Haskell Mini-Course: Functors, Applicatives, Readers, Trave

Generated on 2025-08-22

Table of Contents

1) Overview

Overview
This tutorial explains how the single module above demonstrates core FP patterns in Haskell:
• Either for safe parsing and validation (fail-fast)
• Reader (as plain functions and as Reader/ReaderT) for environment-dependent code
• Functor/Applicative/Monad patterns and when to use each
• traverse/sequence to collect effects across lists
• A custom Validation applicative that accumulates multiple errors
Each section below summarizes the intent, then the Exercises section helps you practice.

2) The Code (verbatim – do not modify)

```
{-# OPTIONS_GHC -Wall #-}
module Main where

import Control.Applicative (liftA2, (*>))
import Data.List.NonEmpty (NonEmpty(..))
import Text.Read (readMaybe)

-- Reader (plain) utilities
import Control.Monad.Reader (Reader, asks, local, runReader)
-- ReaderT transformer (qualified to avoid name clashes)
import qualified Control.Monad.Trans.Reader as RT
import Control.Monad.Trans.Class (lift)
import Data.Foldable (foldMap)

------------------------------------------------------------------
-- Part A — Either
------------------------------------------------------------------

-- (3) readPositive
readPositive :: String -> Either String Int
readPositive s =
  case readMaybe s of
```

```
    Nothing -> Left ("not an Int: " ++ s)
    Just n  -> if n > 0 then Right n else Left ("non-positive: " ++ s)

-- (4) mkRange
mkRange :: String -> String -> Either String (Int,Int)
mkRange loS hiS =
  liftA2 (,) (readPositive loS) (readPositive hiS) >>= \(lo,hi) ->
    if lo < hi then Right (lo,hi) else Left "invalid range: lo>=hi"

-- (5) parseAll
parseAll :: [String] -> Either String [Int]
parseAll = traverse readPositive

------------------------------------------------------------------
-- Part B — Reader (as (-> r))
------------------------------------------------------------------

-- (6) incEnv
incEnv :: Int -> Int
incEnv = fmap (+1) id

-- (7) Applicative combine: Env greeting (curried greet)
data Env = Env { firstName :: String, lastName :: String } deriving Show

greet :: String -> String -> String
greet first last = "Hi " ++ first ++ " " ++ last

fullGreeting :: Env -> String
fullGreeting = greet <$> firstName <*> lastName

-- (8) Reader monad composition
data Cfg = Cfg { base :: Int, factor :: Int } deriving Show

step1 :: Cfg -> Int
step1 = base

step2 :: Int -> Cfg -> Int
step2 x cfg = x * factor cfg

step3 :: Int -> Cfg -> String
step3 x _ = "result=" ++ show x

pipeline :: Cfg -> String
pipeline = step1 >>= step2 >>= step3   -- (-> Cfg) monad

-- (9) local environment tweak (pure functions)
priceWithTax :: Double -> (Double -> Double)
priceWithTax taxRate = \basePrice -> basePrice * (1 + taxRate)

total :: Double -> Double -> Double
total tax basePrice = priceWithTax tax basePrice

totalDiscounted :: Double -> Double -> Double
```

```
totalDiscounted tax basePrice = priceWithTax (tax * 0.9) basePrice

-------------------------------------------------------------------
-- Part C — traverse / sequence
-------------------------------------------------------------------

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

-------------------------------------------------------------------
-- Part D — Mix & Match
-------------------------------------------------------------------

lookupKey :: String -> ([(String, Int)] -> Either String Int)
lookupKey k env =
  case lookup k env of
    Nothing -> Left ("missing: " ++ k)
    Just v  -> Right v

type Assoc = [(String, Int)]

need :: [String] -> (Assoc -> Either String [Int])
need ks env = traverse (\k -> lookupKey k env) ks

data C = C { low :: Int, high :: Int } deriving Show

mkC :: Int -> Int -> Either String C
mkC l h | l < h     = Right (C l h)
        | otherwise = Left "low>=high"

render :: C -> String
render c = "range: [" ++ show (low c) ++ ", " ++ show (high c) ++ ")"

build :: (Int, Int) -> (C -> String)
