



Advanced Functional Programming

03 - Applicative, Traversable, and Foldable

Wouter Swierstra & Trevor L. McDonell

Utrecht University

Beyond the monad

So far, we have seen how *monads* define a common abstraction over many programming patterns.

This kind of abstraction occurs more often in libraries.

In this lecture we will cover:

- applicative functors
- foldable
- traversable
- arrows

We'll motivate the need for applicative functors starting with *examples*.

Sequencing IO operations

```
sequenceIO :: [IO a] -> IO [a]
sequenceIO []      = return []
sequenceIO (c : cs) = do x  <- c
                          xs <- sequenceIO cs
                          return (x : xs)
```

There is nothing 'wrong' with this code – but using `do` notation may seem like overkill. The variable `x` isn't used in the rest of the computation!

We would like to 'apply' one monadic computation to another.

Using liftM2

The liftM2 function is defined as follows:

```
liftM2 :: (a -> b -> c) -> m a -> m b -> m c
liftM2 f ma mb = do a <- ma
                   b <- mb
                   return (f a b)
```

Using liftM2 we can write:

```
sequenceIO :: [IO a] -> IO [a]
sequenceIO [] = return []
sequenceIO (c:cs) = liftM2 (:) c (sequenceIO cs)
```

This even works for *any monad*, not just the IO monad.

More lifting functions

- `liftM` (or `fmap`) lifts functions $a \rightarrow b$
- `liftM2` lifts functions $a \rightarrow b \rightarrow c$
- ...
- `liftM5` lifts functions $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$

Do we need a `liftMn` for every n ?

Time to derive liftMn!

Using ap

The ap function is defined as follows:

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = do f <- mf
              x <- mx
              return (f x)
```

Using ap we can write:

```
sequenceIO :: [IO a] -> IO [a]
sequenceIO [] = return []
sequenceIO (c:cs) =
  return (:) 'ap' c 'ap' sequenceIO cs
```

Evaluating expressions

Another example:

```
data Expr v = Var v | Val Int | Add (Expr v) (Expr v)
```

```
type Env v = Map v Int
```

```
eval :: Expr v -> Env v -> Int
```

```
eval (Var v) env = lookup v env
```

```
eval (Val i) env = i
```

```
eval (Add l r) env = (eval l env) + (eval r env)
```

Once again, we are passing around an environment that is only really used in the Var branch.

An applicative alternative

```
const :: a -> (env -> a)
```

```
const x = \env -> x
```

```
s :: (env -> a -> b) -> (env -> a) -> (env -> b)
```

```
s ef es env = (ef env) (es env)
```

```
eval :: Expr v -> Env v -> Int
```

```
eval (Var v) = lookup v
```

```
eval (Val i) = const i
```

```
eval (Add l r) =
```

```
  const (+) 's' (eval l) 's' (eval r)
```

The `s` combinator lets us 'apply' one computation expecting an environment to another.

Transposing matrices

```
transpose :: [[a]] -> [[a]]
transpose []           = repeat []
transpose (xs : xss) = zipWith (:) xs (transpose xss)
```

Can we play the same trick and find a combinator that will 'apply' a list of functions to a list of arguments?

```
zapp :: [a -> b] -> [a] -> [b]
zapp (f : fs) (x : xs) = (f x) : (zapp fs xs)
```

```
transpose (xs : xss) =
  repeat (:) 'zapp' xs 'zapp' transpose xss
```

What is the pattern?

What do these functions have in common?

```
ap   :: IO (a -> b)    -> IO a      -> IO b
s    :: (env -> a -> b) -> (env -> a) -> (env -> b)
zapp :: [a -> b]       -> [a]        -> [b]
```

Applicative (applicative functors)

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

Note that Functor is a superclass of Applicative.

We can also define fmap in terms of the applicative operations:

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

Applicative (applicative functors)

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

Note that Functor is a superclass of Applicative.

We can also define fmap in terms of the applicative operations:

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

(<$>) f fx = pure f <*> fx
```

Using Applicative operators

This type class leads to a certain code style:

```
sequenceIO :: [IO a] -> IO [a]
```

```
sequenceIO [] = return []
```

```
sequenceIO (c:cs) = (:) <$> c <*> sequenceIO cs
```

Relating Applicative functors and Monads

- Every monad can be given an applicative functor interface.

```
instance Monad m => Applicative m where
```

```
pure :: a -> m a
```

```
pure = return
```

```
mf <*> mx = do f <- mf; x <- mx; return (f x)
```

- But this may not always be the 'right' choice. For example, we have seen the 'zapp' applicative instance for lists; using the instance arising from the list monad gives very different behaviour!
- But not every applicative functor is a monad...

