Todo list

Wild Haskell

Contents

Chapter 1

Your first Haskell program

1.1 Build Tools

1.1.1 Stack

`https://docs.haskellstack.org/en/stable/README/`

make sure ‘stack –version‘ outputs latest stack version, currently ‘Version 1.6.5‘.

‘stack upgrade‘ Add Nix and Cabal

1.2 Editor Integration

1.2.1 VScode with Haskell IDE Engine

[haskell-ide-engine](https://github.com/haskell/haskell-ide-engine)

Install on MacOS, you need to install ‘icu4c‘ on your machine.

git clone https://github.com/haskell/haskell-ide-engine cd haskell-ide-engine make

```
stack install hoogle
```

```
hoogle generate
stack install
```

1.3 Your first haskell program

1.3.1 Using stack

```
stack new guessNumber
```

```
module Main where

import          Control.Monad
```

```haskell
import           Data.Ord      (compare)
import           System.IO     (readLn)
import           System.Random (randomRIO)

guess :: Int -> IO ()
guess secretNumber = do
    print "guess a number"
    guessNumber <- readLn :: IO Int
    case compare guessNumber secretNumber of
        LT -> (print "Too Small!") >> (guess secretNumber)
        EQ -> print "You Win!"
        GT -> (print "Too big!") >> (guess secretNumber)

main :: IO ()
main = do
    secretNumber <- randomRIO (0, 100) :: IO Int
    guess secretNumber
```

Chapter 2

From State to Monad Transformers

2.1 State Monad

First, let's look what is State.

```
newtype State s a = State { runState :: s-> (a, s)}
```

In case you are not familiar with newtype. It is just like *data* but it can only has **exactly one** constructor with exactly one field in it. In this case **State** is the constructor and ‘runState‘ is the single field. The type of ‘State‘ constructor is ‘(s -¿ a, s)) -¿ State s a‘, since ‘State‘ uses record syntax, we have a filed accessor ‘runState‘ and its type is State s a -¿ s -¿ (a, s). ”A State is a function from a state value” to (a produced value, and a resulting state).¡/blockquote¿ An importance takeaway which might not so oblivious to Haskell beginner is that: ‘State‘ is merely a function with type ‘s -¿ (a,s)‘, (a function wrapped inside constructor ‘State‘, to be exact, and we can unwrap it using ‘runState‘).

Secondly, ‘State‘ function takes a ‘s‘representing a state, and produces a tuple contains value ‘a‘ and new state ‘s‘.

exec

The first exercise is to implement ‘exec‘ function, it takes a ‘State‘ function and initial ‘state‘ value, returns the new state.

It is pretty straightforward, we just need to apply ‘State‘ function with ‘state‘ value, and takes the second element from the tuple.

A simple solution can be

```
exec :: State s a -> s -> s
exec f initial = (\ (_, y) -> y) (runState f initial)
```

Since the state value ‘initial‘ appears on both sides of ‘=‘, we rewrite it a little more point-free. ‘runState f‘ is a partial applied function takes ‘s‘ returns ‘(a,s)‘

```haskell
exec f = (\(_, y) -> y) . runState f

exec f = P.snd . runState f
```

We could also write ‘exec‘
‘exec (State f) = P.snd . f‘

2.1.1 Basic Usage of State

```haskell
module Dice where

import Control.Applicative
import Control.Monad.Trans.State
import System.Random

rollDiceIO :: IO (Int, Int)
rollDiceIO = liftA2 (,) (randomRIO (1, 6)) (randomRIO (1,6))

rollNDiceIO :: Int -> IO [Int]
rollNDiceIO 0 = pure []
rollNDiceIO count = liftA2 (:) (randomRIO (1, 6)) (rollNDiceIO (count - 1))

clumsyRollDice :: (Int, Int)
clumsyRollDice = (n, m)
    where
        (n, g) = randomR (1, 6) (mkStdGen 0)
        (m, _) = randomR (1, 6) g

-- rollDice :: StdGen -> ((Int, Int), StdGen)
-- rollDice g = ((n, m), g'')
--     where
--         (n, g') = randomR (1, 6) g
--         (m, g'') = randomR (1, 6) g'

-- use state to construct
rollDie :: State StdGen Int
rollDie = state $ randomR (1, 6)

-- use State as Monad
rollDieM :: State StdGen Int
rollDieM = do generator <- get
              let (value, generator') = randomR (1, 6) generator
              put generator'
              return value

rollDice :: State StdGen (Int, Int)
```

```
rollDice = liftA2 (,) rollDieM rollDieM

rollNDice :: Int -> State StdGen [Int]
rollNDice 0 = state (\s -> ([], s))
rollNDice count = liftA2 (:) rollDieM (rollNDice (count - 1))
```

