

Applicative Functors

Hidden in plain view...

Jim Powers

Patch.com

NYC Functional Programming Meetup

25 October 2011

Inspiration

```
buildName :: String -> Maybe String -> String -> String  
buildName f m l = f ++ " " ++ (maybe "" id m) ++ " " ++ l
```

Inspiration

```
buildName :: String -> Maybe String -> String -> String  
buildName f m l = f ++ " " ++ (maybe "" id m) ++ " " ++ l
```

Won't compile:

```
desired :: Maybe String -> Maybe String -> Maybe String -> Maybe String  
desired f m l = buildName <$> f <$> (Just m) <$> l
```

Inspiration

```
buildName :: String -> Maybe String -> String -> String
buildName f m l = f ++ " " ++ (maybe "" id m) ++ " " ++ l
```

Won't compile:

```
desired :: Maybe String -> Maybe String -> Maybe String -> Maybe String
desired f m l = buildName <$> f <$> (Just m) <$> l
```

Not bad:

```
nice :: Maybe String -> Maybe String -> Maybe String -> Maybe String
nice f m l = do
  fn <- f
  ln <- l
  return $ buildName fn m ln
```

Inspiration

FUNCTIONAL PEARL

Applicative programming with effects

CONOR MCBRIDE

University of Nottingham

ROSS PATERSON

City University, London

Abstract

In this paper, we introduce **Applicative** functors—an abstract characterisation of an applicative style of effectful programming, weaker than **Monads** and hence more widespread. Indeed, it is the ubiquity of this programming pattern that drew us to the abstraction. We retrace our steps in this paper, introducing the applicative pattern by diverse examples, then abstracting it to define the **Applicative** type class and introducing a bracket notation which interprets the normal application syntax in the idiom of an **Applicative** functor. Further, we develop the properties of applicative functors and the generic operations they support. We close by identifying the categorical structure of applicative functors and examining their relationship both with **Monads** and with **Arrows**.

Inspiration

- ▶ Applicative?

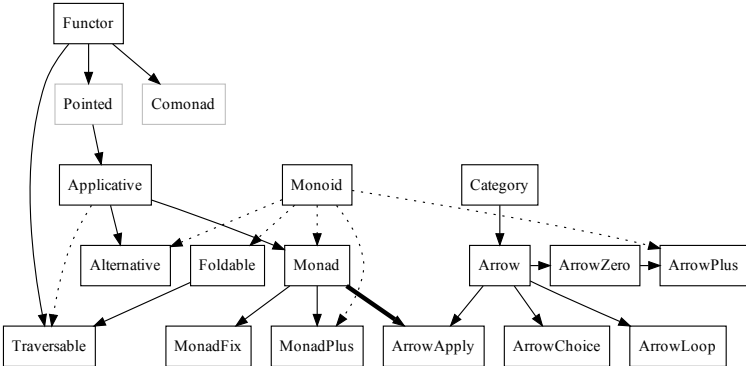
Inspiration

- ▶ Applicative?
- ▶ Functors?

Inspiration

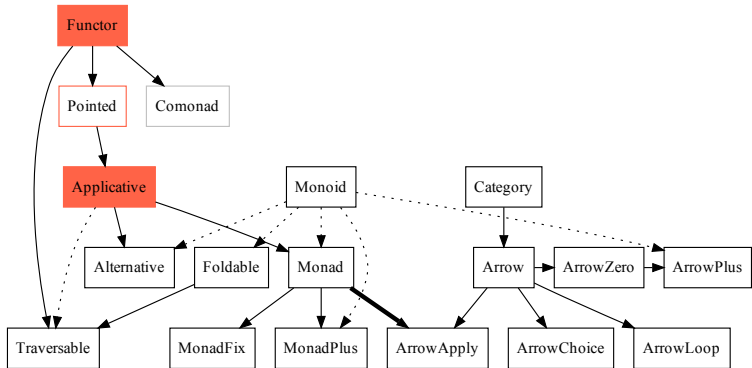
- ▶ Applicative?
- ▶ Functors?
- ▶ Effects?

Family Tree



Typeclassopedia, Brent Yorgey, TMR-issue13

Family Tree



Functor

class Functor f where

`fmap` :: (a -> b) -> f a -> f b

`(<$>)` :: Functor f => (a -> b) -> f a -> f b

`(<$>)` = `fmap`

Functor

class Functor f where

`fmap` :: (a -> b) -> f a -> f b

`(<$>)` :: Functor f => (a -> b) -> f a -> f b

`(<$>)` = `fmap`

Functors enable performing a computation within a context. The function does not know anything about the context it is running in.