Monads vs. applicative functors (1)

```
(<*>      :: Applicative f => f (a -> b) -> f a -> f b
```

```
flip (>>=) :: Monad m      => (a -> m b) -> m a -> m b
```

- The arguments to <*> are (typically) first-order structures (that may contain higher-order data).
- Monadic bind is inherently higher order.
- With monads, subsequent actions can depend on the results of effects: depending on the character the user enters, respond differently.

Monads vs applicative functors (2)

- There are more Applicative functors than there are monads; but monads are more powerful!
- If you *have* an Applicative functor, that's good; having a monad is better.
- If you *need* a monad, that's good; only needing an Applicative functor is better.
- With applicative functors, the structure is statically determined (and can be analyzed or optimized). Consider the following example:

```
miffy :: Monad m => m Bool -> m a -> m a -> m a
```

Imprecise but catchy slogans

Monads are programmable semi-colons!

Applicatives are programmable function application!

Applicative functor laws

- identity

```
pure id <*> u = u
```

- composition

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

- homomorphism

```
pure f <*> pure x = pure (f x)
```

- interchange

```
u <*> pure x = pure (\f -> f x) <*> u
```

Spot the pattern

```
sequenceIO :: [IO a] -> IO [a]
```

```
sequenceIO [] = pure []
```

```
sequenceIO (c : cs) =
```

```
  (:) <$> c <*> sequenceIO cs
```

```
cartesian_product :: [[a]] -> [[a]]
```

```
cartesian_product [] = pure []
```

```
cartesian_product (xs : xss) =
```

```
  (:) <$> xs <*> cartesian_product xss
```

Both these functions take a *list* of applicative actions as argument.

They traverse this list, performing the actions one by one, collecting the results in a list.

Traversing lists

We can define a new function to capture this pattern:

```
sequence :: Applicative f => [f a] -> f [a]
sequence []      = pure []
sequence (x:xs) = pure (:) <*> x <*> sequence xs
```

Clearly we can traverse lists in this fashion – but what other data types support such an operation?

The Traversable class captures those types that can be traversed in this fashion:

```
class Traversable t where
  traverse  :: Applicative f
            => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f
            => t (f a) -> f (t a)
```

It requires a slightly more general traverse than the one we have seen so far.

Traversable

The Traversable class captures those types that can be traversed in this fashion:

```
class Traversable t where
  traverse  :: Applicative f
            => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f
            => t (f a) -> f (t a)
```

It requires a slightly more general traverse than the one we have seen so far.

Question:

Define traverse and sequenceA in terms of each other

Traversable with defaults

```
class Traversable t where

  traverse  :: Applicative f
            => (a -> f b) -> t a -> f (t b)
  traverse f = sequenceA . fmap f

  sequenceA :: Applicative f
            => t (f a) -> f (t a)
  sequenceA = traverse id
```


Traversable: example

```
data Expr v = Var v | Val Int | Add (Expr v) (Expr v)
```

```
instance Traversable Expr where
```

```
  traverse :: Applicative f => (a -> f b) => Expr a -> f (Expr b)
```

```
  traverse f (Var v)   = Var <$> f v
```

```
  traverse f (Val x)   = pure (Val x)
```

```
  traverse f (Add l r) =
```

```
    Add <$> traverse f l <*> traverse f r
```

In general, traverse is just like fmap – only in applicative style.

Introducing Foldable

In the Haskell libraries, Traversable is defined slightly differently.

```
class (Functor t, Foldable t)  
  => Traversable t where
```

What is the Foldable class?

