Functional Thinking, Part 2

Christopher R. Genovese

Department of Statistics & Data Science

Tue 21 Oct 2025
Session #15

Plan

Recap: Functors

Plan

Recap: Functors

Intro to Effects and Specialized Functors

Plan

Recap: Functors

Intro to Effects and Specialized Functors

An Example From Hughes

Plan

Recap: Functors

Intro to Effects and Specialized Functors

An Example From Hughes

A Quick Tour of Zippers

Plan

Announcements

- Assignment timing
- **Reading**:
  - Zippers
    - You could have invented Zippers
    - Huet Zippers
    - Zippers as Derivatives (optional)
  - Hughes, Why Functional Programming Matters
  - Commentary on Hughes with Python examples
  - Starting sequence on categories
    - What is Category Theory?
    - Definitions and Examples
    - What is a Functor? Part 1
    - What is a Functor? Part 2
    - Fibonacci Functor
    - Natural Transformations
  - Backus, Can Programming be Liberated from the Von Neumann Architecture
- **Homework**:
  - **classification-tree-basic** assignment due Thu 23 Oct.
  - Push outstanding mini-assignments when complete

Plan

Recap: Functors

Intro to Effects and Specialized Functors

An Example From Hughes

A Quick Tour of Zippers

Appendix: Another Hughes Example

Recap: Functors

```
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`!)

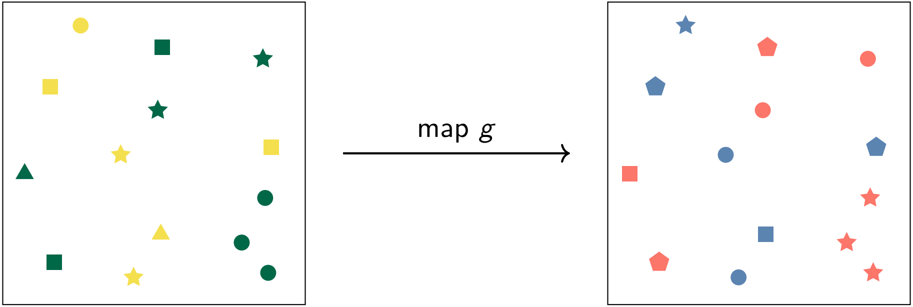
Functors satisfy two important **laws**:

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)`.

Functors as a Computational Context

We can view functors as a ***computational context*** where we can transform the "results" inside it while preserving the context's "shape."




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

laws Functor where
    map id == id
    map g . map h == map (g . h)
```

Functors (cont'd): Basic Examples

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

Some fundamental examples:

Functors (cont'd): Basic Examples

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

Some fundamental examples:

```
data Maybe : Type -> Type where
  None : Maybe a
  Some : a -> Maybe a

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

Functors (cont'd): Basic Examples

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

Some fundamental examples:

```
data List (a : Type) where
  [] : List a
  (::) : a -> List a -> List a

implements Functor List where
  map : (a -> b) -> List a -> List b
  map f [] = []
  map f (x :: rest) = (f x) :: (map f rest)
```

Functors (cont'd): Basic Examples

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

Some fundamental examples:

```
implements Functor ((->) r) where
  map : (a -> b) -> (r -> a) -> (r -> b)

  map f g = f . g          -- (.) is composition

  -- or equivalently
  map f g x = f (g x)
```

Functors (cont'd): Basic Examples

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

Some fundamental examples:

```
data BinaryTree a = Node (BinaryTree a) a (BinaryTree a)