Functor

class Functor f where

`fmap` :: (a -> b) -> f a -> f b

`(<$>)` :: Functor f => (a -> b) -> f a -> f b

`(<$>)` = `fmap`

Functor Laws

`fmap id = id`

`fmap (g . h) = fmap g . fmap h`

Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor Maybe where
```

```
  fmap _ Nothing = Nothing
```

```
  fmap f (Just x) = Just (f x)
```

Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor Maybe where
```

```
  fmap _ Nothing = Nothing
```

```
  fmap f (Just x) = Just (f x)
```

Example

```
mFunc :: (Show a, Show b) => (a -> b) -> Maybe a -> Maybe b
```

```
mFunc f x = f <$> x
```

Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor [] where
```

```
  fmap = map
```


Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor [] where
```

```
  fmap = map
```

Example

```
lFunc :: (Show a, Show b) => (a -> b) -> [a] -> [b]
```

```
lFunc f x = f <$> x
```

Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor (Either b) where
```

```
  fmap f (Left l)  = Left l
```

```
  fmap f (Right r) = Right (f r)
```

Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor (Either b) where
```

```
  fmap f (Left l)  = Left l
```

```
  fmap f (Right r) = Right (f r)
```

Example

```
eFunc :: (Show a, Show c) => (a -> c) -> (Either [String] a) ->  
      (Either [String] c)
```

```
eFunc f x = f <$> x
```

Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor ((->) t) where
```

```
  fmap f g = f . g
```

Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

```
(<$>) = fmap
```

Common Functor Instances

```
instance Functor ((->) t) where
```

```
  fmap f g = f . g
```

Example

```
fFunc :: (Show a, Show b, Show c) => (b -> c) -> (a -> b) -> (a -> c)
```

```
fFunc f x = f <$> x
```

Pointed

```
class Functor f => Pointed f where  
  pure :: a -> f a
```

Pointed

```
class Functor f => Pointed f where  
  pure :: a -> f a
```

The **Pointed** typeclass is not represented in the Haskell standard library by itself. Conceptually, **Pointed** presents the ability to **lift** an ordinary value into a context, thus creating a **point** in that context.

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```


Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

The **Applicative** combines that ability to create a **point** within a context, as well as the ability to apply a **lifted** function within that context.

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

The **Applicative** combines that ability to create a **point** within a context, as well as the ability to apply a **lifted** function within that context.

Each context expresses how to apply the lifted function in its own particular way. Also known as its **idiom**.

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Applicative Laws

```
fmap g x = pure g <*> x
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  (<*>) :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative Maybe where  
  pure = return  
  (<*>) = ap
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative Maybe where  
  pure = return  
  (<*>) = ap
```

Example

```
mAplic :: (a -> b) -> Maybe a -> Maybe b  
mAplic f x = pure f <*> x
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  (<*>) :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative [] where  
  pure = return  
  (<*>) = ap
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative [] where  
  pure = return  
  (<*>) = ap
```

Example

```
lAplic :: (Show a, Show b) => (a -> b) -> [a] -> [b]  
lAplic f x = pure f <*> x
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative (Either e) where  
  pure          = Right  
  Left e <*> _ = Left e  
  Right f <*> r = fmap f r
```


Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative (Either e) where  
  pure      = Right  
  Left e <*> _ = Left e  
  Right f <*> r = fmap f r
```

Example

```
eAplic :: (Show a, Show c) => (a -> c) -> (Either [String] a) ->  
  (Either [String] c)  
eAplic f x = pure f <*> x
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative ((->) a) where  
  pure = const  
  (<*>) f g x = f x (g x)
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Common Functor Instances

```
instance Applicative ((->) a) where  
  pure = const  
  (<*>) f g x = f x (g x)
```

Example

```
fAplic :: (Show a, Show b, Show c) => (b -> c) -> (a -> b) -> (a -> c)  
fAplic f x = pure f <*> x
```

Inspiration

```
buildName :: String -> Maybe String -> String -> String
buildName f m l = f ++ " " ++ (maybe "" id m) ++ " " ++ l
```

Won't compile:

```
desired :: Maybe String -> Maybe String -> Maybe String -> Maybe String
desired f m l = buildName <$> f <$> (Just m) <$> l
```

Not bad:

```
nice :: Maybe String -> Maybe String -> Maybe String -> Maybe String
nice f m l = do
  fn <- f
  ln <- l
  return $ buildName fn m ln
```

Inspiration

```
buildName :: String -> Maybe String -> String -> String
buildName f m l = f ++ " " ++ (maybe "" id m) ++ " " ++ l
```

Won't compile:

