Functional Programming

Lecture 11: Functors & Applicatives

Twan van Laarhoven

29 November 2021

Outline

- Functors
- Applicative functors
- 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:

$$
\begin{array}{lcl}
\mathsf{map} & :: & (a \to b) \to ([a] \to [b]) \\
\mathsf{mapMaybe} & :: & (a \to b) \to (\mathsf{Maybe}\ a \to \mathsf{Maybe}\ b) \\
\mathsf{mapBtree} & :: & (a \to b) \to (\mathsf{Btree}\ a \to \mathsf{Btree}\ b) \\
\mathsf{mapGtree} & :: & (a \to b) \to (\mathsf{Gtree}\ a \to \mathsf{Gtree}\ b)
\end{array}
$$

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
```

Kinds: types for types

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

What about types with multiple arguments?

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

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

What about types with multiple arguments?

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

instance Functor Either where
  fmap :: (a → b) → (Either a → Either b) -- WRONG
```

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

What about types with multiple arguments?

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

instance Functor Either where
  fmap :: (a → b) → (Either a → Either b) -- WRONG
```

The type-checker should disallow these.

Types of types

The type of a type is called its *kind*.

- Normal types have kind $\star$
- Maybe is a type constructor: a function from types to types.
  Its kind is $\star \to \star$

Types of types

The type of a type is called its *kind*.

- Normal types have kind $\star$
- Maybe is a type constructor: a function from types to types. Its kind is $\star \to \star$

Examples:

$$
\begin{array}{lll}
\mathtt{Int} & :: & \star \\
\mathtt{[]} & :: & \star \to \star \\
\mathtt{Either} & :: & \star \to \star \to \star \\
\mathtt{Either\ a} & :: & \star \to \star \\
\mathtt{Either\ a\ b} & :: & \star
\end{array}
$$

Note that type constructors can be partially applied.

Types of classes

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

We know that f a is a type, f a :: $\star$
We know that a is a type, a :: $\star$
So the argument f has kind $\star \rightarrow \star$.

Types of classes

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

We know that f a is a type, f a :: $\star$
We know that a is a type, a :: $\star$
So the argument f has kind $\star \rightarrow \star$.

Type constructors with this kind are:

- []
- Maybe
- Tree
- Either a

More Functor instances

Back to a type constructors with two or more arguments

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

Now we can define an instance

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

Even more Functor instances

We can define

```
instance Functor ((,) a) where
  fmap :: (b → c) → (a,b) → (a,c)
  fmap f (x,y) = (x, f y)
```

(,) is the binary tuple type constructor

Even more Functor instances

We can define

```
instance Functor ((,) a) where
  fmap :: (b → c) → (a,b) → (a,c)
  fmap f (x,y) = (x, f y)
```

(,) is the binary tuple type constructor

What about the type constructor for functions, (→)?

Even more Functor instances

We can define

```
instance Functor ((,) a) where
  fmap :: (b → c) → (a,b) → (a,c)
  fmap f (x,y) = (x, f y)
```

(,) is the binary tuple type constructor

What about the type constructor for functions, (→)?

```
instance Functor ((→) a) where
  fmap :: (b → c) → ((→) a b → (→) a c)
  fmap = (.)
```

Applicative functors

Functor with multiple arguments

Idea: generalize fmap

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

for example

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

We could introduce a class Functor_n for each fmap_n...

class Applicative

Introduce

```
infixl 4 <*>

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

where

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

```
class (Functor f)
      ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

Applicative generalizes Functor

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
```

