Monad Transformers in Haskell

Azad Bolour
Bolour Computing
azadbolour@bolour.com

sample code at:
https://github.com/azadbolour/transformersworkshop

Thank You!

Eitan Chatav - for pairing on monad transformers

James Earl Douglas - for reviewing the slides

Mio Alter - for co-organizing a precursor study session on monad transformers

Intended Audience

fluent in basic Haskell, including monads

unfamiliar with monad transformers
except in passing

Goals

demystify the concept and its implementation

demonstrate its patterns of usage

monad transformers allow
the effects of different monads
to be combined

Why Study Monad Transformers

- **current Haskell standard for combining effects is monad transformers**
- **real-world computations often involve multiple effects, e.g., IO + failure**
- **effective Haskell requires familiarity with patterns of working with multiple effects through monad transformers**
- **to make your monads composable, provide standard transformers for them**

monads are cool … [but] …

[they] require monad transformers for composition

I tried to wrap my head around it but then it exploded [!]

[Martin Odersky - Scala Days - 2015]

[http://www.slideshare.net/Odersky/scala-days-san-francisco-45917092]

Example of Working with Monad Transformer: Persistent

```
dbAction ::
  ReaderT SqlBackend (NoLoggingT (ResourceT IO)) ()

dbAction = do
  runMigration migrateAll
  productId <- insert $ Product "MacBook Pro" $ 2000.00
  product <- get productId
  liftIO $ print product

main :: IO ()
main = runSqlite ":memory:" $ dbAction
```

Composition

(.)	(b -> c) -> (a -> b) ->	(a -> c)
(>=>)	(a -> m b) -> (b -> m c) ->	(a -> m c)
hcompose	(a -> m_1 b) -> (b -> m_2 c) ->	(a -> ??? c)

hcompose	$(a \rightarrow m_1\ b) \rightarrow (b \rightarrow m_2\ c) \rightarrow$	$(a \rightarrow ???\ c)$

primitive monad [m2]	the ??? = transformer	transformed Identity (m1 = Identity)	derived monad
Maybe a	MaybeT m_1 a		
Either e a	ExceptT e m_1 a	ExceptT e Identity	Except e
((->) r)	ReaderT r m_1 a	ReaderT r Identity	Reader r
((,) w)	WriterT w m_1 a	WriterT w Identity	Writer w
s -> (a, s)	StateT s m_1 a	StateT s Identity	State s

Overview

1. >> Motivation

2. Implementation

3. Usage

The Monad Type Class

```
class Applicative struct => Monad struct where
      return :: a -> struct a
      (>>=) :: struct a -> (a -> struct b) -> struct b

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

class Functor struct where
        fmap :: (a -> b) -> struct a -> struct b
```

Shorthand Notational Conventions

- ***struct a*** **or** $struct_a$ **— generic type**
- ***struct a*** **or** $struct_a$ **— generic data constructor**
- ***struct Int*** **or** $struct_{Int}$ **— concrete type**
- $struct1.struct2_a$ **—** ***struct1 (struct2 a)***
- ab **—** ***a -> b***
- abc **—** ***a -> b -> c***
- $struct_{ab}$ **—** ***struct (a -> b)***

Terminology

- effectless function: *f :: a -> b*
  - *f x = 2*x*
- effectful function aka monad factory: *f :: a -> monad b*
  - *f x = return (2*x)*
- substrate [of monad]: monadic type parameter of monad
  - *monad :: [Int] - - substrate = Int*
  - *monad :: State s a - - substrate = a*

return :: substrate -> $monad_{substrate}$
(>>=) :: $monad_a$ -> MonadFactory monad a b -> $monad_b$

Propagation of Effects

Kleisli Composition of effectful functions: (>=>)

(>=>) :: (a -> monad b) -> (b -> monad c) -> (a -> monad c)

(f >=> g) a = f a >>= g

multiple effectful functions may be composed in a chain

the effect is accumulated within the chain

Bind as Flattening fmap

(>>=) :: monad a -> (a -> monad b) -> monad b

join :: (Monad monad) => monad (monad a) -> monad a

bind and join can be defined in terms of each other

join nested = nested >>= id [Control.Monad]

$monad_a$ >>= $monadFactory_{ab}$ =
  join $\$$ $monadFactory_{ab}$ $<\$>$ $monad_a$