build (l,h) =
  case mkC l h of
    Left e  -> const ("error: " ++ e)
    Right c -> const (render c)

-- (20) traverse_ (custom)
traverse_ :: Applicative f => (a -> f b) -> [a] -> f ()
traverse_ _ []     = pure ()
traverse_ g (x:xs) = g x *> traverse_ g xs

-------------------------------------------------------------------
-- Add-on 1: Validation style Applicative that accumulates errors
-------------------------------------------------------------------

data V e a = Failure e | Success a
  deriving (Show, Eq)

instance Functor (V e) where
```

```
  fmap f (Success a)  = Success (f a)
  fmap _ (Failure e)  = Failure e

instance Semigroup e => Applicative (V e) where
  pure = Success
  Success f <*> Success a = Success (f a)
  Failure e1 <*> Failure e2 = Failure (e1 <> e2)
  Failure e  <*> _          = Failure e
  _          <*> Failure e  = Failure e

-- A helper to lift String messages into NonEmpty
one :: a -> NonEmpty a
one x = x :| []

readPositiveV :: String -> V (NonEmpty String) Int
readPositiveV s =
  case readMaybe s of
    Nothing -> Failure (one ("not an Int: " ++ s))
    Just n  -> if n > 0 then Success n else Failure (one ("non-positive: " ++ s))

-- Combine two validated fields, then check a cross-field invariant
mkRangeV :: String -> String -> V (NonEmpty String) (Int, Int)
mkRangeV loS hiS =
  case liftA2 (,) (readPositiveV loS) (readPositiveV hiS) of
    Failure es      -> Failure es
    Success (lo,hi) -> if lo < hi
                         then Success (lo,hi)
                         else Failure (one "invalid range: lo>=hi")

-----------------------------------------------------------------
-- Add-on 2: ReaderT + Either composition
-----------------------------------------------------------------

type App e r a = RT.ReaderT r (Either e) a

askAssoc :: App String Assoc Assoc
askAssoc = RT.ask

needKeyT :: String -> App String Assoc Int
needKeyT k = do
  env <- RT.ask
  case lookup k env of
    Nothing -> lift (Left ("missing: " ++ k))
    Just v  -> pure v

needAllT :: [String] -> App String Assoc [Int]
needAllT = traverse needKeyT

-----------------------------------------------------------------
-- Add-on 3: QuickCheck-style notes (kept simple & runnable without QuickCheck)
-----------------------------------------------------------------

-- We show the Functor laws on a few concrete examples
```

```haskell
functorIdTests :: IO ()
functorIdTests = do
  putStrLn "Functor identity law (Maybe):"
  print (fmap id (Just 5) == (Just 5))
  print (fmap id (Nothing :: Maybe Int) == Nothing)

functorCompTests :: IO ()
functorCompTests = do
  putStrLn "Functor composition law (Maybe):"
  let f = (+1); g = (*2)
  print (fmap (f . g) (Just 10) == (fmap f . fmap g) (Just 10))
  print (fmap (f . g) (Nothing :: Maybe Int) == (fmap f . fmap g) (Nothing :: Maybe Int))

{-
-- If you want real QuickCheck, uncomment and add quickcheck to your build:
import Test.QuickCheck
prop_Functor_Id :: Maybe Int -> Bool
prop_Functor_Id x = fmap id x == x

prop_Functor_Comp :: Fun Int Int -> Fun Int Int -> Maybe Int -> Bool
prop_Functor_Comp (Fun _ f) (Fun _ g) x =
  fmap (f . g) x == (fmap f . fmap g) x
-}

-------------------------------------------------------------------
-- Add-on 4: Reader newtype demo with ask/asks/local
-------------------------------------------------------------------

fullGreetingR :: Reader Env String
fullGreetingR = do
  f <- asks firstName
  l <- asks lastName
  pure (greet f l)

promoGreeting :: Reader Env String
promoGreeting =
  local (\e -> e { lastName = lastName e ++ " (VIP)" }) fullGreetingR

-------------------------------------------------------------------
-- Add-on 5: traverse_ via foldMap (alternative implementation)
-------------------------------------------------------------------

traverse_foldMap_ :: (Applicative f, Monoid (f ())) => (a -> f b) -> [a] -> f ()
traverse_foldMap_ g = foldMap (\x -> g x *> pure ())

-------------------------------------------------------------------
