Functional Programming

Lecture 11: Functors, Applicatives & Monads

Twan van Laarhoven

28 November 2022

Outline

- Functors
- Applicative functors
- Monads
- Example: making code nicer
- Summary

Functors

Containers

What is a container?

- A container (in some way) holds some number of ‘values’
- A container can contain values of any type
- What operations for all containers?

Mapping functions

List is the prime example of a container type.

Recall: map applies a given function to each element of a list

$$
\begin{aligned}
&\mathsf{map} :: (a \to b) \to ([a] \to [b])\\
&\mathsf{map}\ f\ [\,] \qquad\quad = [\,]\\
&\mathsf{map}\ f\ (x : xs) = f\ x : \mathsf{map}\ f\ xs
\end{aligned}
$$

map changes the elements but keeps the structure intact

Mapping functions

Maybe is also a container type

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

Either an empty or a singleton container

Maybe also has a mapping function

```
mapMaybe :: (a → b) → (Maybe a → Maybe b)
mapMaybe f Nothing = Nothing
mapMaybe f (Just a) = Just (f a)
```

Mapping functions (continued)

Map on binary trees

```
data Btree a = Tip a | Bin (Btree a) (Btree a)

mapBtree :: (a → b) → (Btree a → Btree b)
mapBtree f (Tip a) = Tip (f a)
mapBtree f (Bin t u) = Bin (mapBtree f t) (mapBtree f u)
```

Mapping functions (continued)

Map on binary trees

```
data Btree a = Tip a | Bin (Btree a) (Btree a)
mapBtree :: (a → b) → (Btree a → Btree b)
mapBtree f (Tip a) = Tip (f a)
mapBtree f (Bin t u) = Bin (mapBtree f t) (mapBtree f u)
```

Map on general trees

```
data Gtree a = Branch a [Gtree a]
mapGtree :: (a → b) → (Gtree a → Gtree b)
mapGtree f (Branch x ts) = Branch (f x) (map (mapGtree f) ts)
```

The Functor type class

The types of these mapping functions are very similar:

```
map      :: (a → b) → ([a] → [b])
mapMaybe :: (a → b) → (Maybe a → Maybe b)
mapBtree :: (a → b) → (Btree a → Btree b)
mapGtree :: (a → b) → (Gtree a → Gtree b)
```

The Functor type class

The types of these mapping functions are very similar:

```
map      :: (a → b) → ([a] → [b])
mapMaybe :: (a → b) → (Maybe a → Maybe b)
mapBtree :: (a → b) → (Btree a → Btree b)
mapGtree :: (a → b) → (Gtree a → Gtree b)
```

The Functor class abstracts away from the container type

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

Note that f is a type constructor, not a type!
Functor is a so-called constructor class

The Functor type class

The types of these mapping functions are very similar:

```
map       :: (a → b) → ([a] → [b])
mapMaybe  :: (a → b) → (Maybe a → Maybe b)
mapBtree  :: (a → b) → (Btree a → Btree b)
mapGtree  :: (a → b) → (Gtree a → Gtree b)
```

The Functor class abstracts away from the container type

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

An infix synonym for fmap

```
(<$>) :: Functor f ⇒ :: (a → b) → (f a → f b)
(<$>) = fmap
```

Instances of the functor class

Every container type should be made an instance of the functor class

```
instance Functor [] where
  fmap = map
instance Functor Maybe where
  fmap = mapMaybe
instance Functor Btree where
  fmap = mapBtree
instance Functor Gtree where
  fmap = mapGtree
```

Instances of the functor class

Every container type should be made an instance of the functor class

```
instance Functor [] where
  fmap = map
instance Functor Maybe where
  fmap = mapMaybe
instance Functor Btree where
  fmap = mapBtree
instance Functor Gtree where
  fmap = mapGtree
instance Functor IO where
  fmap = liftM
```

Instances of the functor class

Every container type should be made an instance of the functor class

```
instance Functor [] where
  fmap = map
instance Functor Maybe where
  fmap = mapMaybe
instance Functor Btree where
  fmap = mapBtree
instance Functor Gtree where
  fmap = mapGtree
instance Functor IO where
  fmap f act = do { x ← act; pure (f x) }
```