implements 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)
```

Functors (cont'd): Basic Examples

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

Some fundamental examples:

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


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

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

Brief Demo

FP-Concepts (nearing completion, but see documents)

Plan

Effects

Effects refer to ordinary computations augmented with some extra capabilities. We represent effects with types `f : Type -> Type` that *lift* **calculations to actions**.

$$f\ a$$

the effect context

results of the computation

Effects

Effects refer to ordinary computations augmented with some extra capabilities. We represent effects with types `f : Type -> Type` that *lift* **calculations to actions**.

f a

the effect context

results of the computation

There are many different, commonly-used effects. These can all be expressed as Functors, but in practice we need more power to use them than `map` alone can give.

Effects

`List a`
(non-determinism)

`Pair c a`
(conjunction)

`Maybe a`
(partiality)

`Either e a`
(disjunction)

`IO a`
(input/output)

`Reader r a`
(environment)

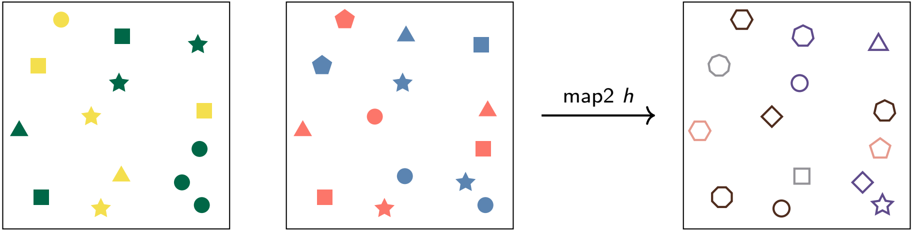
`...`
many more

`Random g a`
(randomness)

`Writer w a`
(logging)

`State s a`
(updating state)

Applicative Functors



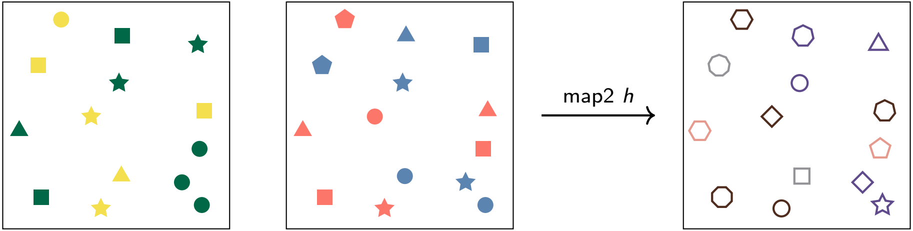

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                  -- Unit equiv ()
    combine : f a -> f b -> f (a, b)
```

Can derive pairs pure and map2, pure and ap, and unit and combine from each other.

```
laws Applicative where
    combine unit a  ~= a ~= combine a unit
    combine a (combine b c) ~= combine (combine a b) c
    combine (map g fa) (map h fb) == bimap g h (combine fa fb)
```

The laws can also be stated in terms of pure and ap or pure and map2.

Applicative Functors

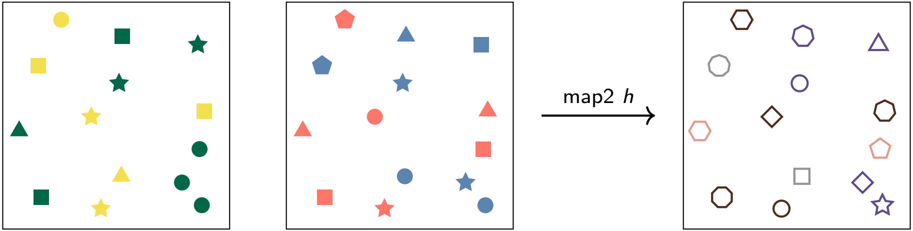



```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c      -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                  -- Unit equiv ()
    combine : f a -> f b -> f (a, b)

implements Applicative Maybe where
    pure : a -> Maybe a
    pure x =

    map2 : (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
    map2 f xs ys =
```

Applicative Functors



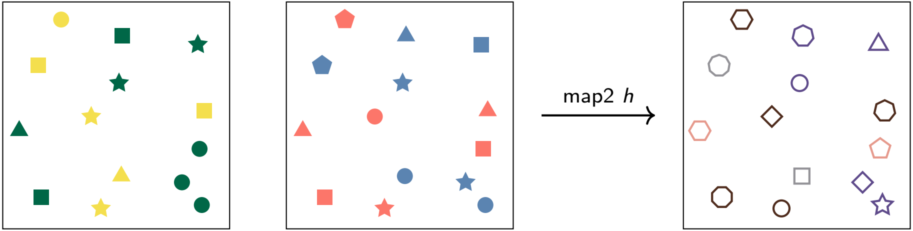

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                 -- Unit equiv ()
    combine : f a -> f b -> f (a, b)