Applicative generalizes Functor

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

$$
\begin{aligned}
&(<\$>) \; :: \; \mathsf{Functor}\ f \Rightarrow \; :: \; (a \to b) \to (f\ a \to f\ b) \\
&(<\$>) = \mathsf{fmap}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fmap}_1 \; :: \; (a \to b) \to f\ a \to f\ b \\
&\mathsf{fmap}_1\ g\ x = g <\$> x
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fmap}_2 \; :: \; (a \to b \to c) \to f\ a \to f\ b \to f\ c \\
&\mathsf{fmap}_2\ g\ x\ y = g <\$> x <\!*\!> y
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fmap}_3 \; :: \; (a \to b \to c \to d) \to f\ a \to f\ b \to f\ c \to f\ d \\
&\mathsf{fmap}_3\ g\ x\ y\ z = g <\$> x <\!*\!> y <\!*\!> z
\end{aligned}
$$

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
```

Examples

```
>>> 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

```haskell
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

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

Examples:

```
>>> (+) <$> [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
  mg <*> mx = do {g ← mg; x ← mx; return (g x)}
```

Example: reading n characters from the keyboard

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

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

```haskell
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
```

Exception handling

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

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

(other cases omitted for reasons of space)

Counting evaluation steps

We could instrument the evaluator to count evaluation steps:

```
type Counter a = (a, Int)
evalC :: Expr → Counter Integer
evalC (Lit i) = (i, 1)
evalC (Div e₁ e₂) = let (v₁, n₁) = evalC e₁
                        (v₂, n₂) = evalC e₂
                    in (div v₁ v₂, 1 + n₁ + n₂)
```

(other cases omitted for reasons of space)

Ugly!

- Neither of the two extensions is difficult
- But each is rather awkward, and obscures the previously clear structure
- How can we simplify the presentation?
- What do they have in common?

Evaluator, applicative style

Same evaluator in an applicative style

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

Recovering the vanilla evaluator

Meet the identity functor

```haskell
newtype Id a = Id { fromId :: a }

instance Functor Id where
  fmap :: (a → b) → Id a → Id b
  fmap f (Id x) = Id (f x)

instance Applicative Id where
  pure :: a → Id a
  pure a = Id a
  (<*>) :: Id (a → b) → Id a → Id b
  Id f <*> Id x = Id (f x)
```

pure is the identity and <*> is function application

Example evaluation:

```haskell
>>> Id (+) <$> 1 <*> 2
Id 3
>>> fromId (evalA good)
3
```

The counter instance

Counters instantiate the functor and applicative class:

```
data Counter a = C a Int    -- a value and a count
  deriving (Show)
instance Functor Counter where
  fmap :: (a → b) → Counter a → Counter b
  fmap f (C a n) = C (f a) n
instance Applicative Counter where
  pure :: a → Counter a
  pure a = C a 0
  (<*>) :: Counter (a → b) → Counter a → Counter b
  C f n₁ <*> C x n₂ = C (f x) (n₁ + n₂)
```

Conting as an effect

Increment the count:

```
tick :: Counter ()
tick = C () 1
```

`tick` is only called for its effect, not its value.

Example:

```
>>> (,) <$> tick <*> return "tock"
(C ((),"tock") 1)
>>> return "tock"
(C "tock" 0)
```

Derived operators

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

```haskell
(*>) :: Applicative f => f a -> f b -> f b
a *> b = pure (\_ y -> y) <*> a <*> b
(<*) :: Applicative f => f a -> f b -> f a
a <* b = pure (\x _ -> x) <*> a <*> b
```

Compare `(>>) :: IO a -> IO b -> IO b`

Counting evaluator, applicative style

to integrate tick we use $*>$

```
evalC :: Expr → Counter Integer
evalC (Lit i)     = tick *> pure i
evalC (Add e1 e2) = tick *> pure (+) <*> evalC e1 <*> evalC e2
evalC (Mul e1 e2) = tick *> pure (*) <*> evalC e1 <*> evalC e2
evalC (Div e1 e2) = tick *> pure div <*> evalC e1 <*> evalC e2
```

Example evaluation:

```
>>> evalC good
C 3 5
```

Take away

Summary

- Containers are Functors: they support fmap
- Applicative allows you to combine zero or more containers
- "value with effects" acts like a container