f x ≡ g x) elements
instance (Domain a, PartialOrd b) ⇒ PartialOrd (Monotone a b) where
```

```
  leq (M f) (M g)       = all (λx → leq (f x) (g x)) elements
instance (Domain a, Ord b) ⇒ Ord (Monotone a b) where
  compare (M f) (M g) = foldMap (λx → compare (f x) (g x)) elements
```

`Lattice` instances are defined pointwise. Note, that while `Monotone` preserve joins and meets, also joins and meets of a arbitrary subset, but it doesn't preserve $\top$ and $\bot$ elements. Yet, all *finite* lattices do have $\top$ and $\bot$ elements.

```
instance Lattice b ⇒ Lattice (Monotone a b) where
  M f ∧ M g = M (λx → f x ∧ g x)
  M f ∨ M g = M (λx → f x ∨ g x)
instance BoundedJoinSemiLattice b ⇒ BoundedJoinSemiLattice (Monotone a b) where
  bottom  = M (const bottom)
instance BoundedMeetSemiLattice b ⇒ BoundedMeetSemiLattice (Monotone a b) where
  top     = M (const top)
```

`Display` instance simply lists the codomain elements:

```
instance (Domain a, Display b) ⇒ Display (Monotone a b) where
  display (M f) = concatMap (display . f) elements
```

The tricky part is the `Domain` instance. First we define a `isMonotone` helper, which checks whether `f :: a → b` is monotone, if it is, it returns `Just (M f)`, otherwise `Nothing`. As we assume that `a` and `b` have lawful `PartialOrd` instances, it's enough to test for `ltPairs` only.

The `elements` are then generated by enumerating all possible functions (between "monotone" `elements` of `a` and `b`) and filtering out not monotone ones. `elements` of `Monotone` doesn't contain all possible functions `a → b`.

```
isMonotone :: (Domain a, PartialOrd b) ⇒ (a → b) → Maybe (Monotone a b)
isMonotone f
  | all (λ(V2 x y) → leq (f x) (f y)) ltPairs = Just (M f)
  | otherwise                                = Nothing
instance (Domain a, Domain b) ⇒ Domain (Monotone a b) where
  elements = insertionSort leq $ case elements :: [b] of
    [] → []
    bs@(b : _) → mapMaybe isMonotone
      [ λa → Map.findWithDefault b a pref
      | pref ← expo elements bs
      ]
expo :: Ord a ⇒ [a] → [b] → [Map a b]
expo []       _  = [Map.empty]
expo (a : as) bs = bs >>= λb → map (Map.insert a b) (expo as bs)
```

Now we can see few examples. `Bool → Bool` instance is shown on Figure 9. That's `ZHO`! The `ZHO → Bool` instance on the same figure is the totally ordered set of size four.

```
outputBoolToBool :: IO ()
outputBoolToBool =
  outputGraph (Proxy :: Proxy (Monotone Bool Bool)) "lattice-bool2bool.png"
outputZHOToBool :: IO ()
outputZHOToBool =
  outputGraph (Proxy :: Proxy (Monotone ZHO Bool)) "lattice-zho2bool.png"
```

The `ZHO → ZHO` and `Bool → ZHO` lattices are getting a little of complexity, see Figure 10. These are simple examples highlighting that exponential objects from total ordered sets to another may be partial (as well as products, `Bool × Bool` is not a toset).

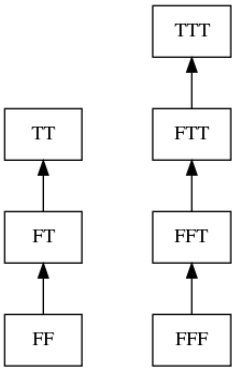


Figure 9: `Bool` → `Bool` and `ZHO` → `Bool` lattices

```
outputZHOToZHO :: IO ()
outputZHOToZHO =
  outputGraph (Proxy :: Proxy (Monotone ZHO ZHO)) "lattice-zho2zho.png"
outputBoolToZHO :: IO ()
outputBoolToZHO =
  outputGraph (Proxy :: Proxy (Monotone Bool ZHO)) "lattice-bool2zho.png"
```

Exponential lattices with `M2` are pretty. `ZHO` → `M2` (Figure 11) has nice planar graph. The `M2` → `ZHO` (Figure 12) has few overlapping edges. `M2` → `M2` (Figure 13) starts to exercise *Graphviz* layout algorithm. Yet the final stress test is (`ZHO` → `ZHO`) → `ZHO`, or $\texttt{ZHO}^{\texttt{ZHO}^{\texttt{ZHO}}}$ is on Figure 14. A beautiful monster.

