

Functional programming

Lecture 04 — Kind, functors and applicatives

Stéphane Vialette

stephane.vialette@univ-eiffel.fr

March 21, 2023

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

What's the type of a type?

What's the type of a type?

Interlude - Phantom types

Functors

Applicative functors

Using newtype to make type class instances

- `kinds` are to types what types are to values.
- Just like values/terms can be classified into types, types can be classified into kinds.
- Just like we can use `:type` in GHCi to check the type of a term, we can use `:kind` to check the kind of a type.

Concrete types

All concrete types, have the kind of `*`.

```
λ> :kind Int
```

```
Int :: *
```

```
λ> :kind Bool
```

```
Bool :: *
```

```
λ> :kind [Int]
```

```
[Int] :: *
```

```
λ> :kind Int -> String
```

```
Int -> String :: *
```

```
λ> :kind Int -> String -> (Double,Double)
```

```
Int -> String -> (Double,Double) :: *
```

Kind

```
data T
```

```
λ> :kind T
```

```
T :: *
```

```
data T = P | Q | R
```

```
λ> :kind T
```

```
T :: *
```

Kind

```
data T a = P a
```

```
λ> :kind T
```

```
T :: * -> *
```

```
data T a = P a | Q a | R a
```

```
λ> :kind T
```

```
T :: * -> *
```

```
data T a = P a | Q a | R
```

```
λ> :kind T
```

```
T :: * -> *
```

Kind

```
data T a b = P a b
```

```
λ> :kind T
```

```
T :: * -> * -> *
```

```
data T a b = P a b | Q a b
```

```
λ> :kind T
```

```
T :: * -> * -> *
```

```
data T a b = P a | Q b
```

```
λ> :kind T
```

```
T :: * -> * -> *
```

```
data T a b = P a b | Q a | R
```

```
λ> :kind T
```

```
T :: * -> * -> *
```

Kind

```
data T a b c = P a b c
```

```
λ> :kind T
```

```
T :: * -> * -> * -> *
```

```
data T a b c d = P a b c d
```

```
λ> :kind T
```

```
T :: * -> * -> * -> * -> *
```

```
data T a b c d e = P a b c d e
```

```
λ> :kind T
```

```
T :: * -> * -> * -> * -> * -> *
```

Higher-Kinded Type

```
λ> :kind []  
[] :: * -> *
```

Higher-Kinded Type

```
data Maybe a = Nothing | Just a
```

```
λ> :kind Maybe
```

```
Maybe :: * -> *
```

```
λ> :kind Maybe Int
```

```
Maybe Int :: *
```

Higher-Kinded Type

```
data Either a b = Left a | Right b
```

```
λ> :kind Either
```

```
Either :: * -> * -> *
```

```
λ> :kind Either Int
```

```
Either Int :: * -> *
```

```
λ> :kind Either Int String
```

```
Either Int String :: *
```

Higher-Kinded Type

```
data NonEmpty f a = NonEmpty a (f a) deriving Show
```

Higher-Kinded Type

```
data NonEmpty f a = NonEmpty a (f a) deriving Show
```

```
λ> :type NonEmpty
```

```
NonEmpty :: a -> f a -> NonEmpty f a
```

```
λ> :kind NonEmpty
```

```
NonEmpty :: (* -> *) -> * -> *
```

Higher-Kinded Type

```
data NonEmpty f a = NonEmpty a (f a) deriving Show
```

```
λ> :type NonEmpty
```

```
NonEmpty :: a -> f a -> NonEmpty f a
```

```
λ> :kind NonEmpty
```

```
NonEmpty :: (* -> *) -> * -> *
```

```
λ> NonEmpty 1 [2,3,4]
```

```
NonEmpty 1 [2,3,4]
```

```
λ> :type NonEmpty 1 [2,3,4]
```

```
NonEmpty 1 [2,3,4] :: Num a => NonEmpty [] a
```

```
λ> NonEmpty 1 [True]
```

```
<interactive>: error:
```

```
No instance for (Num Bool) arising from the literal '1'
```

Higher-Kinded Type

```
data NonEmpty f a = NonEmpty a (f a) deriving Show
```

```
λ> :type NonEmpty
```

```
NonEmpty :: a -> f a -> NonEmpty f a
```

```
λ> :kind NonEmpty
```

```
NonEmpty :: (* -> *) -> * -> *
```

```
λ> NonEmpty 1 Nothing
```

```
NonEmpty 1 Nothing
```

```
λ> NonEmpty 1 (Just 2)
```

```
NonEmpty 1 (Just 2)
```

```
λ> :type NonEmpty 1 (Just 2)
```

```
NonEmpty 1 (Just 2) :: Num a => NonEmpty Maybe a
```

```
λ> NonEmpty 1 (Just True)
```

```
<interactive>: error:
```

```
No instance for (Num Bool) arising from the literal '1'
```

Constraints

The `Constraint` kind covers everything that can appear to the left of an `=>` arrow, including typeclass constraints:

Constraints

The `Constraint` kind covers everything that can appear to the left of an `=>` arrow, including typeclass constraints:

```
λ> :kind Show
Show :: * -> Constraint

λ> :kind Eq
Eq :: * -> Constraint

λ> :kind Ord
Ord :: * -> Constraint

λ> :kind Semigroup
Semigroup :: * -> Constraint

λ> :kind Monoid
Monoid :: * -> Constraint
```