```
desired :: Maybe String -> Maybe String -> Maybe String -> Maybe String
desired f m l = buildName <$> f <$> (Just m) <$> l
```

Not bad:

```
nice :: Maybe String -> Maybe String -> Maybe String -> Maybe String
nice f m l = do
  fn <- f
  ln <- l
  return $ buildName fn m ln
```

Sweet:

```
sweet :: Maybe String -> Maybe String -> Maybe String -> Maybe String
sweet f m l = buildName <$> f <*> (pure m) <*> l
```

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Applicatives can be used as a tool to remove boilerplate!

Applicative

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Applicatives can be used as a tool to remove boilerplate!

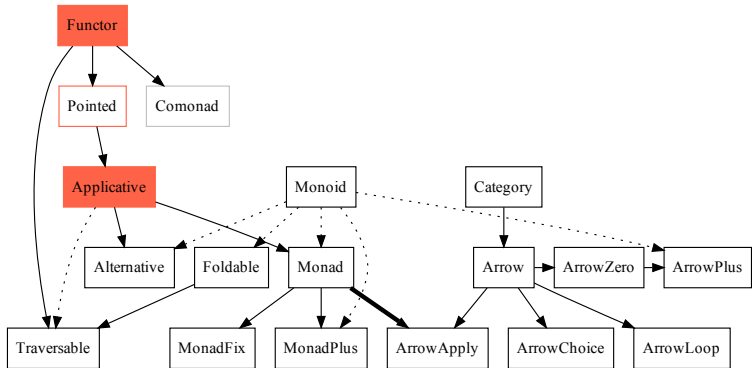
Orly? How?

Applicative

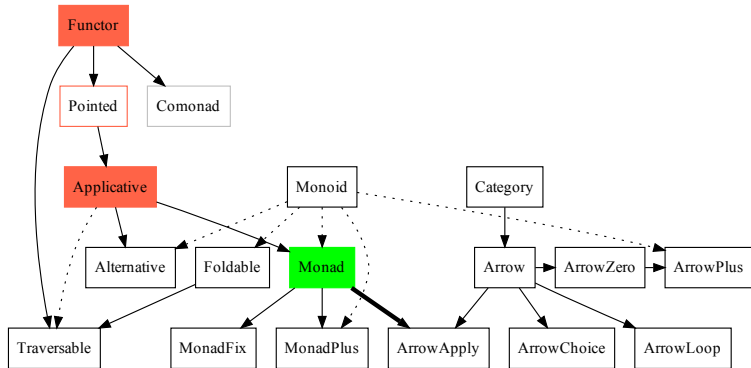
```
class Functor f => Applicative f where  
  pure :: a -> f a  
  <*> :: f (a -> b) -> f a -> f b
```

Predictable web example!

Family Tree



Family Tree



Mo'nads!



<http://twitpic.com/5j3x2t>

Web Example

```
import Data.List (lookup)
import Web.MaybeResult

type Request = [(String,String)]

required :: Request -> String -> MaybeResult String
required r k = maybe (failure $ "Required value for: " ++ k ++ " missing")
  success (lookup k r)

opt :: Request -> String -> MaybeResult (Maybe String)
opt r k = success (lookup k r)
```

Web Example

```
data ResponseValue = Value String
                   | Stream [String]
                   deriving (Show)

type Response = MaybeResult ResponseValue
```

Web Example

Reader Monad

```
newtype Reader r a = Reader {  
  runReader :: r -> a  
}
```

```
instance Monad (Reader r) where  
  return a = Reader $ \_ -> a  
  m >>= k = Reader $ \r -> runReader (k (runReader m r)) r
```

```
myRequired :: String -> Reader Request (MaybeResult String)  
myRequired k =  
  asks (doRead k) >>= return  
  where  
    doRead k r = required r k
```

```
myOptional :: String -> Reader Request (MaybeResult (Maybe String))  
myOptional k =  
  asks (doRead k) >>= return  
  where  
    doRead k r = opt r k
```

Web Example

```
data MaybeResult a = Failure [String]
                    | Success a
    deriving (Eq, Ord, Show, Data, Typeable)

failure :: forall a. String -> MaybeResult a
failure s = Failure [s]

success :: forall a. a -> MaybeResult a
success s = Success s
```

Web Example

```
handler :: Reader Request Response
```

```
handler = do
```

```
    first <- myRequired "first"
```

```
    middle <- myOptional "middle"
```

```
    last <- myRequired "last"
```

```
    return (fmap Value (buildName <$> first <*> middle <*> last))
```

```
browser :: Request -> String
```

```
browser r = case (runReader handler r) of
```

```
    Success (Value v) -> "Page rendered STRING: "++v
```

```
    Success (Stream vs) -> "Page rendered STREAM: "++(show vs)
```

```
    Failure v -> "BZZT! Invalid request: "++(show v)
```


Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
    putStrLn $ show $ browser goodRequest
```

```
    putStrLn $ show $ browser badRequest
```

```
    putStrLn $ show $ browser badRequest1
```