Folding a list

The `foldr` function on lists captures a common pattern – think of it as a functional for-loop.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr f y [] = y
```

```
foldr f y (x:xs) = f x (foldr f y xs)
```

We can use it to define all kinds of simple list traversals:

```
sum = foldr (+) 0
```

```
maximum = foldr max minBound xs
```

```
(++) = \xs ys -> foldr (:) ys xs
```

```
concat = foldr (++) []
```

```
map = \f -> foldr (\x xs -> f x : xs) []
```

Folding: beyond lists

There are many other data types that support some form of fold operator.

```
data Tree a = Empty
           | Leaf a
           | Node (Tree a) (Tree a)

foldTree :: (a -> b -> b) -> b -> Tree a -> b
foldTree f y Empty      = y
foldTree f y (Leaf x)   = f x y
foldTree f y (Node l r) =
  foldTree f (foldTree f y r) l
```

Note that *generic programming* gives a slightly more precise account.

```
class Foldable f where
```

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

```
  foldMap :: Monoid m => (a -> m) -> f a -> m
```

Sometimes it can be easier to define the `foldMap` function – but what is a `Monoid`?

Intermezzo: monoids

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

Here `mempty` should be the unit of the associative operator `mappend`.

Monoids everywhere

- Bool using `&&` and `True`
- Bool using `or` and `False`
- Int using `+` and `0`
- Int using `*` and `1`
- Int using `max` and `minBound`
- List a using `++` and `[]`
- Imperative programs using `;` and `skip`
- `a -> a` using `.` and `id`
- ...

Monoids pop up everywhere!

Defining foldMap

Instead of defining `fold`, sometimes it can be easier to define `foldMap`:

```
foldMap :: Monoid m => (a -> m) -> Tree a -> m
```

```
foldMap f Empty      = mempty
```

```
foldMap f (Leaf x)   = f x
```

```
foldMap f (Node l r) =  
    foldMap f l `mappend` foldMap f r
```

You need to apply `f` to all the `a` values in the tree and combine subtrees using `mappend`.

Why?

What is the point of all this abstraction?

We all agree (I hope!) that `foldr` is useful for lists.

```
sum      = foldr (+) 0
```

```
maximum = foldr max minBound xs
```

```
(++) ys = foldr (:) ys xs
```

```
concat  = foldr (++) []
```

```
map f   = foldr (\x xs -> f x : xs) []
```

Why?

What is the point of all this abstraction?

We all agree (I hope!) that `foldr` is useful for lists.

```
sum      = foldr (+) 0
```

```
maximum = foldr max minBound xs
```

```
(++) ys = foldr (:) ys xs
```

```
concat  = foldr (++) []
```

```
map f   = foldr (\x xs -> f x : xs) []
```

... but we can now give definitions that work for *any* foldable structure.

Why?

What is the point of all this abstraction?

We all agree (I hope!) that `foldr` is useful for lists.

```
sum    :: Foldable f => f Int -> Int
```

```
sum    = foldr (+) 0
```

```
maximum :: Foldable f => f Int -> Int
```

```
maximum = foldr max minBound
```

```
flatten :: Foldable f => f a -> [a]
```

```
flatten = foldMap (\x -> [x]) (++)
```

As a slightly less trivial example, consider the any function:

```
any :: (a -> Bool) -> [a] -> Bool
```

```
any p [] = False
```

```
any p (x:xs) = p x || any p xs
```

How can we generalize this to work on *any* traversable type?

Mighty Booleans

Let's start by finding the right monoidal structure.

Instead of defining an instance for `Bool`, introducing a new type can help clarify the monoidal structure we are using.

```
newtype Might = Might { might :: Bool }
```

```
instance Monoid Might where
```

```
  mempty = Might False
```

```
  (Might x) `mappend` (Might y) = Might (x || y)
```

Generic any

```
any :: Foldable f => (a -> Bool) -> f a -> Bool  
any p = might . foldMap (Maybe . p)
```

Many other functions can be generalized similarly.

The Foldable-Traversable Proposal

As of GHC 7.10, many Prelude functions have been generalized to work over *any* traversable structure – and not just lists.

Suppose we have a data type for binary trees, with the obvious traversable/foldable instances:

```
data Tree a where  
  Leaf  :: a -> Tree a  
  Node  :: Tree a -> Tree a -> Tree a
```

We can use the prelude functions we are used to over this data structure too.

Example folds over trees

```
> let t = (Node (Leaf 1) (Node (Leaf 2) (Leaf 3)))
```

```
> any isEven t
```

```
True
```

```
> length t
```

```
3
```

```
> elem 3 t
```

```
True
```

We no longer need to define specialized functions for trees.

Exercise

What is the foldable instance for `Maybe`?

What about the foldable instance for `pairs`?

But there are also quite surprising examples:

```
minimum (1,1000)
```

```
length (lookup 4 [(2,"Hello"), (4,"World"), (5,"!")])
```

```
null (lookup 3 [])
```

Drawbacks

But there are also quite surprising examples:

```
minimum (1,1000)
```

```
length (lookup 4 [(2,“Hello”), (4,“World”), (5,“!”)])
```

```
null (lookup 3 [])
```

Sometimes code may type check, where you would have liked to see a type error.

If applicative functors generalize the notion of application, can we find a similar abstraction over functions and function composition?

Yes! There is more than a decade of work investigating functional programming using *Arrows*.

```
class Arrow a where
```

```
  arr  :: (b -> c) -> a b c
```

```
  (>>>) :: a b c -> a c d -> a b d
```

```
  first :: a b c -> a (b,d) (c,d)
```

- Just like applicative functors and monads, arrows have several associated laws.
- Many programs using arrows require additional operations – similar to classes such as `MonadPlus`.
- GHC supports special syntax for programming with Arrows, similar to the `do` notation for monads.

Historical context

- Monads were originally studied in the context of program language semantics.
- Only later, was their importance for *structuring* programs discovered (and subsequently, modelling IO)
- Arrows (Hughes 2000) were proposed as an alternative to monads, but they have not been widely adopted.
- More recently, applicative functors have gained a lot of traction in the Haskell community (McBride and Paterson 2008), generalising the interface by Duponcheel and Swierstra (1996).
- Applicative functors, together with the associated Traversable and Foldable classes, are now part of the Haskell standard.

Why care?

Functional programmers are addicted to abstraction: as soon as they spot a pattern, they typically want to abstract over it.

The type classes we have seen today, such as monads, applicative functors, foldable, and traversable, all capture some common pattern.

Using these patterns can save you some boilerplate code.

But *understanding* these patterns can help guide your design.

Is my type a monad? Or is it just applicative? Can I find a Traversable instance? Why not?

Further reading

- *Applicative programming with effects*, McBride and Paterson
- *Monoids: Theme and Variations*, Brent Yorgey
- *Programming with arrows*, John Hughes
- *Idioms are oblivious, arrows are meticulous, monads are promiscuous*, Lindley, Wadler and Yallop
- and much much more!

Appendix: Time for type trickery!

Functor and Foldable from 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)
```

We can write `fmap` and `foldMap` using `traverse` or `sequenceA`

Functor and Foldable from 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)
```

We can write `fmap` and `foldMap` using `traverse` or `sequenceA`

Let's do it!

```
newtype Id a = Id { getId :: a }
```

```
myFmap :: Traversable f => (a -> b) -> f a -> f b
```

```
myFmap f = getId . traverse (Id . f)
```

foldMap via traverse

```
traverse :: Applicative f  
          => (a -> f b) -> t a -> f (t b)  
  
foldMap  :: Monoid m  
          => (a -> m)   -> t a -> m
```

foldMap via traverse

```
traverse :: Applicative f
          => (a -> f b) -> t a -> f (t b)

foldMap  :: Monoid m
          => (a -> m)   -> t a -> m
```

You need to find an Applicative which behaves like a monoid

```
myFoldMap f = getMonoidApplicative
             . traverse (MonoidApplicative . f)
```

foldMap via traverse using Const

The functor you need is called Const

```
data Const k a = Const { getConst :: k }
```

```
instance Functor (Const k) where
```

```
  fmap _ (Const x) = Const x
```

What about the Applicative instance?

foldMap via traverse using Const

```
instance Monoid k => Applicative (Const k) where
  pure = Const mempty
  Const x <*> Const y
    = Const (x `mappend` y)
```