Constraints

The `Constraint` kind covers everything that can appear to the left of an `=>` arrow, including typeclass constraints:

```
λ> :kind Functor
```

```
Functor :: (* -> *) -> Constraint
```

```
λ> :kind Applicative
```

```
Applicative :: (* -> *) -> Constraint
```

```
λ> :kind Monad
```

```
Monad :: (* -> *) -> Constraint
```

```
λ> :kind Foldable
```

```
Foldable :: (* -> *) -> Constraint
```

```
λ> :kind Traversable
```

```
Traversable :: (* -> *) -> Constraint
```

Higher-Kinded Type

```
data Collection f a = Collection (f a) deriving (Show)
```

This type takes a wrapper `f` (such as `[]`) and a concrete type `a` (such as `Int`) and returns a collection of `f a`

Higher-Kinded Type

```
data Collection f a = Collection (f a) deriving (Show)
```

This type takes a wrapper `f` (such as `[]`) and a concrete type `a` (such as `Int`) and returns a collection of `f a`

```
λ> :type Collection [1,2,3]
Collection [1,2,3] :: Num a => Collection [] a
```

```
λ> :type Collection ["Haskell", "rocks!"]
Collection ["Haskell", "rocks!"] :: Collection [] String
```

```
λ> :type Collection Nothing
Collection Nothing :: Collection Maybe a
```

```
λ> :type Collection (Just L.head)
Collection (Just L.head) :: Collection Maybe ([a] -> a)
```

Interlude - Phantom types

What's the type of a type?

Interlude - Phantom types

Functors

Applicative functors

Using newtype to make type class instances

Phantom Types

- **Phantom types** are a way to add extra information to types, eg. to differentiate them, in such a way so that the extra information goes away when type-checking is complete.
- We might want to avoid the Mars Climate Orbiter disaster.

*... software that calculated the total impulse produced by thruster firings produced results in **pound-force seconds**. The trajectory calculation software then used these results - expected to be in **newton-seconds** (incorrect by a factor of 4.45) - to update the predicted position of the spacecraft ...*

Type-level programming

We want to distinguish Fahrenheit and Celsius degrees but don't want to define separate types to store and process the corresponding temperatures.

Type-level programming

We want to distinguish Fahrenheit and Celsius degrees but don't want to define separate types to store and process the corresponding temperatures.

```
type Temp = Double
paperBurning :: Temp
paperBurning = 451      -- Fahrenheit
absoluteZero :: Temp
absoluteZero = -273.15 -- Celcius
```

Type-level programming

We want to distinguish Fahrenheit and Celsius degrees but don't want to define separate types to store and process the corresponding temperatures.

```
λ> paperBurning
```

```
451.0
```

```
λ> absoluteZero
```

```
-273.15
```

```
λ> :type (paperBurning, absoluteZero)
```

```
(paperBurning, absoluteZero) :: (Temp, Temp)
```

```
λ> paperBurning + absoluteZero -- oups !
```

```
177.85000000000002
```

```
λ> :type paperBurning + absoluteZero
```

```
paperBurning + absoluteZero :: Temp
```

Type-level programming

```
-- Fahrenheit degrees
data F

-- Celcius degrees
data C

-- Temperature
newtype Temp u = Temp {getTemp :: Double}
                  deriving (Eq, Ord)
```

Note that the `u` (unit) type variable is not used in the right-hand side : it is a **phantom parameter**.

Type-level programming

```
fahrenheitCharUnit :: String
fahrenheitCharUnit = "F"

celciusCharUnit :: String
celciusCharUnit = "C"

instance Show (Temp F) where
    show = flip (++) fahrenheitCharUnit . show . getTemp

instance Show (Temp C) where
    show = flip (++) celciusCharUnit . show . getTemp

paperBurning :: Temp F
paperBurning = Temp {getTemp = 451}

absoluteZero :: Temp C
absoluteZero = Temp {getTemp = -273.15}
```

Type-level programming

```
λ> paperBurning
```

```
451.0F
```

```
λ> :type paperBurning
```

```
paperBurning :: Temp F
```

```
λ> absoluteZero
```

```
-273.15C
```

```
λ> :type absoluteZero
```

```
absoluteZero :: Temp C
```

```
λ> paperBurning + absoluteZero
```

```
<interactive>: error:
```

```
    Couldn't match type 'C' with 'F'
```

```
      Expected: Temp F
```

```
      Actual: Temp C
```

Crazy types

Unfortunately, we don't have much control over the type we use instead of `u` (unit), besides which, it should be of kind `Type`.

```
crazyTemp :: Temp Bool
crazyTemp = Temp {getTemp = 0}

soCrazyTemp :: Temp (Temp C)
soCrazyTemp = Temp {getTemp = 0}
```

Proxies

- We want to have a value of a type, but the sole purpose of that value is to refer to a type.
- The value itself is never used.
- Such types are called **proxies**.

```
data Proxy a = Proxy
```

- The type `a` is a **phantom**.
- The only value of this type is `Proxy`.
- Available in module `Data.Proxy`.