implements Applicative Maybe where
    pure : a -> Maybe a
    pure = Some

    map2 : (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
    map2 f (Some x) (Some y) = Some (f x y)
    map2 f _ _ = Nothing

map2 (+) (Some 10) (Some 90) == Some 100
```

Applicative Functors

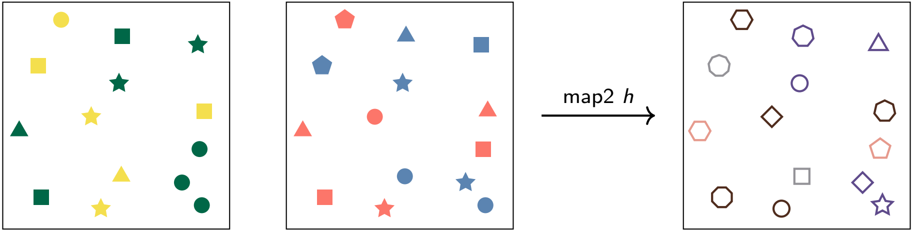



```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                  -- Unit equiv ()
    combine : f a -> f b -> f (a, b)

implements Applicative List where
    pure : a -> List a
    pure x =

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

Applicative Functors



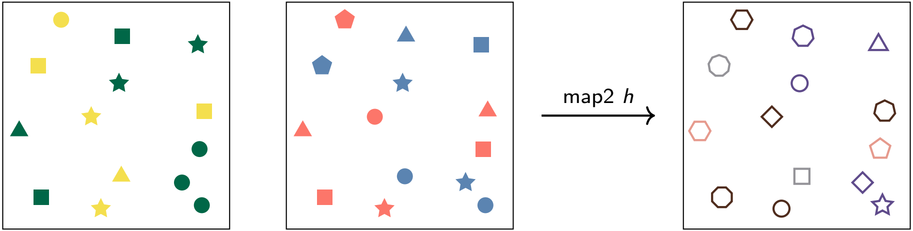

```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                 -- Unit equiv ()
    combine : f a -> f b -> f (a, b)

implements 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]
```

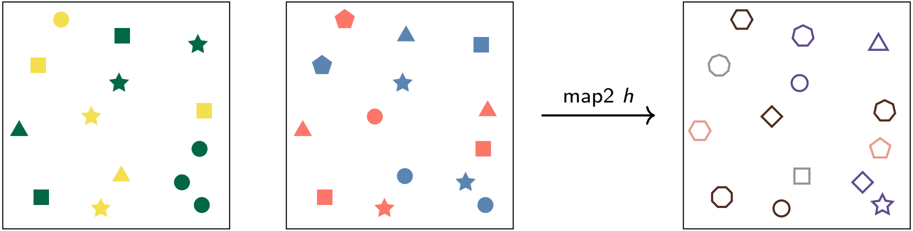
Applicative Functors




```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                 -- Unit equiv ()
    combine : f a -> f b -> f (a, b)

newtype ZipList a = ZipList (List a)    -- newtype is alternative wrapper for a type
implements 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 (x :: xs) (y :: ys) = (f x y) :: (map2 f xs ys)
map2 (+) (ZipList [1, 2, 3]) (ZipList [10, 11]) == ZipList [10, 13]
```

Applicative Functors

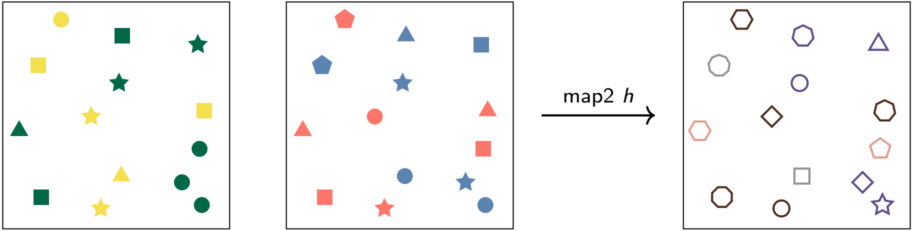



```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                   -- Unit equiv ()
    combine : f a -> f b -> f (a, b)
```