```
outputM2ToM2 :: IO ()
outputM2ToM2 =
  outputGraph (Proxy :: Proxy (Monotone M2 M2)) "lattice-m22m2.png"
outputZHOToM2 :: IO ()
outputZHOToM2 =
  outputGraph (Proxy :: Proxy (Monotone ZHO M2)) "lattice-zho2m2.png"
outputM2ToZHO :: IO ()
outputM2ToZHO =
  outputGraph (Proxy :: Proxy (Monotone M2 ZHO)) "lattice-m22zho.png"
outputBig :: IO ()
outputBig =
  outputGraph (Proxy :: Proxy (Monotone (Monotone ZHO ZHO) ZHO))
    "lattice-big.png"
```

8 Conclusion

That was fun. Pretty pictures. It was nice to learn that **Pos** and **Latt** are a *cartesian closed*.

Definition 8.1 (Awodey 6.2)**.** A category is called *cartesian closed*, if it has all finite products and exponentials.

In 6.6 section Awodey explains the correspondence between CCCs and λ-calculus:

$$\text{CCC} \sim \lambda\text{-calculus}$$

That means that we can write simply typed λ-calculus (STLC) programs, and interpret them in CCC of our liking. Conal Elliott describes how we can compile STLC to CCCs and gives

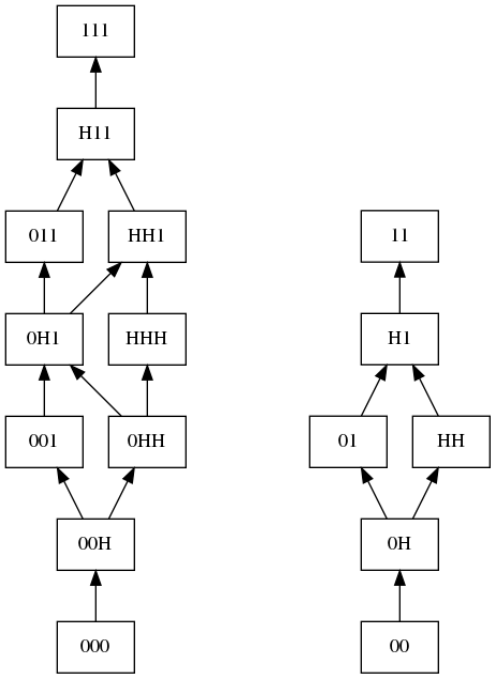


Figure 10: `ZHO` $\to$ `ZHO` and `Bool` $\to$ `ZHO` lattices

more examples of CCCs [2], there are practical applications! It's an interesting question, what would a programming with lattices be useful for?

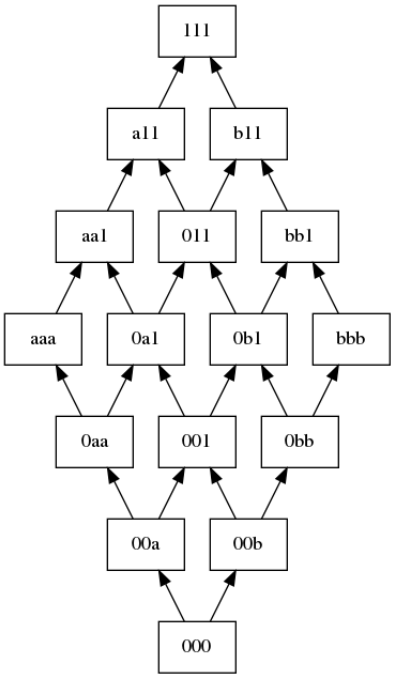


Figure 11: `ZH0` → `M2` lattice

References

[1] AWODEY, S. *Category Theory*, 2nd ed. Oxford University Press, Inc., New York, NY, USA, 2010.

[2] ELLIOTT, C. Compiling to categories. *Proc. ACM Program. Lang. 1*, ICFP (Aug. 2017), 27:1–27:27.

A Insertion sort

Insertion sort topographically orders the list, given a partial order decision procedure.

```
insertionSort :: (a → a → Bool) → [a] → [a]
insertionSort le = go where
  go []     = []
  go (x:xs) = insert x (go xs)
  insert x []                   = [x]
  insert x (y:ys) | le x y      = x:y:ys
                  | otherwise = y:insert x ys
```

B Display

Functions to render pretty pictures.