Applicative functors

Functor with multiple arguments

fmap applies a function to a *container* of arguments.
What if we have a function with multiple arguments?
Idea: generalize fmap

$$
\begin{aligned}
\mathrm{fmap}_0 &:: a \to f\ a \\
\mathrm{fmap}_1 &:: (a \to b) \to f\ a \to f\ b \\
\mathrm{fmap}_2 &:: (a \to b \to c) \to f\ a \to f\ b \to f\ c \\
\mathrm{fmap}_3 &:: (a \to b \to c \to d) \to f\ a \to f\ b \to f\ c \to f\ d
\end{aligned}
$$

for example

```
>>> fmap₂ (+) (Just 1) (Just 2)
Just 3
```

We could introduce a class $\mathrm{Functor}_n$ for each fmap_n...

pure and apply

Introduce

```
infixl 4 <*>
pure  :: a → f a
(<*>) :: f (a → b) → f a → f b
```

where

- pure puts a value into a *container* of type f a
- <*> is generalized function application: it applies a *container* of functions to a *container* of arguments, producing a container of results.

```
infixl 4 <*>
pure  :: a → f a
(<*>) :: f (a → b) → f a → f b
```

Making fmap_n

we can now define

```
fmap0 :: a → f a
fmap0 = pure

fmap1 :: (a → b) → f a → f b
fmap1 g x = pure g <*> x

fmap2 :: (a → b → c) → f a → f b → f c
fmap2 g x y = pure g <*> x <*> y

fmap3 :: (a → b → c → d) → f a → f b → f c → f d
fmap3 g x y z = pure g <*> x <*> y <*> z
```

Making fmap_n

```haskell
infixl 4 <*>
pure  :: a → f a
(<*>) :: f (a → b) → f a → f b
```

we can now define

```haskell
fmap0 :: a → f a
fmap0 = pure

fmap1 :: (a → b) → f a → f b
fmap1 g x = pure g <*> x

fmap2 :: (a → b → c) → f a → f b → f c
fmap2 g x y = (pure g <*> x) <*> y

fmap3 :: (a → b → c → d) → f a → f b → f c → f d
fmap3 g x y z = ((pure g <*> x) <*> y) <*> z
```

Applicative class

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

Should agree with Functor instance:

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

Combining Functor and Applicative notation

If the first argument to $<\!\!*\!\!>$ is pure, you can use fmap

$$(<\$>) :: \text{Functor } f \Rightarrow (a \to b) \to (f\ a \to f\ b)$$
$$(<\$>) = \text{fmap}$$

$$\text{fmap}_2 :: (a \to b \to c) \to f\ a \to f\ b \to f\ c$$
$$\text{fmap}_2\ g\ x\ y = g <\$> x <\!\!*\!\!> y$$

$$\text{fmap}_3 :: (a \to b \to c \to d) \to f\ a \to f\ b \to f\ c \to f\ d$$
$$\text{fmap}_3\ g\ x\ y\ z = g <\$> x <\!\!*\!\!> y <\!\!*\!\!> z$$

Combining Functor and Applicative notation

If the first argument to $<\!*\!>$ is pure, you can use fmap

$$(<\$>) :: \text{Functor } f \Rightarrow (a \to b) \to (f\ a \to f\ b)$$
$$(<\$>) = \text{fmap}$$

$$\text{fmap}_2 :: (a \to b \to c) \to f\ a \to f\ b \to f\ c$$
$$\text{fmap}_2\ g\ x\ y = g <\$> x <\!*\!> y$$

$$\text{fmap}_3 :: (a \to b \to c \to d) \to f\ a \to f\ b \to f\ c \to f\ d$$
$$\text{fmap}_3\ g\ x\ y\ z = g <\$> x <\!*\!> y <\!*\!> z$$

Note: Actually these are called `liftA2`, `liftA3`, etc.