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

```
unit =
combine fa fb =
```

Applicative Functors

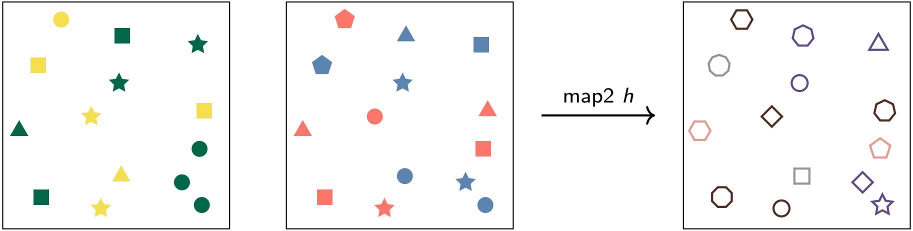



```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                  -- Unit equiv ()
    combine : f a -> f b -> f (a, b)
```

unit is an identity (up to isomorphism) and combine is associative (up to isomorphism)

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

Applicative Functors

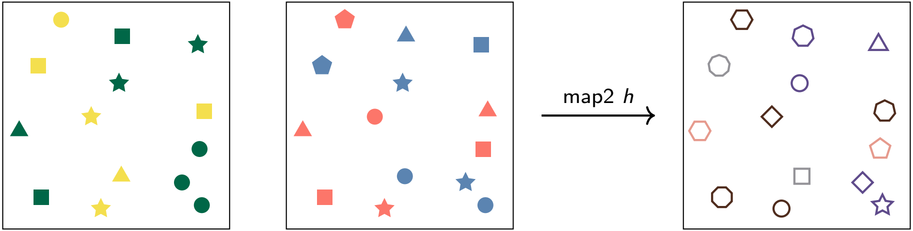



```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                  -- Unit equiv ()
    combine : f a -> f b -> f (a, b)
```

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

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

Applicative Functors




```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b
    unit : f Unit                                 -- Unit equiv ()
    combine : f a -> f b -> f (a, b)
```

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) (combine fa fb)
```

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

Brief Demo

FP-Concepts (nearing completion, but see documents)

Folds, Traversals, and Filters

Contexts that can be reduced to a summary value one piece at a time are ***foldable***:

```
trait Foldable (f : Type -> Type) where
    foldM : Monoid m => (a -> m) -> f a -> m
    fold  : (a -> b -> a) -> a -> f b -> a
```

Contexts in which elements can be removed are ***filterable***:

```
trait Functor f => Filterable (f : Type -> Type) where
    mapMaybe : (a -> Maybe a) -> f a -> f b
```

Contexts that can be transformed to one of the same *shape* by executing an *effectful* function one element at a time are ***traversable***:

```
trait (Functor t, Foldable t) => Traversable (t : Type -> Type) where
    traverse : Applicative f => (a -> f b) -> t a -> f (t b)
    sequence : Applicative f => t (f a) -> f (t a)
```

Brief Demo

FP-Concepts (nearing completion, but see documents)

A Common Pattern: Composing *Programs*

$$f(x) = 3x^2 + 4 = (\blacksquare + 4) \circ (3\blacksquare) \circ (\blacksquare^2)$$

Function composition is associative with a unit (a monoid).

We can think of programs as being composed in a similar way.

```python
def a(x):
    print('Hello, ', end='')
    return x + 1

def b(x):
    print('world!')
    return x + 2

def main():
    c = a(1) + b(2)
    print( f'c = {c}')
    return 0
```

```python
def alt_main():
    c = b(2) + a(1)
    print( f'c = {c}')
    return 0
```

Are `main` and `alt_main` the same program?

Composing Programs (cont'd)

In Python/R, + is adding numbers, and addition should be *commutative*.

But we are not adding numbers, we are adding *programs*!

```
 a  : Int -> IO Int
 b  : Int -> IO Int
(+) : Int -> Int -> Int
```

We need a distinction between calculations and actions/effects.

Composing Programs (cont'd)

In general, we want to *compose* programs, but we cannot just do it (`Int -> IO Int` does not compose with `Int -> IO Int`).

