Functional Thinking, Part 2

Christopher R. Genovese

Department of Statistics & Data Science

01 Oct 2024
Session #11

Plan

Review

Plan

Review

More Examples

Plan

Announcements

- **Reading**:
  - Thursday:
    - What is Category Theory?
    - Definitions and Examples
    - What is a Functor? Part 1
    - What is a Functor? Part 2
    - Fibonacci Functor
    - Natural Transformations
- **Homework**: `monoidal-folds` or `zippers` due next Tuesday. (As I said earlier, talk to me if you find these overwhelming; that window is closing.)

Plan

Review

More Examples

Design Activity

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

```
instance Functor Maybe where
    map : (a -> b) -> Maybe a -> Maybe b
    map f None = None
    map f (Some x) = Some (f x)
```

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

```
instance Functor List where
    map : (a -> b) -> List a -> List b
    map f Nil = Nil
    map f (Cons x rest) = Cons (f x) (map f rest)
```

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

```
instance Functor ((->) r) where
    map : (a -> b) -> (r -> a) -> (r -> b)
    map = (.)    -- compose!
```

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

```
type BinaryTree a = Node (BinaryTree a) a (BinaryTree a)

instance Functor BinaryTree where
    map : (a -> b) -> BinaryTree a -> BinaryTree b
    map f (Node left x right) = Node (map f left) (f x) (map f right)
```

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

```
type Tree a = Node { value : a
                   , children : List (Tree a)
                   }

instance Functor Tree where
    map : (a -> b) -> Tree a -> Tree b

    map f tree = Node { value = f(tree.value)
                      , children = map f (tree.children)
                      }
```

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

What does this do?

```
map (map (__ + 1)) [Some 10, None, Some -1, None, None, Some 99]
```

Review and Laws

```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b
```

A Functor is a type constructor with an associated function that ***lifts*** a transformation of values to a transformation of "containers." (We could call it `lift`!)

Functors satisfy two important **laws**:

1. `map id == id`
2. `(map g) . (map f) == map (g . f)`

The first law says that lifting the identity gives you the identity. The second law says that composing the lift of g and the lift of f is the same as lifting $g \circ f$. This is equivalent to `compose (map f) (map g) == map (compose f g)`.

What does this do?

```
tree = Node { value=10
            , children=[ Node { value=20, children=[] }
                       , Node { value=30, children=[ Node {value=40, children=[]} ]
                       , Node { value=50, children=[ Node {value=60, children=[]}
                                                   , Node {value=70, children=[]}
                                                   ]
                              }
                       ]
            }
map (__ * 2) tree
```

A Quick Python Implementation

See `lec11.py` in the `documents` repository.

Discussion: A perspective on our computations, implementation is nice too

A few general points:

1. Mixins
2. Partial application
3. Recursion
4. Inductive definitions
5. Generic functions
6. Common interface
7. Lifting

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c    -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                               -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                                -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

```
instance Applicative List where
    pure : a -> List a
    pure x =

    map2 : (a -> b -> c) -> List a -> List b -> List c
    map2 f (Cons x xs) (Cons y ys) =
```

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                                 -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

```
instance Applicative List where
    pure : a -> List a
    pure x = [x]

    map2 : (a -> b -> c) -> List a -> List b -> List c
    map2 f xs ys = [f x y for x <- xs, y <- ys]

map2 (+) [1, 2, 3] [10, 11] == [11, 12, 12, 13, 13, 14]
```

Could we define a different instance?

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                                -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

```
newtype ZipList a = ZipList (List a)   -- newtype is alternative wrapper for a type
instance Applicative ZipList where
    pure : a -> ZipList a
    pure x = ZipList [x]

    map2 : (a -> b -> c) -> ZipList a -> ZipList b -> ZipList c
    map2 _ _ Nil = ZipList Nil
    map2 _ Nil _ = ZipList Nil
    map2 f (Cons x xs) (Cons y ys) = Cons (f x y) (map2 f xs ys)

map2 (+) (ZipList [1, 2, 3]) (ZipList [10, 11]) == ZipList [10, 13]
```

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                                 -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

How do we derive `unit` and `pair`?

```
unit =
pair fa fb =
```

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                                -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

`unit` is an identity (up to isomorphism) and `pair` is associative (up to isomorphism)

```
unit = pure ()
pair fa fb = map2 (,) fa fb
```

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                                -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

Can you go the other way, defining `pure` and `map2` from `unit` and `pair` (and `map`)?

```
pure a =
map2 g fa fb =
```

Another Trait

Generalizing Functor

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- aka lift2

    -- equivalent derived features, monoidal operations
    unit : f Unit                                  -- Unit equiv ()
    pair : f a -> f b -> f (a, b)
```

This satisfies several laws as well (see `unit` and `pair` below).

Remember `Applicative` is a `Functor`. These are equivalent up to *isomorphism* only.