-- Alternate styles (point-free / do-notation where it helps)
-------------------------------------------------------------------

-- A3-alt) readPositive (point-free-ish helper)
readPositivePF :: String -> Either String Int
readPositivePF = maybeErr . readMaybe
  where
```

```
    maybeErr Nothing  = Left "not an Int"
    maybeErr (Just n) = if n > 0 then Right n else Left "non-positive"

-- A4-alt) mkRange using do-notation
mkRangeDo :: String -> String -> Either String (Int,Int)
mkRangeDo loS hiS = do
  lo <- readPositive loS
  hi <- readPositive hiS
  if lo < hi then pure (lo,hi) else Left "invalid range: lo>=hi"

-- B7-alt) fullGreeting point-free (same shape as fullGreeting)
fullGreetingPF :: Env -> String
fullGreetingPF = greet <$> firstName <*> lastName

-- B8-alt) pipeline with explicit composition
pipelinePF :: Cfg -> String
pipelinePF cfg = step3 (step2 (step1 cfg) cfg) cfg

-- C12-alt) traverse with safeDiv (same but explicit)
safeDivs :: [Int] -> Maybe [Int]
safeDivs = traverse (safeDiv 100)

-- D15-alt) lookupKey using maybe
lookupKeyPF :: String -> (Assoc -> Either String Int)
lookupKeyPF k env = maybe (Left ("missing: " ++ k)) Right (lookup k env)

-- D16-alt) need (more point-free)
needPF :: [String] -> (Assoc -> Either String [Int])
needPF ks env = traverse (`lookupKey` env) ks

-- D20-alt) traverse_ using foldr
traverse_foldr :: Applicative f => (a -> f b) -> [a] -> f ()
traverse_foldr g = foldr (\x acc -> g x *> acc) (pure ())

----------------------------------------------------------------
-- Demo helpers
----------------------------------------------------------------

sep :: String -> IO ()
sep title = putStrLn ("\n--- " ++ title ++ " ---")

showEitherList :: Show a => Either String [a] -> String
showEitherList (Left e)  = "Left " ++ show e
showEitherList (Right x) = "Right " ++ show x

main :: IO ()
main = do
  sep "Part A — Either"
  print (fmap (+1) (Right 4 :: Either String Int))
  print (fmap (+1) (Left "err" :: Either String Int))
  print (Right 3 >>= (\x -> Right (x*10) :: Either String Int))
  print (Left "bad" >>= (\x -> Right (x*10) :: Either String Int))
  print (liftA2 (+) (Left "A") (Left "B") :: Either String Int)
```

```
print (readPositive "10")
print (readPositive "0")
print (readPositive "abc")
print (mkRange "2" "5")
print (mkRange "5" "2")
putStrLn (showEitherList (parseAll ["3","2","x","5"]))
putStrLn (showEitherList (parseAll ["3","2","5"]))

sep "Part B — Reader"
print (incEnv 41)  -- 42
print (fullGreeting (Env "Ada" "Lovelace"))
putStrLn (pipeline (Cfg 3 7))
print (total 0.15 100)
print (totalDiscounted 0.15 100)

sep "Part C — traverse / sequence"
print (sequence [Just 1, Just 2, Just 3])
print (sequence [Just 1, Nothing, Just 3])
print (traverse (safeDiv 100) [5,4,0,2])

sep "Part D — Mix & Match"
let env = [("a",10),("b",20)] :: Assoc
print (lookupKey "a" env)
print (lookupKey "c" env)
print (need ["a","b"] env)
print (need ["a","z"] env)
putStrLn (render (C 1 4))
putStrLn (build (1,4) (C 100 200))
putStrLn (build (4,1) (C 100 200))

sep "Add-on 1 — Validation (accumulating)"
print (readPositiveV "10")
print (readPositiveV "0")
print (mkRangeV "2" "5")
print (mkRangeV "0" "x")   -- two errors accumulated

sep "Add-on 2 — ReaderT + Either"
let assoc = [("a",1),("b",2)] :: Assoc
print (RT.runReaderT (needAllT ["a","b"]) assoc)
print (RT.runReaderT (needAllT ["a","z"]) assoc)

sep "Add-on 3 — QuickCheck-style sanity checks"
functorIdTests
functorCompTests

sep "Add-on 4 — Reader ask/asks/local"
print (runReader fullGreetingR (Env "Grace" "Hopper"))
print (runReader promoGreeting (Env "Grace" "Hopper"))

sep "Add-on 5 — traverse_ variants"
print (traverse_ (\n -> if n>0 then Just () else Nothing) [1,2,3])
print (traverse_ (\n -> if n>0 then Just () else Nothing) [1,0,3])
```

```
  print (traverse_foldMap_ (\n -> if n>0 then Just () else Nothing) [1,2,3])
  print (traverse_foldr (\n -> if n>0 then Just () else Nothing) [1,2,3])

  putStrLn "Done."
```