```
displayDomain :: forall a. (Domain a,Display a) ⇒ Proxy a → String
displayDomain _ = unlines $
  [ "digraph G {"
```

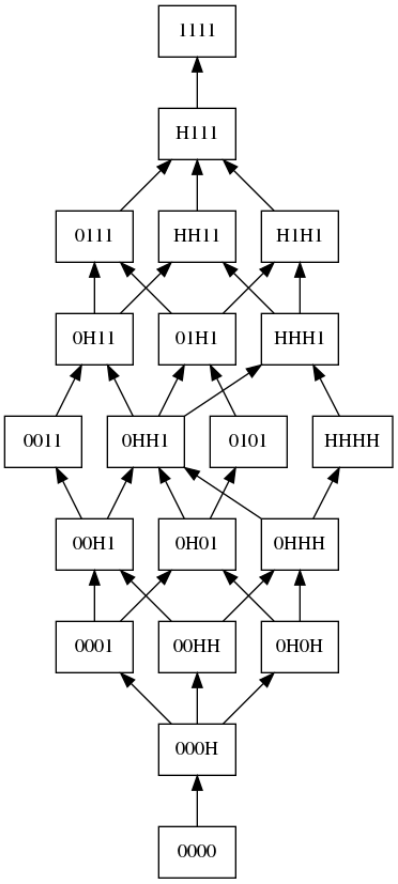


Figure 12: M2 → ZHO lattice

```
  , "rankdir=BT;"
  , "node [shape=box];"
  ] ++
  [ show (display x) ++ " -> " ++ show (display y) ++ ";"
  | V2 x y ← ltPairs :: [V2 a]
  ] ++
  [ "}"
  ]
outputGraph :: (Domain a, Display a) ⇒ Proxy a → FilePath → IO ()
outputGraph p fp = void $ readProcess "dot"
  ["-Tpng","-o" ++ fp]
  (displayDomain p)
```

C Another ZHO

This is a wrapper over ZHO with complete ltPairs.

```
newtype ZHO2 = ZHO2 ZHO deriving (Eq, Ord, PartialOrd, Display)
instance Domain ZHO2 where
  elements = map ZHO2 elements
  ltPairs  = [V2 x y | x ← elements, y ← elements, leq x y]
outputZHO2 :: IO ()
```

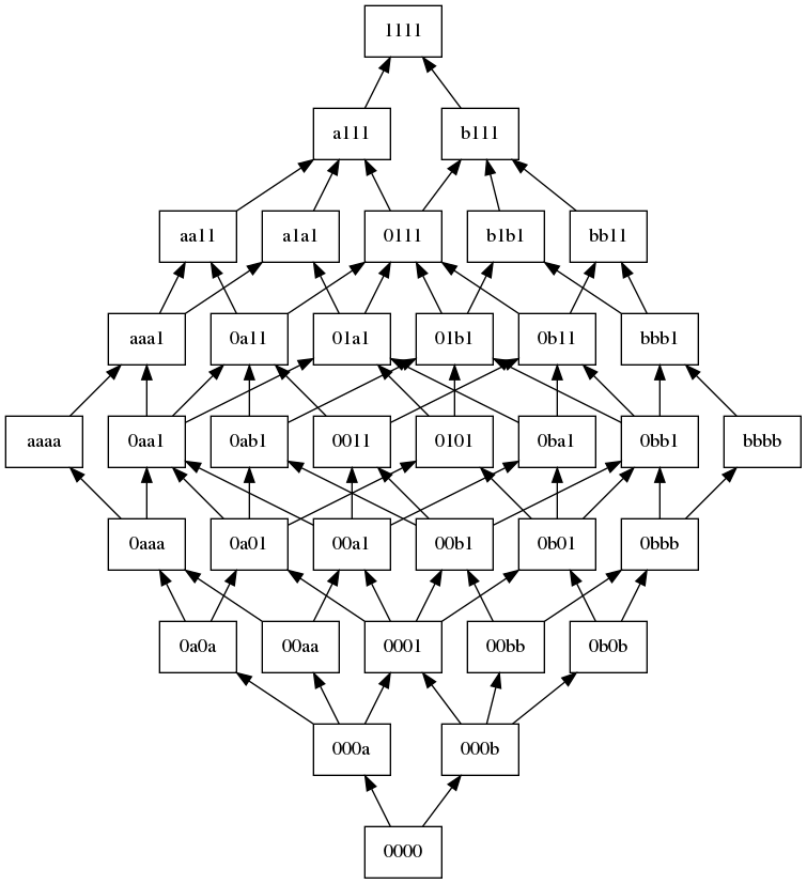


Figure 13: `M2` → `M2` lattice

```
outputZHO2 = outputGraph (Proxy :: Proxy ZHO2) "lattice-zho2.png"
```

D Main

The `main` function outputs all the images we defined above.