Proxies

```
data Proxy a = Proxy

class UnitName u where
  unitName :: Proxy u -> String

instance UnitName F where
  unitName :: Proxy F -> String
  unitName _ = "F"

instance UnitName C where
  unitName :: Proxy C -> String
  unitName _ = "C"

instance UnitName u => UnitName (Temp u) where
  unitName _ = unitName (Proxy :: Proxy u)
```

Proxies

```
newtype Temp u = Temp {getTemp :: Double}

instance UnitName u => Show (Temp u) where
  show = flip (++) symb . show . getTemp
  where
    symb = unitName (Proxy :: Proxy u)
```

```
λ> paperBurning
451.0F
```

```
λ> absoluteZero
-273.15C
```

The whole story

```
-- Fahrenheit degrees
data F

-- Celcius degrees
data C

newtype Temp u = Temp {getTemp :: Double}
```

The whole story

```
data Proxy a = Proxy

class UnitName u where
  unitName :: Proxy u -> String

instance UnitName F where
  unitName :: Proxy F -> String
  unitName _ = "F"

instance UnitName C where
  unitName :: Proxy C -> String
  unitName _ = "C"

instance UnitName u => UnitName (Temp u) where
  unitName _ = unitName (Proxy :: Proxy u)
```

The whole story

```
instance UnitName u => Show (Temp u) where
  show = flip (++) symb . show . getTemp
  where
    symb = unitName (Proxy :: Proxy u)
```

```
λ> Temp {getTemp = 0} :: Temp C
0.0C
```

```
λ> Temp {getTemp = 0} :: Temp F
0.0F
```

The whole story

```
celciusToFahrenheit :: Temp C -> Temp F
celciusToFahrenheit ct = Temp {getTemp = f}
  where
    c = getTemp ct
    f = (c*9/5) + 32

fahrenheitToCelsius :: Temp F -> Temp C
fahrenheitToCelsius ft = Temp {getTemp = c}
  where
    f = getTemp ft
    c = (f-32) * 5/9
```

The whole story

```
λ> celciusToFahrenheit (Temp {getTemp = 0} :: Temp C)  
32.0F
```

```
λ> fahrenheitToCelsius (Temp {getTemp = 32} :: Temp F)  
0.0C
```

The whole story

```
mkCelcius :: Double -> Temp C
```

```
mkCelcius = Temp
```

```
mkFahrenheit :: Double -> Temp F
```

```
mkFahrenheit = Temp
```

```
λ> celciusToFahrenheit (mkCelcius 0)  
32.0F
```

```
λ> fahrenheitToCelsius (mkFahrenheit 32)  
0.0C
```

Functors

What's the type of a type?

Interlude - Phantom types

Functors

Applicative functors

Using newtype to make type class instances

- A **Functor** is any type that can act as a **generic container**.
- A Functor allows us to transform the underlying values with a function, so that the values are all updated, but the structure of the container is the same.
- Haskell represents the concept of a functor with the **Functor** typeclass. This typeclass has a single required function **fmap**.

Functor

```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
```

- Functor in Haskell is a **typeclass** that provides two methods : **fmap** and **(<\$)**.
- To implement a Functor instance for a data type, you need to provide a type-specific implementation of **fmap**.
- **fmap** is a higher ordered function taking two inputs:
 - (i) a **transformation** function from an **a** type to a **b** type, and
 - (ii) a functor containing values of type **a**.

Kind signature of Functor

```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
```

- The kind of **Functor** is $(* \rightarrow *) \rightarrow \text{Constraint}$, which means that we can implement **Functor** for types whose kind is $* \rightarrow *$.
- In other words, we can implement **Functor** for types that have one unapplied type variable.

Kind signature of Functor

```
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
```

- (<\$) replaces all locations in the input with the same value.
The default definition is `fmap . const`, but this may be overridden with a more efficient version.

```
x <$ y
= { definition of <$                                }
  (fmap . const) x y
= { definition of function composition }
  fmap (const x) y
```

The big picture

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

```
class Functor f where
```

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

Functor laws

In addition to providing a function `fmap` of the specified type, functors are also required to satisfy two equational laws.

- Identity

Applying `id` function to the wrapped value changes nothing:

```
fmap id == id
```

- Composition

Applying `fmap` sequentially is the same as applying `fmap` with the composition of functions:

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

Maybe is a functor

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just  (f x)
```

Maybe is a functor

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just  (f x)
```

```
λ> fmap (+1) Nothing
Nothing
```

```
λ> fmap (+1) (Just 2)
Just 3
```

```
λ> fmap ((*3) . (+1)) Nothing
Nothing
```

```
λ> fmap ((*3) . (+1)) (Just 2)
Just 9
```

[] is a functor

```
instance Functor [] where
  fmap _ []      = []
  fmap f (x:xs) = f x:fmap f xs
```

[] is a functor

```
instance Functor [] where  
  fmap = map -- Data.List.map
```

[] is a functor

```
instance Functor [] where
```

```
  fmap f = map
```

```
λ> fmap (+1) []  
[]
```

```
λ> fmap (+1) [1,2,3]  
[2,3,4]
```

```
λ> fmap Just [1,2,3]  
[Just 1,Just 2,Just 3]
```

```
λ> fmap (not . even) [1,2,3,4,5,6]  
[True,False,True,False,True,False]
```

```
λ> (fmap not . fmap even) [1,2,3,4,5,6]  
[True,False,True,False,True,False]
```