How about draw card from a deck

```
module Deck where

import Control.Applicative
import Control.Monad.Trans.State
import Data.List
import System.Random

data Rank = One | Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten | Jack | Queue | K

data Suit = Diamonds | Clubs | Hearts | Spades deriving (Bounded, Enum, Show, Eq, Ord)

data Card = Card Suit Rank deriving (Show, Eq, Ord)

type Deck = [Card]

fullDeck :: Deck
fullDeck = [Card suit rank  | suit <- enumFrom minBound,
                              rank <- enumFrom minBound]

removeCard :: Deck -> Int -> Deck
removeCard [] _ = []
removeCard deck index = deck' ++ deck''
    where (deck', remain) = splitAt (index + 1) deck
          deck''     = drop 1 remain


drawCard :: State (StdGen, Deck) Card
drawCard = do (generator, deck) <- get
              let (index, generator') = randomR (0, length deck ) generator
              put (generator', removeCard deck index)
              return $ deck !! index

drawNCard :: Int -> State (StdGen, Deck) [Card]
drawNCard 0 = state (\s -> ([], s))
drawNCard count = liftA2 (:) drawCard (drawNCard $ count - 1)
```

How about folding a list using ‘State‘ https://github.com/yuanw/applied-haskell/blob/2018/monad-transformers.mdhow-about-state

```
foldState :: (b -> a -> b) -> b -> [a] -> b
foldState f accum0 list0 =
    execState (mapM_ go list0) accum0
  where
    go x = modify' (\accum -> f accum x)
```

2.2 Monad Transformers

2.2.1 Motivation

Why we cannot compose any two monad

```
import Control.Applicative

newtype Compose f g a = Compose (f (g a))

instance (Functor f, Functor g) => Functor (Compose f g) where
    fmap f (Compose h) = Compose ((fmap . fmap) f h)


instance (Applicative f, Applicative g) => Applicative (Compose f g) where
    pure = Compose . pure . pure
    (Compose h)  <*> (Compose j) = Compose $ pure (<*>) <*> h <*> j

instance (Monad f, Monad g) => Monad (Compose f g) where
     (Compose fga) >>= m = Compose $ fga >>= \ ga -> let gfgb = ga >>= (return . m) in
```

https://stackoverflow.com/questions/7040844/applicatives-compose-monads-dont

2.2.2 Build a composed monad by hand

```
newtype StateEither s e a = StateEither
    { runStateEither :: s -> (s, Either e a)
    }
```

Let's implement the functor instance of this typeP

References `https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State` `https://haskell.fpcomplete.com/library/rio`

Chapter 3

Learning Lens

The content of this chapter comes fron Simon Peyton Jones's Lenses: Compositional data access and mainpulation talk.

3.1 Native Approach

```
data Person = Person { name    :: String,
                     , address :: Address
                     , salary  :: Int }

data Address = Address { road :: String
                       , city :: String
                       , zip  :: String }
```

The goals for lens
* To access and update a given field * Compose Lense

```
data LensR s a = LensR { viewR :: s -> a
                       , setR  :: a -> s -> s }

data Prisms s a = Prisms { match :: s -> Either s a
                         , build :: a -> s}
```

Type ‘s‘ represent the overall record we try to access or update, Type ‘a‘ represent the field we are try to acccess or update.
How to compose ‘LensR‘

```
composeL :: LensR s1 s2 -> LensR s2 a -> LensR s1 a
composeL (L v1 s1) (L v2 s2)
    = L (\ s -> v2 (v1 s))
        (\ a s -> s1 (s2 a (v1 s)) s)
```

The two big problems with this approach is to inefficent, and inflexiable.

3.1.1 A step further on this Native Approach

```
data LensR s a
    = L { viewR :: s -> a
        , setR  :: a -> s -> a
        , mod   :: (a -> a) -> s -> s
        , modM  :: (a -> Maybe a) -> s -> Maybe s
        , modIO :: (a -> IO a) -> s -> IO s }
```

The last two (or three if you know your functor well) share lots of commonality.
If we can abstract the common pattern

```
data LensR s a
    = L { viewR :: s -> a
        , setR  :: a -> s -> a
        , mod   :: (a -> a) -> s -> s
        , modF  :: Functor f => (a -> f a) -> s -> f s }

type Lens' s a = forall f. Functor f => (a -> f a) -> s -> f s
```

You may think ‘Lens'‘ looks nothing like ‘LensR‘, but they are actually isomorphic.
Side Bar :: Isomorphic
If ‘A‘ and ‘B‘ are isomorphic, it means we are getting ‘B‘ to ‘A‘ without lost any information, and vice verse.