3) Part A — Either

Use Either for parsing/validation. Applicative (liftA2) parses independent fields; Monad (>>=) checks cross-field constraints like lo < hi. traverse readPositive turns [String] into Either String [Int].

4) Part B — Reader

Functions (r -> a) form Functor/Applicative/Monad. <$> and <*> feed the same Env to multiple getters; >>= threads the same Cfg through steps.

5) Part C — traverse / sequence

Flip structure and effect: traverse :: (a -> f b) -> t a -> f (t b). With Maybe/Either, failures short-circuit; successes collect.

6) Part D — Mix & Match

Combine Reader and Either: lookups read from an environment and may fail. Build separates validation (mkC) from rendering (pure function).

7) Add-on 1 — Validation

Applicative that accumulates errors via Semigroup/NonEmpty, allowing multiple validation messages.

8) Add-on 2 — ReaderT + Either

A tiny app stack: configuration + fail-with-message. Traverse batches dependent reads.

9) Add-on 3 — Functor law checks

Executable checks for identity and composition laws on Maybe.

10) Add-on 4 — Reader utilities

asks extracts fields; local runs a computation under a tweaked environment.

11) Add-on 5 — traverse_ variants

Three ways to sequence effects while discarding results; foldMap variant needs Monoid (f ()).

12) Exercises (keyed to sections)

Part A — Either (Parsing & Validation)

- 1) Evaluate: fmap (+1) (Right 4); fmap (+1) (Left "err" :: Either String Int).
- 2) Write inputs that make readPositive return Left for (a) non-number, (b) non-positive.
- 3) Predict mkRange "2" "5" and mkRange "5" "2" results; explain why.
- 4) Using parseAll, find the first failing element in ["3","2","x","5"].
- 5) Modify the idea: enforce lo < hi after parsing both ends (explain Applicative vs Monad use).

Part B — Reader (Functions as Environments)

- 6) Explain incEnv = fmap (+1) id in terms of composing functions.
- 7) For Env {firstName, lastName}, explain fullGreeting = greet <$> firstName <*> lastName.
- 8) For Cfg {base,factor}, trace pipeline (Cfg 3 7).
- 9) Compare total vs totalDiscounted on (tax=0.15, base=100).

Part C — traverse / sequence

- 10) Evaluate: sequence [Just 1, Just 2, Just 3] and sequence [Just 1, Nothing, Just 3].
- 11) Evaluate: traverse (safeDiv 100) [5,4,0,2] — where does it stop and why?
- 12) Replace Maybe with Either String in safeDiv to return division-by-zero messages.

Part D — Mix & Match

- 13) Using need ["a","b"] with env = [("a",10),("b",20)], compute the result.
- 14) Using need ["a","z"] with same env, what happens and why?
- 15) Explain how build makes a total renderer from possibly failing mkC.

Add-on 1 — Validation (Accumulating Errors)

- 16) Give inputs that produce two errors in mkRangeV (e.g., both ends invalid).
- 17) Why does V require Semigroup for its error type? Give a concrete error combination.

Add-on 2 — ReaderT + Either

- 18) Show RT.runReaderT (needAllT ["a","b"]) [("a",1),("b",2)].
- 19) Show RT.runReaderT (needAllT ["a","z"]) [("a",1),("b",2)]. Explain the Left.

Add-on 3/4/5 — Laws & Utilities

- 20) Run functorIdTests and functorCompTests. Explain why they print True.
- 21) Show how local changes an Env in promoGreeting compared to fullGreetingR.
- 22) Compare traverse_, traverse_foldr, traverse_foldMap_ in terms of constraints.

13) Glossary

Functor: Type constructor supporting fmap to map a -> b over f a.
Applicative: Extends Functor; apply effectful functions to effectful args with <*>; liftA2 for binary functions.
Monad: Extends Applicative; bind (>>=) allows dependent sequencing.
Reader: Environment-passing pattern; plain functions r -> a behave as a Reader.
ReaderT: Adds a read-only environment on top of another effect.
Either: Error-aware computation: Left e for error, Right a for success.
Maybe: Partial computation: Nothing = failure, Just a = success.
traverse: Map with effects and flip: t a -> f (t b).
sequence: Flip t (f a) to f (t a).
NonEmpty: List with at least one element; useful for aggregating one-or-more errors.
Semigroup: Types with an associative (<>) operation.
Monoid: Semigroup with identity mempty; required by foldMap aggregation.