Todo list

Wild Haskell

Contents

Chapter 1

Your first Haskell program

1.1 Setup GHC and Build Tool(s)

In the current Haskell ecosystem, there are 3 major build tools: cabal, stack, and Nix.

1.1.1 cabal via ghcup

https://github.com/haskell/ghcup#installation

1.1.2 Stack

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

make sure ‘stack –version‘ outputs latest stack version, currently ‘Version 1.10.1‘. ‘stack upgrade –git‘

1.1.3 Nix

Nix is not avaible for windows out of box.

https://github.com/Gabriel439/haskell-nix

```
curl https://nixos.org/nix/install | sh
nix-env --install cabal2nix
nix-env --install nix-prefetch-git
```

If you already installed cabal through whatever methods, you probably want to skip install cabal via nix.

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
```

```haskell
module Main where

import           Control.Monad
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

Monad

Tackling the Awkward Squad

Chapter 3

Applicative

Applicative Programming with Effects

Chapter 4

From State to Monad Transformers

4.1 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)‘

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

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

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

4.1.1 Basic Usage of State

```
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)
```

```haskell
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

```haskell
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)
```

4.2 Monad Transformers

4.2.1 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

4.2.2 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

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

instance Functor (StateEither s e) where
    f `fmap` StateEither se = StateEither (\ s -> let (s', eitherA) = se s in (s', f `:

instance Applicative (StateEither s e) where
    pure a = StateEither (\ s -> (s, Right a))
```

```haskell
    StateEither h <*> StateEither g = StateEither (\ s -> let (s', eitherF) = h s
                                                              (s'', a)      = g s in (s'', either

instance Monad (StateEither s e) where
    StateEither f >>= g = StateEither $ \ s ->
        case f s of
            (s', Left e) -> (s', Left e)
            (s', Right a) -> (runStateEither $ g a) s'

execStateEither :: StateEither s e a -> s -> s
execStateEither stateE = fst . runStateEither stateE

modify' :: (s -> Either e s) -> StateEither s e ()
modify' f = StateEither $ \ s ->
    case f s of
        Left e -> (s, Left e)
        Right s' -> (s', Right ())

foldTerminate :: (b -> a -> Either b b) -> b -> [a] -> b
foldTerminate f accum xs=
        execStateEither  (mapM_ go xs) accum
    where go x = modify' (`f` x)
```

https://mmhaskell.com/blog/2017/3/6/making-sense-of-multiple-monads

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

Chapter 5

STM

5.1 TVar

Shared memory location that support atomic memory transcations.

```haskell
newTVar :: a -> STM (TVar a)
```

```haskell
newTVarIO :: a -> IO (TVar a)
```

Chapter 6

Learning Lens

The content of this chapter comes fron Simon Peyton Jones's Lenses: Compositional data access and mainpulation talk, and Edward Kmett's NYC Haskell Meetup talk.

6.1 Motivation

The record problem often represented as the hello world for Lens.

The problem: heterogeneous composite data

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

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

```
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

6.2 Native Approach

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

Type **s** represent the overall record we try to access or update, Type **a** represent the field we are try to access or update.

How to compose **LensR**

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

The two big problems with this approach is to inefficient, and inflexible.

6.2.1 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.

> 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=sfWzUMViP0M&t=441s

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwiS6dD65PfdAhVSJjQIHRzzC9
url=http%3A%2F%2Fwww.cs.ox.ac.uk%2Fpeople%2Fjeremy.gibbons%2Fpublications%
2Fpoptics.pdf&usg=AOvVaw3KgxUg3x37WIToXgmRu79C

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

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

6.3 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)
```

6.3.1 Semantic Editor Combinator

Let's start with Setter, which is a nice entry point for learning Lens.

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
```

we can generalize this to any functor. **(.)** is the **fmap** for the functor instance.

```
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)) ->
```

This means we compose a function under arbitrary depth Functor 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. But it allows us to have an unified type represents them. Functor is a semantic Editor Combinator fmap :: Functor f =¿ SEC (f a) (f b) a b

first is a also SEC, fmap is a SEC

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

6.3.2 Setters

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

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

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

Mixing 'traverse' and 'fmap' might lead to odd behavior /todofind a case

```
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
```

`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)`

Third Program: Upload Spreadsheet to google bigquery

In this chapter, we are going to write a command line program to upload a spreadsheet file’s content to Google BigQuery. There is awesome library Gogol provides Haskell binding to Google API.

We need gogol 0.3.0 or higher, Make sure your project’s resolver version is lts-9.0 or later. You can verify it by doing ‘stack list-dependencies‘.

Starting Point

So how to use Gogol library ? The lib provides an [example](https://github.com/brendanhay/gogol/blob/develop/examples/sr for Google Cloud Storage.

Let’s do a little change, so it fetch all the bigquery project the current default google credential has access to.

You need to install gcloud, and setup default

```
gcloud init
gcloud auth application-default login
```

and let’s put the following code into our ‘Main.hs‘.