```
main :: IO ()
main = do
  outputBool
  outputZHO
  outputZHO2
  outputM2
  outputBoolBool
  outputBoolZHO
  outputBoolOrBool
  outputBoolBoolOrBool
  outputBoolToBool
  outputZHOToBool
  outputBoolToZHO
  outputZHOToZHO
  outputM2ToM2
  outputZHOToM2
```

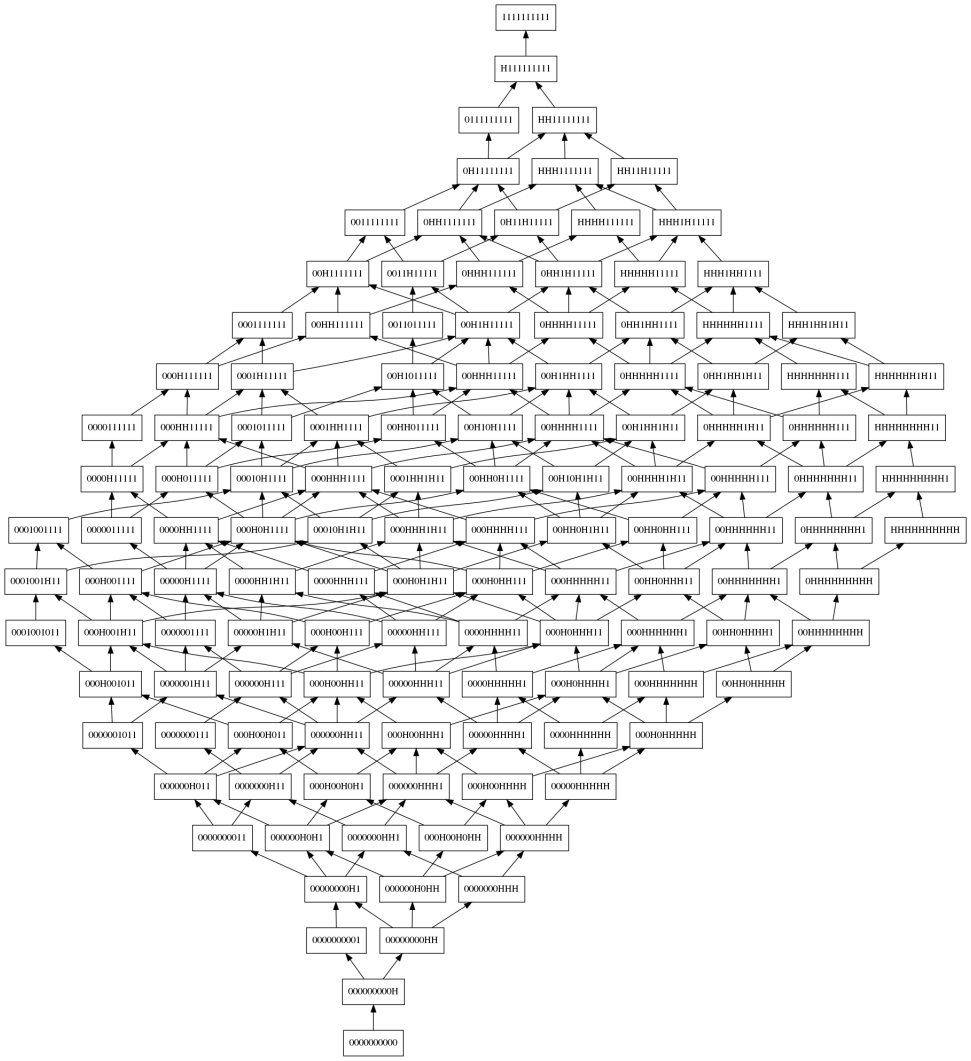


Figure 14: $(\texttt{ZHO} \to \texttt{ZHO}) \to \texttt{ZHO}$ lattice

```
outputM2ToZHO
outputBig
```