Before proving ‘Lens'‘ and ‘LensR‘ is isomorphic, allow me introduce two oddly looking functors ‘Const‘ and ‘Identity‘.

Break Down::

```
view :: Lens' s a -> s -> a => ((a -> f a) -> s -> f s) -> s -> a
view ln s = getConst (ln const s)

let fred = P {_name = "Fred", _salary = 100}
name :: Lens' Person String
name fn (P n s) -> (\ n' -> P n' s) <$> fn n

view name fred
= (getConst . name . const) fred
= getConst ((\n' -> P n' s) <$> Const "Fred")
= getConst . Const "Fred"
= "Fred"

set name "John" fred
= getID $ name (Id . Const  "John") fred
= getID $ (\ n' -> P n' 100) <$> ((ID. Const  "John" ) fred)
= getID $ (\ n' -> P n' 100) <$> Id "John"
= getID $ ID (P "John" 100 )
= P "John" 100
```

Compose Lens'

Function application

https://www.youtube.com/watch?v=sfWzUMViPOM&t=441s

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwiS6dD65PfdAhVSJjQIHRzzC9IQFjAAegQIBBAC&url=http%3A%2F%2Fwww.cs.ox.ac.uk%2Fpeople%2Fjeremy.gibbons%2Fpublications%2Fpoptics.pdf&usg=AOvVaw3KgxUg3x37WIToXgmRu79C

The problem: heterogeneous composite data

Worse: consider access to the A in a composiation data structure Maybe (A x B) built using both sums and products.

Sum part comes with Maybe ‘data Maybe a = Just a — Nothing‘

```
*Main Lib> import Control.Lens
*Main Lib Control.Lens> ("hello","world") ^. _2
"world"
*Main Lib Control.Lens>
```

3.2 What is a Lens ?

Lenses address some part of a “structure” that always exists, either look that part, or set that part. “structure” can be a computation result, for example, the hour in time. Functional setter and getter. Data.Lens

```
data Lens s a = Lens { set  :: s -> a -> a
                     , view :: s -> a
                     }
view :: Lens s a -> s -> a
set  :: Lens s a -> s -> a -> s
```

view looks up an attribute a from s view is the getter, and set is the setter. Laws 1. set l (view l s) s = s 2. view l (set l s a) = a 3. set l (set l s a) b = set l s b Law 1 indicates lens has no other effects since set and view in Lens both start with s -¿, so we can fuse the two functions into a single function ‘s -¿ (a -¿ s, a)‘.

So we could define Lens as

```
data Lens s a = Lens (s -> (a -> s), a)
data Store s a = Store (s -> a) s
data Lens s a = Lens (s -> Store a s)
```

Side bar Store Comonad urlhttps://stackoverflow.com/questions/8428554/what-is-the-comonad-typeclass-in-haskell hurlttps://stackoverflow.com/questions/8766246/what-is-the-store-comonad

Lens can form a category

Semantic Editor Combinator

The Power is in the dot

```
(.)           :: (b -> c) -> (a -> c) -> (a -> c)
(.).(.)       :: (b -> c) -> (a1 -> a2 -> b) -> a1 -> a2 -> c
(.).(.).(.) :: (b -> c) -> (a1 -> a2 -> a3 -> b) -> a1 -> a2 -> a3 -> c
```

Detail Explantation

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \ t -> f (g t)
```

'((.).(.)) (+1) (+) 10 10 = 21'

$$((.).(.))(+1)(+)1010 = (.)((.)(+1))(+)1010 = ((.)(+1))(+10)10 = ((+1).(+10))10 = (+1)((+10)10) = (+1) \tag{3.1}$$

'(.).(.) :: (b -¿ c) -¿ (a -¿ b) -¿ a -¿ c' 'dotF . dotG :: (b -¿ c) -¿ (a -¿ c) -¿ a -¿ c' 'dotF :: b -¿ c' 'dotG :: a -¿ b'

since 'dotF' is just an alias to '(.)' 'dotF :: (u -¿ v) -¿ (s -¿ u) -¿ s -¿ v' '(u -¿ v) -¿ (s -¿ u) -¿ s -¿ v === b -¿ c' therefore 'b === (u -¿ v)' 'c === (s -¿ u) -¿ s -¿ v'

'dotG' is also an alias to '(.)' 'dotG :: (y -¿ z) -¿ (x -¿ y) -¿ x -¿ z' '(y -¿ z) -¿ (x -¿ y) -¿ x -¿ z === a -¿ b' therefore 'a === (y -¿ z)' 'b === (x -¿ y) -¿ x -¿ z'

since 'b' appears on both side

''' a === y -¿ z b === (u -¿ v) b === (x -¿ y) -¿ x -¿ z c === (s -¿ u) -¿ s -¿ v '''

we can deduct

''' u === x -¿ y v === x -¿ z '''

therefore 'c === (s -¿ x -¿ y) -¿ s -¿ x -¿ z'

'(.).(.) === a -¿ c' '(.).(.) === (y -¿ z) -¿ (s -¿ x -¿ y) -¿ s -¿ x - z'

we can generalize this to any functor

