Functors

transforming data in context

Susan Potter

2019-07-03

Functors

Functors: What is a [covariant] functor?

- a covariant functor is a an *abstraction* that models a common computational pattern
- `Functor` is a typeclass in Haskell for a type constructor to implement (*iff* it satisfies the laws)
- `Functor` in Haskell is essentially an *endo*- functor from the category of `Hask` to `Hask` where `Hask` is the category of types in Haskell.
- Note: unless discussing category theory we will use the term *functor* to refer to an endo-functor from `Hask` to `Hask` and when referencing `Functor` we refer to the typeclass in Haskell that represents this concept.
- Unless otherwise specified, *functor* implies covariant functor.

Type constructors: Examples

```haskell
-- Simple contexts
data One a = One a
data Option a = Nothing | Something a
data Or e a = Failure e | Success a
data Pair a = Pair a a

-- "Many" contexts
data Many a = Empty | a :. Many a
data ManyReversed a = REmpty | Many a :- a
data NonEmpty a = a :> Many a

-- Deferred contexts
data Deferred a = Lazy (() -> a) | Now a
```

Functors: Examples

```
transformExactlyOne :: (a -> b) -> One a -> One b
transformExactlyOne f (One a) = One (f a)

transformZeroOrOne :: (a -> b) -> Optional a -> Optional b
transformZeroOrOne _ Nothing       = Nothing
transformZeroOrOne f (Something a) = Something (f a)

transformZeroOrMany :: (a -> b) -> Many a -> Many b
transformZeroOrMany f (a :. as)
  = f a :. transformZeroOrMany f as

transformOneOrMore :: (a -> b) -> NonEmpty a -> NonEmpty b
transformOneOrMore f (a :> as)
  = f a :> transformZeroOrOne f as
```

Functors: Generalize `transform*` pattern

- Using the typeclasses construct we can extend the API that a type exposes if it provides an impelementation (aka *instance*) for a typeclass.
- Typeclasses in Haskell are (crudly speaking) like interfaces in Java after the release where Java interfaces were permitted to have default implementations to interface methods.

Functors: Basic Intuition

- In a wrapped context (the type constructor defines the context) we take the `a` inside and then transform it into something else then rewrap in that same initial context giving us a `f b`.
- A transform *after* an `a` is produced inside of the context.

Functors: The definition (in Haskell)

```haskell
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
        -- Defined in ‘GHC.Base’
        -- Note: (<$) is defined in terms of fmap as
        -- default implementation thus why only fmap
        -- is required to be impelemented in each
        -- instance.
```

Functors: An instance example

```haskell
instance Functor One where
  fmap f (One a) = One (f a)
```

`<$>` / `fmap`: The usage (in `ghci`)

```
 1  effpee> import qualified Data.Char as C
 2  effpee> import Data.Functor ((<$>), Functor (..))
 3  effpee> import Data.Maybe
 4
 5  effpee> (+1) <$> Just 5
 6  Just 6
 7
 8  effpee> fmap C.toUpper ("Hello world!" :: String)
 9  "HELLO WORLD!"
10
11  -- Explain the type on line 14 to yourself
12  effpee> :t (.) <$> (3 :: Int, even :: Int -> Bool)
13  (.) <$> (3 :: Int, even :: Int -> Bool)
14    :: (Int, (z -> Int) -> z -> Bool)
```

fmap-ing over functions

```
effpee> :t C.ord
C.ord :: Char -> GHC.Types.Int

effpee> :t C.toUpper
C.toUpper :: Char -> Char

effpee> f = C.ord <$> C.toUpper
f :: Char -> GHC.Types.Int

effpee> f 'A'
65

effpee> f 'a'
65
```

fmap-ing over IO

```
effpee> import qualified Data.String as S

effpee> :t S.words
S.words :: String -> [String]

effpee> action = (S.words <$> getLine)
action :: GHC.Types.IO [String]

effpee> action
wow is me <-- this was interactive user input
["wow","is","me"]
```

<$: The usage (in `ghci`)

```
effpee> 56 <$ Just True
Just 56

effpee> :t C.toUpper
C.toUpper :: C.Char -> C.Char

effpee> :t s <$ C.toUpper
s <$ C.toUpper :: C.Char -> String

effpee> f = s <$ C.toUpper

effpee> f ’p’
"Functors are cool"
```

Functors: The laws

- Mapping the identity function (`(a -> a)`) over a functor is the same as doing nothing.

```
fmap id = id
```

- Mapping function `g` over a functor and then mapping a function `f` over the resulting functor value should be equivalent to mapping the composed function `f . g` over the original functor value at once.

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

Functors: Counter-Examples

Can we implement a `Functor` instance for the following data types:

```
newtype FromA z a = FromA (a -> z) -- Fix z=()
-- fmap :: (a -> b) -> FromA () a -> FromA () b

newtype Predicate a = Predicate (a -> Bool)
-- fmap :: (a -> b) -> Predicate a -> Predicate b

newtype Tuple k v = Tuple (k, v)
-- fmap :: (a -> b) -> Tuple Int a -> Tuple Int b

-- Can you define the above fmap types?
```

Note: `Tuple` have `Functor` implementations.

Functors: So what?

Properties naturally fall out from these functor laws, e.g.:

```
-- length of a list remains constant under @fmap@ operation
forall f. length (f <$> xs) == length xs

-- @fmap@-ing over @Nothing@ always produces @Nothing@:
forall f. f <$> Nothing == Nothing

-- @fmap@-ing over @Left@ values always produces input:
forall f. f <$> Left err == Left err

-- @fmap@-ing over @(a -> b)@ is composition:
forall f g. f <$> g == f . g
```

Functors: What now?

`Functor` provides an invaluable building block that so many abstractions depend upon:

- `Applicative` functor family (`Alt`, `Alternative`, ...)
- `Monad` family (`MonadPlus`, `MonadFail`, ...)
- `Comonad`
- foundation of recursion schemes
- `Profunctor`
- `Coyoneda`

[Covariant] `Functor` s are the *dual* (waves hands: intuitively opposite to) `Contravariant` [functors].

Abstractions of the same shape

A `Functor` has the following *kind*:

```
effpee> import Data.Functor

effpee> :k Functor
Functor :: (* -> *) -> Constraint

effpee> :k Contravariant
Contravariant :: (* -> *) -> Constraint

effpee> :k Invariant
Invariant :: (* -> *) -> Constraint

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

Functors: Alternatives

Functors can either be:

- *covariant*
- *invariant*
- *contravariant*