Maybe instance

```haskell
instance Applicative Maybe where
  pure :: a → Maybe a
  pure = Just

  (<*>) :: Maybe (a → b) → Maybe a → Maybe b
  Nothing  <*> _  = Nothing
  (Just g) <*> my = fmap g my
```

Examples

```haskell
>>> pure (+1) <*> (Just 1)
Just 2
>>> pure (+) <*> (Just 1) <*> (Just 2)
Just 3
>>> pure (+) <*> Nothing <*> (Just 2)
Nothing
```

Maybe instance

```
instance Applicative Maybe where
  pure :: a → Maybe a
  pure = Just
  (<*>) :: Maybe (a → b) → Maybe a → Maybe b
  Nothing  <*> _  = Nothing
  (Just g) <*> my = fmap g my
```

Exceptional programming:

applying pure functions to arguments that may fail without managing the propagation of failure explicitly

List instance

The standard prelude contains the following instance

```
instance Applicative [] where
  pure :: a → [a]
  pure x = [x]
  (<*>) :: [a → b] → [a] → [b]
  gs <*> xs = [g x | g ← gs, x ← xs]
```

pure transforms a value into a singleton list;
<*> takes a list of functions and a list of arguments and applies each function to each argument

List instance

The standard prelude contains the following instance

```
instance Applicative [] where
  pure :: a → [a]
  pure x = [x]
  (<*>) :: [a → b] → [a] → [b]
  gs <*> xs = [g x | g ← gs, x ← xs]
```

View [a] as a generalisation of Maybe a:

- empty list denotes no success
- non-empty list represents *all possible ways* a result may succeed

Hence applicative style for lists supports non-deterministic programming.

List instance

The standard prelude contains the following instance

```haskell
instance Applicative [] where
  pure :: a → [a]
  pure x = [x]
  (<*>) :: [a → b] → [a] → [b]
  gs <*> xs = [g x | g ← gs, x ← xs]
```

Examples:

```haskell
>>> (+) <$> [1,2] <*> [10,100]
[11,101,12,102]
>>> pure (++) <*> subsequences "hi" <*> pure " world"
[" world","h world","i world","hi world"]
```

IO instance

IO type can be made into an applicative functor using the following declaration:

```
instance Applicative IO where
  pure :: a → IO a
  pure = return
  (<*>) :: IO (a → b) → IO a → IO b
  act_g <*> act_x = do {g ← act_g; x ← act_x; return (g x)}
```

Example: reading n characters from the keyboard

```
getChars :: Int → IO String
getChars 0 = pure []
getChars n = pure (:) <*> getChar <*> getChars (n−1)
```

Derived operators

There are also one-sided versions of $<\!\!*\!\!>$
useful if a computation is only executed for its effect

$$(*\!\!>) :: \text{Applicative } f \Rightarrow f\ a \to f\ b \to f\ b$$
$$a *\!\!> b = \text{pure } (\backslash\_\ y \to y) <\!\!*\!\!> a <\!\!*\!\!> b$$

$$(<\!\!*) :: \text{Applicative } f \Rightarrow f\ a \to f\ b \to f\ a$$
$$a <\!\!* b = \text{pure } (\backslash x\ \_ \to x) <\!\!*\!\!> a <\!\!*\!\!> b$$

Compare $(\gg) :: \text{IO } a \to \text{IO } b \to \text{IO } b$

Example: evaluator

Expressions

Recall the datatype of expressions

```
data Expr
  = Lit Integer      -- a literal
  | Add Expr Expr    -- addition
  | Mul Expr Expr    -- multiplication
  | Div Expr Expr    -- integer division
```

Small extension: integer division

```
good, bad :: Expr
good  = Div (Lit 7) (Div (Lit 4) (Lit 2))
bad   = Div (Lit 7) (Div (Lit 2) (Lit 4))
```

The vanilla evaluator

Recall the evaluation function

```
eval :: Expr → Integer
eval (Lit i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
```

example evaluations:

```
>>> eval good
3
>>> eval bad
*** Exception: divide by zero
```

Handling failure

Evaluation may fail, because of division by zero
Let's handle the exceptional behaviour:

```
eval :: Expr → Maybe Integer
eval (Lit i)      = Just i
eval (Add e₁ e₂) =
  case eval e₁ of
    Nothing → Nothing
    Just v₁ → case eval e₂ of
      Nothing → Nothing
      Just v₂ → Just (v₁ + v₂)
```

(other cases omitted for reasons of space)