[] is a functor

```
instance Functor [] where
```

```
  fmap = map -- Data.List.map
```

```
λ> xs = [1,2,3]
```

```
λ> fmap Just xs
```

```
[Just 1,Just 2,Just 3]
```

```
λ> L.intersperse Nothing . fmap Just $ xs
```

```
[Just 1,Nothing,Just 2,Nothing,Just 3]
```

```
λ> fmap (fmap (+1)) . L.intersperse Nothing . fmap Just $ xs
```

```
[Just 2,Nothing,Just 3,Nothing,Just 4]
```

```
λ> [y | y <- [Just 2,Nothing,Just 3,Nothing,Just 4]]
```

```
[Just 2,Nothing,Just 3,Nothing,Just 4]
```

```
λ> [y | Just y <- [Just 2,Nothing,Just 3,Nothing,Just 4]]
```

```
[2,3,4]
```

Either is a functor ... but wait !

```
λ> :kind Either
```

```
Either :: * -> * -> *
```

```
λ> :kind Functor
```

```
Functor :: (* -> *) -> Constraint
```

Either is a functor ... but wait !

```
λ> :kind Either
```

```
Either :: * -> * -> *
```

```
λ> :kind Functor
```

```
Functor :: (* -> *) -> Constraint
```

```
λ> :kind Either Int
```

```
Either Int :: * -> *
```

```
λ> :kind Either Bool
```

```
Either Bool :: * -> *
```

```
λ> :kind Either (Maybe Bool)
```

```
Either (Maybe Bool) :: * -> *
```

```
λ> :kind Either (Either Int String)
```

```
Either (Either Int String) :: * -> *
```

Either is a functor

```
instance Functor (Either a) where
  fmap f (Left x)  = Left x
  fmap f (Right x) = Right (f x)
```

Either is a functor

```
instance Functor (Either a) where
```

```
  fmap f (Left x)  = Left x
```

```
  fmap f (Right x) = Right (f x)
```

- `Either` has kind `* -> * -> *`, so we can't write

```
instance Functor Either where.
```

- If we write `instance Functor (Either a) where`, the function `fmap` has type :

```
fmap :: (b -> c) -> Either a b -> Either a c
```

Either is a functor

```
instance Functor (Either a) where
```

```
    fmap f (Left x)  = Left x
```

```
    fmap f (Right x) = Right (f x)
```

```
λ> :type fmap (++ "rock!") (Left 0)
```

```
fmap (++ "rock!") (Left 1) :: Num a => Either a [Char]
```

```
λ> fmap (++ "rock!") (Left 0)
```

```
Left 0
```

```
λ> :type fmap (++ "rock!") (Right "Haskell ")
```

```
fmap (++ "rock!") (Right "Haskell ") :: Either a [Char]
```

```
λ> fmap (++ "rock!") (Right "Haskell ")
```

```
Right "Haskell rock!"
```

Replicating

```
λ> fmap (L.replicate 3) []  
[]
```

```
λ> fmap (L.replicate 3) [1,2,3,4]  
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
```

```
λ> fmap (replicate 3) Nothing  
Nothing
```

```
λ> fmap (replicate 3) (Just 1)  
Just [1,1,1]
```

```
λ> fmap (replicate 3) (Left 1)  
Left 1
```

```
λ> fmap (replicate 3) (Right 1)  
Right [1,1,1]
```

Functions as functors

```
instance Functor ((->) a) where  
  fmap = (.)
```

Functions as functors

```
instance Functor ((->) a) where  
    fmap = (.)
```

```
λ> :type fmap (*3) (+100)  
fmap (*3) (+100) :: Num a => a -> a
```

```
λ> fmap (*3) (+100) 1  
303
```

```
λ> (*3) `fmap` (+100) $ 1  
303
```

```
λ> (*3) . (+100) $ 1  
303
```

Make your own functor

```
data Tree a = E | N (Tree a) a (Tree a)
             deriving Show

instance Functor Tree where
    fmap _ E           = E
    fmap f (N lt x rt) = N (fmap f lt) (f x) (fmap f rt)
```

Make your own functor

```
data Tree a = E | N (Tree a) a (Tree a)
              deriving Show

instance Functor Tree where
    fmap _ E           = E
    fmap f (N lt x rt) = N (fmap f lt) (f x) (fmap f rt)

mkL :: a -> Tree a
mkL x = N E x E

λ> fmap (*2) E
E

λ> fmap (*2) (mkL 1)
N E 2 E

λ> fmap (*2) (N (mkL 1) 2 (mkL 3))
N (N E 2 E) 4 (N E 6 E)
```

A pathological example

```
data CMaybe a = CNothing | CJust Int a deriving (Show)

instance Functor CMaybe where
    fmap f CNothing      = CNothing
    fmap f (CJust c x) = CJust (c+1) (f x)
```

A pathological example

```
data CMaybe a = CNothing | CJust Int a deriving (Show)

instance Functor CMaybe where
    fmap f CNothing      = CNothing
    fmap f (CJust c x) = CJust (c+1) (f x)
```

Does this obey the functor laws?

```
λ> fmap id (CJust 0 "Haskell")
CJust 1 "Haskell"
```

```
λ> id (CJust 0 "Haskell")
CJust 0 "Haskell"
```

Applicative functors

What's the type of a type?