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
  putStrLn $ show $ browser goodRequest
```

```
  putStrLn $ show $ browser badRequest
```

```
  putStrLn $ show $ browser badRequest1
```

► => “Page rendered STRING: James Powers”

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
  putStrLn $ show $ browser goodRequest
```

```
  putStrLn $ show $ browser badRequest
```

```
  putStrLn $ show $ browser badRequest1
```

- ▶ => "Page rendered STRING: James Powers"
- ▶ => "BZZT! Invalid request: [Required value for:last missing]"

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
  putStrLn $ show $ browser goodRequest
```

```
  putStrLn $ show $ browser badRequest
```

```
  putStrLn $ show $ browser badRequest1
```

- ▶ ==> "Page rendered STRING: James Powers"
- ▶ ==> "BZZT! Invalid request: [Required value for:last missing]"
- ▶ ==> "BZZT! Invalid request: [Required value for:first missing]"

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
  putStrLn $ show $ browser goodRequest
```

```
  putStrLn $ show $ browser badRequest
```

```
  putStrLn $ show $ browser badRequest1
```

- ▶ ==> "Page rendered STRING: James Powers"
- ▶ ==> "BZZT! Invalid request: [Required value for:last missing]"
- ▶ ==> "BZZT! Invalid request: [Required value for:first missing]"
- ▶ ==> Hrm...

Web Example

```
data MaybeResult a = Failure [String]
                    | Success a
    deriving (Eq, Ord, Show, Data, Typeable)

instance Functor MaybeResult where
    fmap _ (Failure s) = Failure s
    fmap f (Success x) = Success (f x)

instance Applicative MaybeResult where
    pure = Success
    Failure e1 <*> Failure e2 = Failure (e1 ++ e2)
    Failure e1 <*> Success _  = Failure e1
    Success _  <*> Failure e2 = Failure e2
    Success f  <*> Success a  = Success (f a)

failure :: forall a. String -> MaybeResult a
failure s = Failure [s]

success :: forall a. a -> MaybeResult a
success s = Success s
```

Web Example

```
handler :: Reader Request Response
```

```
handler = do
```

```
    name <- asks buildNamehandler
```

```
    return (fmap Value name)
```

```
buildNamehandler :: Request -> MaybeResult String
```

```
buildNamehandler r = buildName <$> (required r "first") <*> (opt r  
"middle") <*> (required r "last")
```

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
  putStrLn $ show $ browser goodRequest
```

```
  putStrLn $ show $ browser badRequest
```

```
  putStrLn $ show $ browser badRequest1
```

► => “Page rendered STRING: James Powers”

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
  putStrLn $ show $ browser goodRequest
```

```
  putStrLn $ show $ browser badRequest
```

```
  putStrLn $ show $ browser badRequest1
```

- ▶ => "Page rendered STRING: James Powers"
- ▶ => "BZZT! Invalid request: [Required value for:last missing]"

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
```

```
  putStrLn $ show $ browser goodRequest
```

```
  putStrLn $ show $ browser badRequest
```

```
  putStrLn $ show $ browser badRequest1
```

- ▶ ==> "Page rendered STRING: James Powers"
- ▶ ==> "BZZT! Invalid request: [Required value for:last missing]"
- ▶ ==> "BZZT! Invalid request: [Required value for:first missing, Required value for:last missing]"

Web Example