Handling failure

Evaluation may fail, because of division by zero

Let's handle the exceptional behaviour:

```
eval :: Expr → Maybe Integer
eval (Lit i)      = Just i
eval (Div e1 e2) =
  case eval e1 of
    Nothing → Nothing
    Just v1 → case eval e2 of
      Nothing → Nothing
      Just v2 | v2 == 0   → Nothing
              | otherwise → Just (v1 `div` v2)
```

(other cases omitted for reasons of space)

Evaluator, applicative style

Initial evaluator in an applicative style

```
eval :: (Applicative f) ⇒ Expr → f Integer
eval (Lit i)      = pure i
eval (Add e1 e2) = pure (+) <*> eval e1 <*> eval e2
eval (Mul e1 e2) = pure (*) <*> eval e1 <*> eval e2
eval (Div e1 e2) = pure div <*> eval e1 <*> eval e2
```

two changes compared to the vanilla evaluator

- prefix: `(+) a b` instead of `a + b`
- application made explicit: `pure f <*> a <*> b` instead of `f a b`

still pure, but much easier to extend

Handling failure, applicative style?

Let's check for division by 0 again:

```
eval :: Expr → Maybe Integer
eval (Lit i)       = pure i
eval (Add e₁ e₂) = pure (+) <*> eval e₁ <*> eval e₂
eval (Mul e₁ e₂) = pure (*) <*> eval e₁ <*> eval e₂
eval (Div e₁ e₂) = case eval e₂ of
  Just 0 → Nothing
  v₂     → pure div <*> eval e₁ <*> v₂
```

Cleaned up most cases, except Div

Monads

Handling failure, abstracted

Suppose div raised an exception if its second argument is zero

```
safediv :: Integer → Integer → Maybe Integer
safediv _ 0 = Nothing
safediv x y = Just (x `div` y)
```

We cannot use

```
pure safediv <*> eval e1 <*> eval e2
```

Because safediv has type Integer → Integer → Maybe Integer, instead of Integer → Integer → Integer.
The arguments of <*> are independent. The 'shape' of the output can not depend on values in the container.

Bind operator

Pattern: return Nothing if argument is Nothing, otherwise apply a function.

```
(>>=) :: Maybe a → (a → Maybe b) → Maybe b
mx >>= f = case mx of
  Nothing → Nothing
  Just x  → f x
```

Bind operator

Pattern: return Nothing if argument is Nothing, otherwise apply a function.

```
(>>=) :: Maybe a → (a → Maybe b) → Maybe b
mx >>= f = case mx of
  Nothing → Nothing
  Just x  → f x
```

We can now complete the evaluator

```
eval :: Expr → Maybe Integer
...
eval (Div e1 e2) =
  eval e1 >>= \x1 →
  eval e2 >>= \x2 →
  safediv x1 x2
```

(other cases omitted for reasons of space)

Monad class

This works for other type constructors as well, via

```haskell
class (Applicative m) => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

  return = pure
```

$\gg=$ (pronounced as "bind") allows you to generate an impure computation based on the value of another impure computation.
The second computation can depend on the result of the first.

`return` is just another name for `pure` (for historical reasons).

Have you seen these combinators before?

do notation

Haskell provides a special notation for monadic expressions

```
do
  x₁ ← m₁                              m₁ >>= \x₁ →
  x₂ ← m₂                 =            m₂ >>= \x₂ →
    ...                                   ...
  x_n ← m_n                            m_n >>= \x_n →
  f x₁ x₂ ... x_n                      f x₁ x₂ ... x_n
```

Note: do is layout sensitive.

See also lecture 10

do notation (continued)

We can ignore the result with

$$(\gg) \;::\; \text{Monad } m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b$$
$$a \gg b = a \gg\!= \backslash\_ \rightarrow b$$

Make local declarations with **let** statements

$$\begin{array}{l} \textbf{do} \\ \quad x_1 \leftarrow m_1 \\ \quad \textbf{let}\ x_2 = \text{nonMonadicCode}\ x_1 \\ \quad x_3 \leftarrow m_3 \\ \quad m_4 \\ \quad f\ x_1\ x_2\ x_3 \end{array} \quad = \quad \begin{array}{l} m_1 \gg\!= \backslash x_1 \rightarrow \\ \textbf{let}\ x_2 = \text{nonMonadicCode}\ x_1\ \textbf{in} \\ m_3 \gg\!= \backslash x_3 \rightarrow \\ m_4 \gg \\ f\ x_1\ x_2\ x_3 \end{array}$$