Interlude - Phantom types

Functors

Applicative functors

Using newtype to make type class instances

Applicative

- **Applicative** is the class for **applicative functors**.
- **Applicative functors** are functors with extra laws and operations.
- **Applicative** is an intermediate class between **Functor** and **Monad**.
- **Applicative** enables the eponymous **applicative style** (*i.e.*, a convenient way of structuring functorial computations), and also provides means to express a number of important patterns.

Kind signature of Applicative

```
type Applicative :: (* -> *) -> Constraint
class Functor f => Applicative f where
  pure    :: a -> f a
  (<*>)   :: f (a -> b) -> f a -> f b
  liftA2  :: (a -> b -> c) -> f a -> f b -> f c
  (*>)    :: f a -> f b -> f b
  (<*)    :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) / liftA2) #-}
```

- The kind of **Applicative** is $(* \rightarrow *) \rightarrow \text{Constraint}$, which means that we can implement **Applicative** for types whose kind is $* \rightarrow *$.
- If we want to make a type constructor part of the **Applicative** typeclass, it has to be in **Functor** first.

Applicative

```
type Applicative :: (* -> *) -> Constraint
class Functor f => Applicative f where
  pure    :: a -> f a
  (<*>)   :: f (a -> b) -> f a -> f b
  liftA2  :: (a -> b -> c) -> f a -> f b -> f c
  (*>)    :: f a -> f b -> f b
  (<*)    :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) / liftA2) #-}
```

- To implement a Functor instance for a data type, you need to provide a type-specific implementations of `pure` and either a type-specific implementations of `(<*>)` or `liftA2`.

The big picture

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

```
class Functor f where
```

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

```
class (Functor f) => Applicative f where
```

```
    pure  :: a -> f a
```

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

Maybe is an applicative functor

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

Maybe is an applicative functor

```
instance Applicative Maybe where
```

```
  pure = Just
```

```
  Nothing  <*> _ = Nothing
```

```
  (Just f) <*> x = fmap f x
```

```
λ> :type pure 1
```

```
pure 1 :: (Applicative f, Num a) => f a
```

```
λ> pure 1 :: Maybe Int
```

```
Just 1
```

```
λ> :type pure "Haskell"
```

```
pure "Haskell" :: Applicative f => f String
```

```
λ> pure "Haskell" :: Maybe String
```

```
Just "Haskell"
```

Maybe is an applicative functor

```
instance Applicative Maybe where
```

```
  pure = Just
```

```
  Nothing  <*> _ = Nothing
```

```
  (Just f) <*> x = fmap f x
```

```
λ> Just (+3) <*> Just 9
```

```
Just 12
```

```
λ> pure (+3) <*> Just 9
```

```
Just 12
```

```
λ> fmap (+3) (Just 9)
```

```
Just 12
```

```
λ> pure (+3) <*> Nothing
```

```
Nothing
```

```
λ> Nothing <*> Just 9
```

```
Nothing
```

The applicative style

- **Applicative functors** and the **applicative style** of doing `pure f <*> x <*> y <*> ...` allow us to take a function that expects parameters that aren't necessarily wrapped in functors and use that function to operate on several values that are in functor contexts.
- The function can take as many parameters as we want, because it's always partially applied step by step between occurrences of `<*>`.

The applicative style

```
λ> pure (+) <*> Just 3 <*> Just 5  
Just 8
```

The applicative style

```
λ> pure (+) <*> Just 3 <*> Just 5
```

```
Just 8
```

```
    pure (+) <*> Just 3 <*> Just 5
```

```
= { <*> is left-associative }
```

```
    (pure (+) <*> Just 3) <*> Just 5
```

```
= { definition of pure      }
```

```
    (Just (+) <*> Just 3) <*> Just 5
```

```
= { definition of <*>      }
```

```
    fmap (+) (Just 3) <*> Just 5
```

```
= { definition of fmap     }
```

```
    Just (3+) <*> Just 5
```

```
= { definition of <*>      }
```

```
    fmap (3+) (Just 5)
```

```
= { definition of fmap     }
```

```
    Just 8
```

The applicative style

```
λ> pure (+) <*> Just 3 <*> Just 5  
Just 8
```

```
λ> Nothing <*> Just 3 <*> Just 5  
Nothing
```

```
λ> pure (+) <*> Nothing <*> Just 5  
Nothing
```

```
λ> pure (+) <*> Just 3 <*> Nothing  
Nothing
```

```
λ> pure (*) <*> (pure (+) <*> Just 1 <*> Just 2) <*> (Just 4)  
Just 12
```

The applicative style

```
λ> combine x y = pure (+) <*> Just x <*> Just y
λ> :type combine
combine :: Num a => a -> a -> Maybe a
λ> zipWith combine [0,4..20] [100..]
[Just 100,Just 105,Just 110,Just 115,Just 120,Just 125]
```

The applicative style

```
λ> fmap ((<*>) (pure (+1))) . Just [1,2,3,4]  
[Just 2,Just 3,Just 4,Just 5]
```

```
λ> [pure (+1) <*> Just n | n <- [1,2,3,4]]  
[Just 2,Just 3,Just 4,Just 5]
```

```
λ> foldr (\x -> (:)) (pure (+1) <*> Just x) [] [1,2,3,4]  
[Just 2,Just 3,Just 4,Just 5]
```