```
pure a = map (const a) unit
map2 g fa fb = map (uncurry g) (pair fa fb)
```

Here, `const a` is the constant function and `uncurry : (a -> b -> c) -> (a, b) -> c` reconfigures a function's arguments.

Currying and Partial Application

```
curry : ((a, b) -> c) -> a -> b -> c
uncurry : (a -> b -> c) -> (a, b) -> c

addPair = uncurry (+)
1 + 2 == (+) 1 2 == 3
addPair (1, 2) == 3

swap : (a, b) -> (b, a)
swap (x, y) = (y, x)

swap2 = curry swap
swap (1, 2) == (2, 1)
swap2 1 2 == (2, 1)
```

Plan

Review

More Examples

Design Activity

Pruning a (Game) Tree

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) =
prune n (Node v cs) =
```

Pruning a (Game) Tree

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) = Node v Nil
prune n (Node v cs) = Node v (map (prune (n - 1)) cs)
```

Pruning a (Game) Tree

We need to prune the tree.

```
prune : Natural -> Tree a -> Tree a
prune 0 (Node v cs) = Node v Nil
prune n (Node v cs) = Node v (map (prune (n - 1)) cs)
```

Now we have a more realistic evaluation

```
evaluate : Position -> Number
evaluate pos = gameTree pos |> prune 4 |> map static |> maximize
```

which gives a lookahead of 4 moves.

We are using *lazy evaluation* here. This evaluates positions only as *demanded* by `maximize` so the whole tree is never in memory.

We can optimize this further.

Newton-Raphson Square Roots

Consider the recurrence relation $a_{n+1} = (a_n + x/a_n)/2$ for $x > 0$ and $a_0 = x$. As n increases, $a_n \to \sqrt{x}$.

We might compute with this typically with

```
def sqrt(x, tolerance=1e-7):
    u = x
    v = x + 2.0 * tolerance
    while abs(u - v) > tolerance:
        v = u
        u = 0.5 * (u + x / u)
    return u
```

We will put this in a more modular style with ingredients that can be reused for other problems.

Newton-Raphson Square Roots (cont'd)

```
next : Real -> Real -> Real
next x a = (a + x / a) / 2

iterate f x = Cons x (iterate f (f x))     -- a lazy sequence
iterate (next x) init   -- lazy sequence of sqrt approximations

approx x = iterate (next x) x

within : Real -> NonEmptyList Real -> Real
within tol (Cons a0 (Cons a1 rest))
   | (abs(a0 - a1) <= tol) = a1
   | otherwise             = within tol (Cons a1 rest)
within tol (Cons a0 Nil)   = a0

relative : Real -> NonEmptyList Real -> Real
relative tol (Cons a0 (Cons a1 rest))
   | (abs(a0/a1 - 1) <= tol) = a1
   | otherwise               = relative tol (Cons a1 rest)
relative tol (Cons a0 Nil)   = a0

a_sqrt x tol = within tol (approx x)
r_sqrt x tol = relative tol (approx x)
```

Let's see this in action. These same primitives apply *as is* in other problems e.g., differentiation, integration, simulation.

Numerical Derivatives

```
divDiff f x h = (f (x + h) - f x) / h
halve x = x / 2
derivative f x h0 = map (divDiff f x) (iterate halve h0)

within tol (derivative f x h0)

-- sharpen error terms that look like a + b h^n
sharpen n (Cons a (Cons b rest))
  = Cons ((b * (2**n) - a) / (2**n - 1)) (sharpen n (Cons b rest))

order (Cons a (Cons b (Cons c rest)))
  = round (log2 ((a - c) / (b - c) - 1))

improve seq = sharpen (order seq) seq

within tol (improve (derivative f x h0))

within tol (improve (improve (derivative f x h0)))
```

- ... can improve even further.

Plan

A Quick Tour of Zippers

See **zippers** in problem bank.

Design Activity: Dominoes

See **dominoes** activity in problem-bank.

What are the layers of responsibility here?

What are the entities we need to manage/track?

What are the data? the actions? the calculations?

Sketch out the structure of the task.

For consideration later:

```
type Algebra f a = f a -> a
type CoAlgebra f a = a -> f a

-- Lazily build and reduce the tree
-- >>> is chained composition
hylo : Functor f => Algebra f a -> CoAlgebra f b -> b -> a
hylo alg coalg = coalg >>> map (hylo alg coalg) >>> alg
```

THE END