Maybe instance

```haskell
instance Monad Maybe where
  return :: a → Maybe a
  return x = Just x
  (>>=) :: Maybe a → (a → Maybe b) → Maybe b
  Nothing >>= _ = Nothing
  Just x  >>= k = k x
```

Example:

```haskell
>>> Just 10 >>= (`safediv` 5)
Just 2
```

List instance

```
instance Monad [] where
  return :: a → [a]
  return x = [x]
  (>>=) :: [a] → (a → [b]) → [b]
  xs >>= k = [ v | x ← xs, v ← k x ]
```

Example:

```
>>> do { x ← [1..5]; y ← [0,10]; pure (x + y) }
[1,11,2,12,3,13,4,14,5,15]
```

List instance

```
instance Monad [] where
  return :: a → [a]
  return x = [x]
  (>>=) :: [a] → (a → [b]) → [b]
  xs >>= k = [ v | x ← xs, v ← k x ]
```

Example:

```
>>> do { x ← [1..5]; y ← [0,10]; pure (x + y) }
[1,11,2,12,3,13,4,14,5,15]

>>> [ x + y | x ← [1..5], y ← [0,10] ]
[1,11,2,12,3,13,4,14,5,15]
```

Exception handling evaluator

Evaluator using the Monad class

```
eval :: Expr → Maybe Integer
eval (Lit i)       = pure i
eval (Add e₁ e₂) = (+) <$> eval e₁ <*> eval e₂
eval (Mul e₁ e₂) = (*) <$> eval e₁ <*> eval e₂
eval (Div e₁ e₂) = do
  v₁ ← eval e₁
  v₂ ← eval e₂
  safediv v₁ v₂
```

Exception handling evaluator

Evaluator using the Monad class

```
eval :: Expr → Maybe Integer
eval (Lit i)       = pure i
eval (Add e₁ e₂) = liftA2 (+) (eval e₁) (eval e₂)
eval (Mul e₁ e₂) = do
  v₁ ← eval e₁
  v₂ ← eval e₂
  pure (v₁ * v₂)
eval (Div e₁ e₂) = do
  v₁ ← eval e₁
  v₂ ← eval e₂
  safediv v₁ v₂
```

Have we been here before?

```
mapM       :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_      :: Monad m => (a -> m b) -> [a] -> m ()
foldM      :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
filterM    :: Monad m => (a -> m Bool) -> [a] -> m [a]
replicateM :: Monad m => Int -> m a -> m [a]
```

Have we been here before?

```
mapM  :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM_ :: Monad m ⇒ (a → m b) → [a] → m ()
foldM :: Monad m ⇒ (b → a → m b) → b → [a] → m b
filterM :: Monad m ⇒ (a → m Bool) → [a] → m [a]

replicateM :: Monad m ⇒ Int → m a → m [a]
replicateM 0 m = return []
replicateM n m = do
  x ← m
  xs ← replicateM (n−1) m
  pure (x : xs)
```

Have we been here before?

```
mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]

replicateM :: Monad m => Int -> m a -> m [a]
replicateM 0 m = pure []
replicateM n m = (:) <$> m <*> replicateM (n-1) m
```

Have we been here before?

```
mapM  :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM_ :: Monad m ⇒ (a → m b) → [a] → m ()
foldM :: Monad m ⇒ (b → a → m b) → b → [a] → m b
filterM :: Monad m ⇒ (a → m Bool) → [a] → m [a]

replicateM :: Monad m ⇒ Int → m a → m [a]
replicateM 0 m = pure []
replicateM n m = (:) <$> m <*> replicateM (n−1) m

join :: Monad m ⇒ m (m a) → m a
join mmx = do { mx ← mmx; mx }
```

Take away

Summary

- Containers are Functors: they support fmap
- Applicative allows you to combine zero or more containers
- Monads allows effects to depend on values in a container
- A small hierarchy in order of expressiveness:
  - functor
  - applicative functor
  - monad