The applicative style

```
λ> Just (+) <*> Just 3 <*> Just 5  
Just 8
```

```
λ> Just (+) <*> pure 3 <*> pure 5  
Just 8
```

```
λ> pure (+) <*> Just 3 <*> pure 5  
Just 8
```

```
λ> pure (+) <*> pure 3 <*> Just 5  
Just 8
```

```
λ> pure (+) <*> pure 3 <*> pure 5  
8
```

The applicative style

`Control.Applicative` exports a function called `<$>`, which is just `fmap` as an infix operator :

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

The applicative style

`Control.Applicative` exports a function called `<$>`, which is just `fmap` as an infix operator :

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

```
λ> (+) <$> Just 3 <*> Just 5  
Just 8
```

```
λ> (+) 3 5  
8
```

The applicative style

```
(+) <$> Just 3 <*> Just 5
= { definition of <$> }
  fmap (+) (Just 3) <*> Just 5
= { definition of fmap }
  Just (3+) <*> Just 5
= { definition of <*> }
  fmap (3+) (Just 5)
= { definition of fmap }
  Just 8
```

The applicative style

```
(+) <$> Nothing <*> Just 5
= { definition of <$> }
  fmap (+) Nothing <*> Just 5
= { definition of fmap }
  Nothing <*> Just 5
= { definition of <*> }
  Nothing
```

The applicative style

```
(+) <$> Just 3 <*> Nothing
= { definition of <$> }
  fmap (+) (Just 3) <*> Nothing
= { definition of fmap }
  Just (3+) <*> Nothing
= { definition of <*> }
  fmap (3+) Nothing
= { definition of fmap }
  Nothing
```

[] is an applicative functor

```
instance Applicative [] where
  pure x      = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

[] is an applicative functor

```
instance Applicative [] where
```

```
  pure x      = [x]
```

```
  fs <*> xs = [f x | f <- fs, x <- xs]
```

```
λ> :type pure 1
```

```
pure 1 :: (Applicative f, Num a) => f a
```

```
λ> pure 1 :: [Int]
```

```
[1]
```

```
λ> :type pure "Haskell"
```

```
pure "Haskell" :: Applicative f => f String
```

```
λ> pure "Haskell" :: [String]
```

```
["Haskell"]
```

[] is an applicative functor

```
instance Applicative [] where
```

```
  pure x    = [x]
```

```
  fs <*> xs = [f x | f <- fs, x <- xs]
```

```
λ> [(+1),(*10)] <*> [1,2,3,4]
```

```
[2,3,4,5,10,20,30,40]
```

```
λ> [even,odd] <*> [1,2,3,4]
```

```
[False,True,False,True,True,False,True,False]
```

```
λ> [] <*> [1,2,3,4]
```

```
[]
```

```
λ> [(+1),(*10)] <*> []
```

```
[]
```

[] is an applicative functor

```
instance Applicative [] where
```

```
  pure x    = [x]
```

```
  fs <*> xs = [f x | f <- fs, x <- xs]
```

```
λ> (++) <$> ["A","B","C"] <*> ["x","y","z"]  
["Ax","Ay","Az","Bx","By","Bz","Cx","Cy","Cz"]
```

```
λ> [even,odd] <*> [1,2,3,4]  
[False,True,False,True,True,False,True,False]
```

```
λ> [] <*> [1,2,3,4]  
[]
```

```
λ> [(+1),(*10)] <*> []  
[]
```

[] is an applicative functor

Using the **applicative style** on lists is often a good replacement for list comprehensions:

[] is an applicative functor

Using the **applicative style** on lists is often a good replacement for list comprehensions:

```
λ> [x*y | x <- [2,5,10], y <- [8,10,11]]  
[16,20,22,40,50,55,80,100,110]
```

```
λ> (*) <$> [2,5,10] <*> [8,10,11]  
[16,20,22,40,50,55,80,100,110]
```

```
λ> binaries = [b:bs | bs <- "":binaries, b <- ['0','1']]  
λ> take 10 binaries  
["0","1","00","10","01","11","000","100","010","110"]
```

```
λ> binariesA = pure (flip (:)) <*> "":binariesA <*> ['0','1']  
λ> take 10 binariesA  
["0","1","00","10","01","11","000","100","010","110"]
```

Functions as applicative functors

```
instance Applicative ((->) r) where
    pure x  = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

Functions as applicative functors

```
instance Applicative ((->) r) where
```

```
  pure x  = (\_ -> x)
```

```
  f <*> g = \x -> f x (g x)
```

```
λ> (pure 1) "Haskell"
```

```
1
```

```
λ> pure 1 "Haskell" -- thanks to currying
```

```
1
```

```
  pure 1 "Haskell"
```

```
= { definition of pure }
```

```
  (\_ -> 1) "Haskell"
```

```
= { function application }
```

```
1
```

Functions as applicative functors

```
instance Applicative ((->) r) where
```

```
  pure x  = (\_ -> x)
```

```
  f <*> g = \x -> f x (g x)
```

```
λ> :type (+) <$> (+3) <*> (*100)
```

```
(+) <$> (+3) <*> (*100) :: Num a => a -> a
```

```
λ> (+) <$> (+3) <*> (*100) $ 5
```

```
508
```