```
goodRequest = [("first", "James"), ("last", "Powers")]
```

```
badRequest = [("first", "James")]
```

```
badRequest1 = [("middle", "Matthew")]
```

```
main = do
  putStrLn $ show $ browser goodRequest
  putStrLn $ show $ browser badRequest
  putStrLn $ show $ browser badRequest1
```

- ▶ ==> "Page rendered STRING: James Powers"
- ▶ ==> "BZZT! Invalid request: [Required value for:last missing]"
- ▶ ==> "BZZT! Invalid request: [Required value for:first missing, Required value for:last missing]"
- ▶ ==> Whoa! Now dats powah!

Applicatives vs. Monads

- ▶ Monads enable **conditional** (short-circuited) computation.

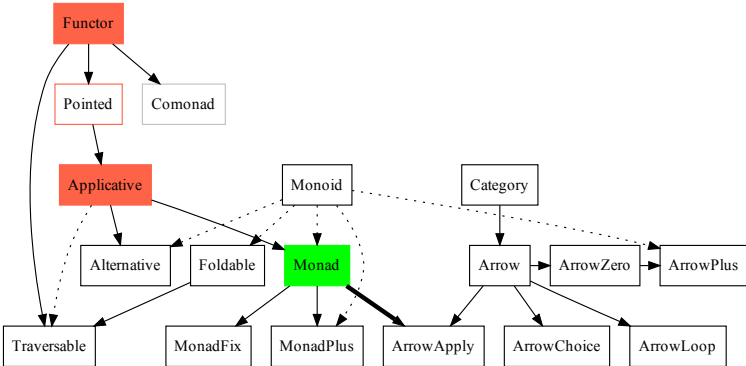
Applicatives vs. Monads

- ▶ Monads enable **conditional** (short-circuited) computation.
- ▶ Monads can be **stacked** through **monad transformers**.

Applicatives vs. Monads

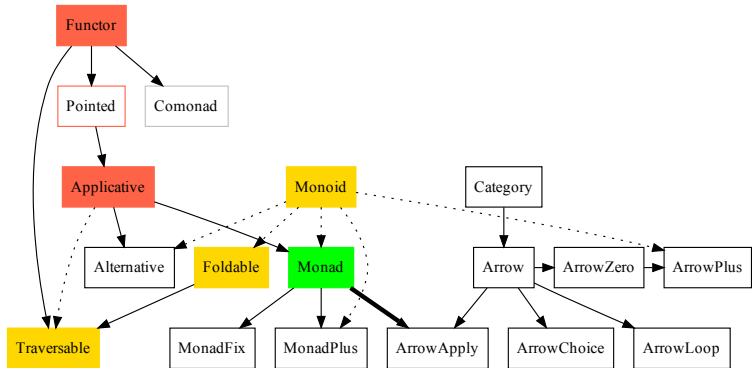
- ▶ Monads enable **conditional** (short-circuited) computation.
- ▶ Monads can be **stacked** through **monad transformers**.
- ▶ Monads are much more restrictive and hence less common.

Family Tree



Typeclassopedia, Brent Yorgey, TMR-issue13

Family Tree



Monoid

```
class Monoid a where  
  mempty  :: a  
  mappend :: a -> a -> a
```

Foldable

```
class Foldable t where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldl :: (a -> b -> a) -> a -> t b -> a
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
```

Must Read!

Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

Erik Meijer *

Maarten Fokkinga [†]

Ross Paterson [‡]

Abstract

We develop a calculus for lazy functional programming based on recursion operators associated with data type definitions. For these operators we derive various algebraic laws that are useful in deriving and manipulating programs. We shall show that all example functions in Bird and Wadler's "Introduction to Functional Programming" can be expressed using these operators.

All sorts of folds...

- ▶ Anamorphism

All sorts of folds...

- ▶ Anamorphism
- ▶ Apomorphism

All sorts of folds...

- ▶ Anamorphism
- ▶ Apomorphism
- ▶ Hylomorphism

All sorts of folds...

- ▶ Anamorphism
- ▶ Apomorphism
- ▶ Hylomorphism
- ▶ Paramorphism

Must Read!

The Essence of the Iterator Pattern

Jeremy Gibbons and Bruno C. d. S. Oliveira

Oxford University Computing Laboratory

Wolfson Building, Parks Road, Oxford OX1 3QD, UK

<http://www.comlab.ox.ac.uk/jeremy.gibbons/>

<http://www.comlab.ox.ac.uk/bruno.oliveira/>

Abstract

The ITERATOR pattern gives a clean interface for element-by-element access to a collection, independent of the collection's shape. Imperative iterations using the pattern have two simultaneous aspects: *mapping* and *accumulating*. Various existing functional models of iteration capture one or other of these aspects, but not both simultaneously. We argue that McBride and Paterson's *applicative functors*, and in particular the corresponding *traverse* operator, do exactly this, and therefore capture the essence of the ITERATOR pattern. Moreover, they do so in a way that nicely supports modular programming. We present some axioms for traversal, discuss modularity concerns, and illustrate with a simple example, the *wordcount* problem.

Traversable

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
```

Traversable

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
```

```
traverse id [Just 1, Just 2]
```

```
=> Just [1,2]
```

Traversable

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
```

```
traverse id [Just 1, Just 2]
```

```
=> Just [1,2]
```

```
traverse id [Just 1, Nothing, Just 2]
```

```
=> Nothing
```