```
   (.)     : (b ->    c) -> (a ->    b) -> (a ->    c)
 semicolon : (b -> IO c) -> (a -> IO b) -> (a -> IO c)
```

Composition of programs – computations with context attached.

Examples of other computations:

- Async functions
- Random Variables
- Missing Data

Let's see these in action

Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

Monads are Applicative Functors with even more power and expressiveness.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)
```

Both `bind` and `join` can be defined in terms of the other.

```
laws Monad where
  bind (pure x) f == f x
  bind m pure == m
  bind (bind m f) g == bind m (\x -> bind (f x) g)
```

Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

Monads are Applicative Functors with even more power and expressiveness.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)
```

Laws easier to express (and more familiar!) in terms of `kleisli`:

```
kleisli pure f == f
kleisli f pure == f
kleisli (kleisli f g) h == kleisli f (kleisli g h)
```

Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

Monads are Applicative Functors with even more power and expressiveness.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)

implements Monad Maybe where
    bind : Maybe a -> (a -> Maybe b) -> Maybe b
    bind Nothing _ =
    bind (Some x) f =

    join : Maybe (Maybe a) -> Maybe a
    join Nothing =
    join (Some m) =
```

Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

Monads are Applicative Functors with even more power and expressiveness.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)

implements Monad Maybe where
    bind : Maybe a -> (a -> Maybe b) -> Maybe b
    bind Nothing _ = Nothing
    bind (Some x) f = f x

    join : Maybe (Maybe a) -> Maybe a
    join Nothing = Nothing
    join (Some m) = m
```

Two Useful Examples

Let $\mathcal{A}$ be a set and let $\mathbb{P}(\mathcal{A})$ be its *power set*. Then $\mathbb{P}$ is a monad with

$$\begin{aligned}\text{pure}(x) &= \{x\} \\ \text{join}(\mathcal{P}) &= \bigcup_{\mathcal{B}\in\mathcal{P}} \mathcal{B}\end{aligned}$$

where $x \in \mathcal{A}$ and $\mathcal{P} \in \mathbb{P}(\mathbb{P}(\mathcal{A}))$. (Note that join maps to $\mathbb{P}(\mathcal{A})$.)

Two Useful Examples

Let $\mathcal{A}$ be a set and let $\mathbb{P}(\mathcal{A})$ be its *power set*. Then $\mathbb{P}$ is a monad with

$$\mathrm{pure}(x) = \{x\}$$
$$\mathrm{join}(\mathcal{P}) = \bigcup_{\mathcal{B} \in \mathcal{P}} \mathcal{B}$$

where $x \in \mathcal{A}$ and $\mathcal{P} \in \mathbb{P}(\mathbb{P}(\mathcal{A}))$. (Note that join maps to $\mathbb{P}(\mathcal{A})$.)

Let $\mathcal{F}$ be a finite set and let $\mathbb{D}(\mathcal{F})$ be the set of probability distributions supported in $\mathcal{F}$. Then $\mathbb{D}$ is a monad with

$$\mathrm{pure}(x) = \delta_x$$
$$\mathrm{bind}(d, c) = \sum_{x \in \mathcal{F}} c(\blacksquare \mid x)\, d(x).$$

What are d and c here?

Two Useful Examples

Let $\mathcal{A}$ be a set and let $\mathbb{P}(\mathcal{A})$ be its *power set*. Then $\mathbb{P}$ is a monad with

$$\begin{aligned} \text{pure}(x) &= \{x\} \\ \text{join}(\mathcal{P}) &= \bigcup_{\mathcal{B} \in \mathcal{P}} \mathcal{B} \end{aligned}$$

where $x \in \mathcal{A}$ and $\mathcal{P} \in \mathbb{P}(\mathbb{P}(\mathcal{A}))$. (Note that join maps to $\mathbb{P}(\mathcal{A})$.)

Let $\mathcal{F}$ be a finite set and let $\mathbb{D}(\mathcal{F})$ be the set of probability distributions supported in $\mathcal{F}$. Then $\mathbb{D}$ is a monad with

$$\begin{aligned} \text{pure}(x) &= \delta_x \\ \text{bind}(d, c) &= \sum_{x \in \mathcal{F}} c(\blacksquare \mid x)\, d(x). \end{aligned}$$

What are d and c here?

We can think of these monads as representing values in an enhanced context.

Brief Demo

FP-Concepts (nearing completion, but see documents)

Plan

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 = x :: (iterate f (f x))     -- a lazy sequence

iterate (next x) init   -- lazy sequence of sqrt approximations

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

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

a_sqrt init tol   = within tol (iterate (next x) init)
r_sqrt init tol x = relative tol (iterate (next x) init)
```