Functions as applicative functors

```
(+) <$> (+3) <*> (*100) $ 5
= { definition of <$> : f <$> x = fmap f x           }
  fmap (+) (+3) <*> (*100) $ 5
= { definition of map : fmap f g = (\x -> f (g x)) }
  \x -> (+) (x+3) <*> (*100) $ 5
= { definition of <*> : f <*> g = \x -> f x (g x)   }
  \y -> (\x -> (+) (x+3)) y (y*100) $ 5
= { rewrite inner lambda                               }
  \y -> (+) (y+3) (y*100) $ 5
= { function application                               }
  (5+3) + (5*100)
= { arithmetics                                         }
  508
```

Functions as applicative functors

```
λ> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 8  
[11.0,16.0,4.0]
```

```
λ> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (`div` 2) $ 8  
[11,16,4]
```

```
λ> (\x y z -> [x,y,z]) <$>  
  (++) "." <*> (++) "!" <*> (++) "?" $ "A"  
["A.", "A!", "A?"]
```

```
λ> (\x y z -> [x,y,z]) <$>  
  (replicate 1) <*> (replicate 2) <*> (replicate 3) $ 'A'  
["A", "AA", "AAA"]
```

Functions as applicative functors

- We can think of functions as **boxes** that contain their eventual results, so doing

```
k <$> f <*> g
```

creates a function that will call **k** with the eventual results from **f** and **g**

- When we do something like

```
(+) <$> Just 3 <*> Just 5
```

we're using **+** on values that might or might not be there, which also results in a value that might or might not be there.

- When we do

```
(+) <$> (+10) <*> (+5)
```

we're using **+** on the future return values of **(+10)** and **(+5)** and the result is also something that will produce a value only when called with a parameter.

ZipList

There are actually more ways for lists to be applicative functors.

```
newtype ZipList a = ZipList { getZipList :: [a] }  
instance Functor ZipList where  
    fmap f (ZipList xs) = ZipList (map f xs)  
instance Applicative ZipList where  
    pure x                = ZipList (repeat x)  
    ZipList fs <*> ZipList xs = ZipList (zipWith id fs xs)
```

ZipList

```
λ> fmap (+1) . ZipList $ []  
ZipList {getZipList = []}  
  
λ> fmap (+1) . ZipList $ []  
ZipList {getZipList = []}  
  
λ> fmap (*3) . fmap (+1) . ZipList $ [1,2,3,4]  
ZipList {getZipList = [6,9,12,15]}
```

ZipList

```
λ> pure 'a' :: ZipList Char
aaaaaaaaaaaaaaaaaaaaa... ^C Interrupted.
λ> take 10 $ getZipList (pure 'a' :: ZipList Char)
"aaaaaaaaaa"
```

ZipList

```
λ> getZipList $ (+) <$> ZipList [1,2,3]
      <*> ZipList [100,100,100]
[101,102,103]

λ> getZipList $ (+) <$> ZipList [1,2,3]
      <*> ZipList [100,100..]
[101,102,103]

λ> getZipList $ max <$> ZipList [1,2,3,4,5,3]
      <*> ZipList [5,3,1,2]
[5,3,3,4]

λ> getZipList $ (,,) <$> ZipList "dog"
      <*> ZipList "cat"
      <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Hiding applicative functors

```
liftA2 :: (Applicative f) =>
        (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

Hiding applicative functors

```
liftA2 :: (Applicative f) =>
        (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

```
λ> liftA2 (:) (Just 1) (Just [2,3,4])
Just [1,2,3,4]
```

```
λ> (:) <$> Just 1 <*> Just [2,3,4]
Just [1,2,3,4]
```

```
λ> pure (:) <*> Just 1 <*> Just [2,3,4]
Just [1,2,3,4]
```

Hiding applicative functors

```
liftA2 :: (Applicative f) =>
        (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

```
λ> liftA2 (++) (Just [1,2,3,4]) (Just [5,6,7,8])
Just [1,2,3,4,5,6,7,8]
```

```
λ> (++) <$> Just [1,2,3,4] <*> Just [5,6,7,8]
```

```
λ> pure (++) <*> Just [1,2,3,4] <*> Just [5,6,7,8]
Just [1,2,3,4,5,6,7,8]
```

Combining applicative functors

`sequenceA` takes a list of applicatives and returns an applicative that has a list as its result value.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
```

```
sequenceA [] = pure []
```

```
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

Combining applicative functors

```
λ> sequenceA [Just 1, Just 2]  
Just [1,2]
```

Combining applicative functors

```
λ> sequenceA [Just 1, Just 2]  
Just [1,2]
```

```
sequenceA [Just 1, Just 2]  
= { definition of sequenceA          }  
  (:) <$> Just 1 <*> sequenceA [Just 2]  
= { definition of sequenceA          }  
  (:) <$> Just 1 (<*> Just 2 <*> sequenceA [])  
= { definition of sequenceA          }  
  (:) <$> Just 1 (<*> Just 2 <*> pure [])  
= { definition of pure                }  
  (:) <$> Just 1 <*> Just 2 <*> Just []  
= { definition of <$>, fmap and <*> }  
  Just [1,2]
```