```
fmap               :: Functor f  => (a -> b) -> f a -> f b
fmap . fmap        :: ( Functor f, Functor g)   => (a -> b) -> f (g a) -> f (g b)
fmap . fmap . fmap :: (Functor f, Functor g, Functor h) => (a -> b) -> f (g (h a)) ->
```

it means we compose a function under arbitrary level deep nested context Semantic Editor Combinator

'type SEC s t a b = (a -¿ b) -¿ s -¿ t' it like a functor (a -¿ b) -¿ f a -¿ f b 'fmap' is Semantic Editor Combinator 'fmap . fmap' is also a SEC

so we use 's' to generalize 'f a', 'f (g a)', 'f (g (h a))' and 't' to generalize 'f b', 'f (g b)', 'f (g (h b))'. It may seems counterintuitive at first, we lost the relation between a and s, b and t. Functor is a semantic Editor Combinator 'fmap :: Functor f =¿ SEC (f a) (f b) a b'

first is a also SEC ?

```
first :: SEC (a, c) (b,c) a b
first f (a, b) = (f a, b)
```

Setters We can compose Traversable the way as we can compose '(.)' and 'fmap'

```
traverse                     :: (Traversable f, Applicative m) => (a -> m b) -> f a 
traverse . traverse          :: (Traversable f, Traversable g, Applicative m) => (a 
traverse . traverse . traverse :: (Traversable f, Traversable g, Traversable h, Applica
```

traverse is a generalized version of mapM, and work with any kind of Foldable not just List.

```
class (Functor f, Foldable f) => Traversable f where
   traverse :: Applicative m => (a -> m b) -> f a -> m (f b)

fmapDefault :: forall t a b. Traverable t => (a -> b) -> t a -> t b
fmapDefault f = runIndentity . traverse (Identity . f)
```

build ‘fmap‘ out from traverse we can change ‘fmapDefault‘

```
over l f = runIdentity . l (Identity . f)
over traverse f = runIdentity . traverse (Identity . f)
                = fmapDefault f
                = fmap f
```

type of ‘over :: ((a -¿ Identity b) -¿ s -¿ Identity t) -¿ (a -¿ b) -¿ s -¿ t‘ type Setter s t a b = (a -¿ Identity b) -¿ s -¿ Identity t so we could rewrite ‘over‘ as over :: Setter s t a b -¿ (a -¿ b) -¿ s -¿ t let’s apply setter

```
mapped :: Functor f => Setter (f a) (f b) a b
mapped f = Identity . fmap (runIdentity . f)
over mapped f = runIdentity . mapped (Identity . f)
              = runIdentity . Identity . fmap (runIdentity . Identity . f)
              - fmap f
```

Examples

```
over mapped (+1) [1,2,3]  ===> [2,3,4]
over (mapped . mapped) (+1) [[1,2], [3]] ===> [[2,3], [4]]
chars :: (Char -> Identity Char) -> Text -> Identity Text
chars f = fmap pack . mapped f . unpack
```

Laws for setters Functor Laws: 1. ‘fmap‘ id = id 2. fmap f . fmap g = fmap (f . g)

Setter Laws for a legal Setter l. 1. over l id = id 2. over l f . over l g = over l (f . g)

Practices Simplest lens `(1,2,3) ^. _2    ===>` 2 view `_2 (1, 2, 3)`

References SPJ Lenses: compositional data access and manipulation Edward Kmett’s NYC Haskell Meetup talk http://comonad.com/haskell/Lenses-Folds-and-Traversals-NYC.pdf)Edward Kmett’s NYC Haskell Meetup talk slide `http://hackage.haskell.org/package/lens-tutorial-1.0.3/docs/Control-Lens-Tutorial.html` `https://www.youtube.com/watch?v=H01dw-BMmlE` `https://www.youtube.com/watch?v=QZy4Yml3LTY` `https://www.youtube.com/watch?v=T88TDS7L5DY` `http://lens.github.io/tutorial.html` `https://www.reddit.com/r/haskell/comments/9ded97/is_learning_how_to_use_the_lens_library_worth_it/e5hf9ai/` `https://blog.jle.im/entry/lenses-products-prisms-sums.html`