These same primitives apply to give us other approximations, e.g., numerical differentiation, integration, ...

Implementation: Lazy Sequences

`iterate f x = x :: (iterate f (f x))` creates a ***lazy sequence*** of the form $x, f(x), f(f(x)), f(f(f(x))), \ldots$.

Let's create an implementation of `iterate` as well as functions

- `take n lazy_seq` that returns a list of the first n items from `lazy_seq`.
- `drop n lazy_seq` that returns the tail of `lazy_seq` that drops the first n items.
- `split_at n lazy_seq` that returns a pair: a list with the first n items and the lazy sequence of remaining items.

We can then use this to implement the above.

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 (a :: (b :: rest))
  = ((b * (2**n) - a) / (2**n - 1)) :: (sharpen n (b :: rest))

order (a :: (b :: (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

Recap: Functors

Intro to Effects and Specialized Functors

An Example From Hughes

A Quick Tour of Zippers

Appendix: Another Hughes Example

Zippers for Trees

Zippers are a data structure that enable the exploration and modification of other data structures (like trees) in a functional way.

Goals: Move freely through a tree, allow local modifications that can maintain the original tree while efficiently sharing structure.

Metaphor: Moving through the directory/folder tree on your computer.

Plan

Minimax Analysis of Game Trees

Consider a traditional, two-player perfect-information game like tic-tac-toe or chess. We can analyze a game by looking at the “game tree” and scoring positions heuristically.

Assume we have types `Player` and `Position`. For instance,

```
data Player = X | O
data Position = TopLeft | TopMid | TopRight | ... | BotRight
```

We will let these be generic types as they can apply to any game.

To keep things simple, we’ll start by ignoring the player, assuming that player can be determined from a position.

Minimax Analysis of Game Trees (cont'd)

We have a function

```
moves : Position -> List Position
```

and define (using rose trees)

```
propagate : (a -> List a) -> a -> Tree a
propagate f x = Node { value = x
                     , children = map (propagate f) (f x)
                     }
gameTree : Position -> Tree Position
gameTree = propagate moves
```

Minimax Analysis of Game Trees (cont'd)

We could handle the player as follows.

```
next : Player -> Player
moves : Player -> Position -> List Position
```

defining

```
propagate : (a -> List a) -> (a -> List a) -> a -> Tree a
propagate f1 f2 x = Node { value = x
                         , children = map (propagate f2 f1) (f1 x)
                         }
gameTree : Player -> Position -> Tree Position
gameTree player = propagate (moves player) (moves (next player))
```

But we'll keep to the simple version in what follows.

Minimax Analysis of Game Trees (cont'd)

Now imagine that we have some heuristic static evaluator for some position:

```
static : Position -> Number
```

Assume that the results are negative when they favor one player and positive when they favor another. This is a local guess that we will refine by analyzing the game tree. Note that `map static : Tree Position -> Tree Number`.

To extend our static analyzer, we lookahead in the tree, taking account of the best (greedy) move at each stage.

```
maximize : Tree Number -> Number
maximize (Node v []) = v
maximize (Node v sub) = max (map minimize sub)

minimize : Tree Number -> Number
minimize (Node v []) = v
minimize (Node v sub) = min (map maximize sub)

evaluate : Position -> Number
evaluate = maximize . map static . gameTree
```

This is fine, but it might not terminate. (Why?) And it can take a long time in any case.

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

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

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

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

Minimax Analysis of Game Trees (cont'd)

We need to prune the tree.

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

Now we have a more realistic evaluation

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

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.

The End