Combining applicative functors

```
λ> sequenceA [Just 1,Just 2,Just 3,Just 4]  
Just [1,2,3,4]
```

```
λ> sequenceA [Just 1,Just 2,Nothing,Just 4]  
Nothing
```

```
λ> sequenceA [(+3),(+2),(+1)] 3  
[6,5,4]
```

```
λ> getZipList $ ZipList [(+3),(+2),(+1)] <*> pure 3  
[6,5,4]
```

```
λ> sequenceA [[1,2,3],[4,5,6]]  
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```

```
λ> sequenceA [[1,2,3],[], [4,5,6]]  
[]
```

Combining applicative functors

```
λ> map (\f -> f 7) [(>4),(<10),odd]  
[True,True,True]
```

```
λ> map ($ 7) [(>4),(<10),odd]  
[True,True,True]
```

```
λ> and $ map (\f -> f 7) [(>4),(<10),odd]  
True
```

Combining applicative functors

```
λ> sequenceA [[1,2,3],[4,5,6]]  
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```

```
λ> [[x,y] | x <- [1,2,3], y <- [4,5,6]]  
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```

```
λ> sequenceA [[1,2],[3,4]]  
[[1,3],[1,4],[2,3],[2,4]]
```

```
λ> [[x,y] | x <- [1,2], y <- [3,4]]  
[[1,3],[1,4],[2,3],[2,4]]
```

```
λ> sequenceA [[1,2],[3,4],[5,6]]  
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],  
 [2,4,5],[2,4,6]]
```

```
λ> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]  
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],  
 [2,4,5],[2,4,6]]
```

Combining applicative functors

```
sequenceA [[1,2],[3,4]]
= { definition of sequenceA          }
  (:) <$> [1,2] <*> sequenceA [[3,4]]
= { definition of sequenceA          }
  (:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [])
= { definition of sequenceA          }
  (:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])
= { definition of <$>, fmap and <*> }
  (:) <$> [1,2] <*> [[3],[4]]
= { definition of <$>                }
  fmap (:) [1,2] <*> [[3],[4]]
= { definition of fmap and <*>       }
  [[1,3],[1,4],[2,3],[2,4]]
```

Using newtype to make type class instances

What's the type of a type?

Interlude - Phantom types

Functors

Applicative functors

Using newtype to make type class instances

Making type class instances

- Many times, we want to make our types instances of certain type classes, but the type parameters just don't match up for what we want to do.
- Use `newtype` to get around limitations.

Making type class instances

```
newtype Pair b a = Pair { getPair :: (a,b) }
```

We can make it an instance of `Functor` so that the function is mapped over the first component:

```
instance Functor (Pair c) where  
    fmap f (Pair (x,y)) = Pair (f x, y)
```

Making type class instances

```
newtype Pair b a = Pair { getPair :: (a,b) }
```

We can make it an instance of `Functor` so that the function is mapped over the first component:

```
instance Functor (Pair c) where  
    fmap f (Pair (x,y)) = Pair (f x, y)
```

```
λ> getPair $ fmap (*100) (Pair (2,3))  
(200,3)
```

```
λ> getPair $ fmap reverse (Pair ("london calling", 3))  
("gnillac nodnol",3)
```

On newtype laziness

- `newtype` is usually faster than `data`.
- The only thing that can be done with `newtype` is turning an existing type into a new type, so internally, Haskell can represent the values of types defined with `newtype` just like the original ones, only it has to keep in mind that the their types are now distinct.
- This fact means that not only is `newtype` faster, it's also lazier.

On newtype laziness

```
data T = T { getInner :: Bool }
```

```
sayHello :: T -> String
```

```
sayHello (T _) = "hello!"
```

On newtype laziness

```
data T = T { getInner :: Bool }
```

```
sayHello :: T -> String
```

```
sayHello (T _) = "hello!"
```

```
λ> sayHello undefined
```

```
*** Exception: Prelude.undefined
```

```
...
```

On newtype laziness

```
data T = T { getInner :: Bool }
```

```
sayHello :: T -> String
```

```
sayHello (T _) = "hello!"
```

- Types defined with the `data` keyword can have multiple value constructors (even though `T` only has one).
- So in order to see if the value given to our function conforms to the `(T _)` pattern, Haskell has to evaluate the value just enough to see which value constructor was used when we made the value.

On newtype laziness

```
newtype T = T { getInner :: Bool }  
  
sayHello :: T -> String  
sayHello (T _) = "hello!"
```

On newtype laziness

```
newtype T = T { getInner :: Bool }
```

```
sayHello :: T -> String
```

```
sayHello (T _) = "hello!"
```

```
λ> sayHello undefined
```

```
"hello!"
```

On newtype laziness

```
newtype T = T { getInner :: Bool }
```

```
sayHello :: T -> String
```

```
sayHello (T _) = "hello!"
```

- when we use `newtype`, Haskell can internally represent the values of the new type in the same way as the original values.
- It doesn't have to add another box around them, it just has to be aware of the values being of different types.
- And because Haskell knows that types made with the `newtype` keyword can only have one constructor, it doesn't have to evaluate the value passed to the function to make sure that it conforms to the `(T _)` pattern because `newtype` types can only have one possible value constructor and one field!