```haskell
module Main where

import Control.Lens                 ((&), (.~), (<&>), (?~))
import Control.Monad.Trans.Resource (runResourceT)
import Control.Monad.IO.Class

import System.IO (stdout)
import Network.Google.Auth   (Auth, Credentials (..), initStore)
import qualified Network.Google         as Google
import qualified Network.Google.BigQuery as BigQuery
import Network.HTTP.Conduit (Manager, newManager, tlsManagerSettings)
import           Network.Google.Auth.Scope       (AllowScopes (..),

example :: IO BigQuery.ProjectList
example = do
    lgr  <- Google.newLogger Google.Debug stdout
    m <- liftIO (newManager tlsManagerSettings) :: IO Manager
    c <- Google.getApplicationDefault m
 -- Create a new environment which will discover the appropriate
 -- AuthN/AuthZ credentials, and explicitly state the OAuth scopes
 -- we will be using below, which will be enforced by the compiler:
     env  <- Google.newEnvWith c lgr m <&>
           (Google.envLogger .~ lgr)
         . (Google.envScopes .~ BigQuery.bigQueryScope)
    runResourceT . Google.runGoogle env $ Google.send BigQuery.projectsList

 main :: IO ()
 main = do
     projects <- example
     print projects
```

You need add following build depends to make stack build successed.

```
build-depends:       base >= 4.7 && < 5
                   , bytestring
                   , conduit
                   , conduit-extra
                   , gogol
                   , gogol-bigquery
                   , gogol-core
                   , http-conduit
                   , lens
                   , resourcet
```

Quite few things need to unpack here.
Create a bigquery dataset

```
{-# LANGUAGE RankNTypes      #-}
{-# LANGUAGE TemplateHaskell #-}

module Main where

import           Control.Lens

data Point = Point
  { _postionX :: Double
  , _postionY :: Double} deriving (Show)

makeLenses ''Point

data Segment = Segment {
  _segmentStart :: Point,
  _segmentEnd   :: Point
} deriving (Show)
makeLenses ''Segment

makePoint :: (Double , Double) -> Point
makePoint = uncurry Point

makeSegment :: (Double, Double) -> (Double, Double) -> Segment
makeSegment start end = Segment (makePoint start) (makePoint end)

updateSegment :: Segment -> Segment
updateSegment = (segmentStart .~ makePoint (10, 10)) . (segmentEnd .~ makePoint (10, 10))
```

http://www.scs.stanford.edu/16wi-cs240h/slides/concurrency-slides.html#(1)
catchJust]

Add exception handle in the guess number program

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

Detail Explantation

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

$$
\begin{aligned}
&((.).(.))(+1)(+)1010 = 21\\
&((.).(.))(+1)(+)1010\\
&= (.)((.)(+1))(+)1010\\
&= ((.)(+1))(+10)10\\
&= ((+1).(+10))10\\
&= (+1)((+10)10)\\
&= (+1)(10+10)\\
&= 21
\end{aligned}
$$

```
(.).(.) :: (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 **(.)**

$$dotF :: (u- > v)- > (s- > u)- > s- > v(u- > v)- > (s- > u)- > s- > v === b- > c \tag{6.1}$$

therefore

$$b === (u- > v)c === (s- > u)- > s- > v \tag{6.2}$$

'dotG' is also an alias to '(.)'

$$dotG :: (y- > z)- > (x- > y)- > x- > z(y- > z)- > (x- > y)- > x- > z === a- > b \tag{6.3}$$

therefore

$$a === (y- > z)b === (x- > y)- > x- > z \tag{6.4}$$

since **b** appears on both side

$$a === y- > zb === (u- > v)b === (x- > y)- > x- > zc === (s- > u)- > s- > v \tag{6.5}$$

we can deduct

$$
\begin{aligned}
&u === x- > y\\
&v === x- > z23
\end{aligned}
$$

therefore

$$c === (s- > x- > y)- > s- > x- > z \tag{6.6}$$

$(.).(.) === a- > c$

Chapter 7

Exception

References

- stanford cs240
- fp Complete