Flattening

the inner and outer monads have the same monadic type so flattening makes sense

```
factory :: Int -> Maybe Int
factory x = Just (2 * x)

join $ fmap factory (Just 10)
= join $ Just (Just 20)
= Just 20
```

Kleisli Composition via *fmap* and *join*

```
f :: a -> monad b
g :: b -> monad c

(f >=> g) a =
    let monad_b = f a                            -- Step 1. apply f.
        monad_monad(c) = g <$> monad_b           -- Step 2. map g.
    in join monad_monad(c)                       -- Step 3. flatten.
```

Heterogeneous Composition

```
f :: a -> SomeMonad Int
g :: Int -> Maybe Int
g x = Just (2 * x)
(f >=> g) a = join $ g <$> f a
```

fmap of composition still works - flattening does not!

f a	g <$> f a	nested value
`[1, 2]`	`Just . (2*) <$> [1, 2]`	`[Just 2, Just 4]`
`Right 5`	`Just . (2*) <$> Right 5`	`Right (Just 10)`
monad$_{Int}$	`Just . (2*) <$>` monad$_{Int}$	`monad (Maybe Int)`

from Homogeneous Bind to Heterogeneous Bind

Homogeneous

```
nester :: m a -> (a -> m b) -> m (m b)
nester m_a factory_ab = factory_ab <$> m_a

(>>=) m_a factory_ab = join $ nester m_a factory_ab
```

Heterogeneous

```
nester :: (Monad m, Monad blendingMonad) =>
   m a -> (a -> blendingMonad b) -> m (blendingMonad b)
nester m_a factory_ab = factory_ab <$> m_a

hBind m_a factory_ab = hJoin $ nester m_a factory_ab
```

hBind for Maybe

```
maybeFactory_ab :: a -> Maybe b

maybeHBind baseMonad_a maybeFactory_ab =
   maybeHJoin $ maybeFactory_ab <$> baseMonad_a

let

maybeHJoin :: baseMonad a
                -> (a -> Maybe b)
                -> MaybeBlender baseMonad b
```

Blender Abstraction

```
class (Monad blendingMonad) =>
  MonadBlender blender blendingMonad where
    hJoin :: (Monad m, Monad blendingMonad) =>
        m (blendingMonad b) -> blender m b

instance MonadBlender MaybeBlender Maybe where
  hJoin :: (Monad baseMonad) =>
    baseMonad (Maybe a) -> MaybeBlender baseMonad a
```

Propagating Nested Effects

```
data MaybeBlender baseMonad a = ???

instance Monad baseMonad =>
      instance Monad (MaybeBlender baseMonad)
```

About hJoin

- *starts off with a nesting of the two monads, e.g., 'baseMonad (**Maybe** a)'*
- *converts the nested monads to an appropriate data structure that:*
    - *retains the effects of both monads*
    - *is itself a monad: can propagate the combined effects*
    - *can just box the nested monad in a data structure*
    - *or do some proper processing of the nested monad - poor man's flattening*

Heterogeneous Composition with Blenders

```
hBind :: (Monad m, MonadBlender blender blendingMonad) =>
    m a -> (a -> blendingMonad b) -> blender m b
hBind mₐ blendingFactory = hJoin $ blendingFactory <$> mₐ

hCompose :: (Monad m, MonadBlender blender blendingMonad) =>
    (a -> m b) -> (b -> blendingMonad c) ->  (a -> blender m c)

hCompose f blendingFactory = \a -> hBind (f a) blendingFactory
hCompose f blendingFactory = \a -> hJoin $ blendingFactory <$> (f a)
```

from Blenders to Transformers

- ***blenders and transformers are different abstractions***
- ***but they use common data structures***
- ***the common data structures are named: MonadT, e.g., MaybeT***
- ***each primitive monad, Mnd a, e.g.,***

  Maybe a, reader a: r -> a, writer a: (a, w), etc.

  has a related data structure: MndT baseMonad a, e.g., MaybeT baseMonad a
- ***MndT blends the effect of Mnd and baseMonad***

from Blenders to Transformers

- ***blenders and transformers differ as abstractions (classes)***
- ***the key concepts of the blender abstraction are hJoin (and its derivatives, hBind, hCompose)***
  hJoin :: baseMonad (Maybe a) -> MaybeT baseMonad a
- ***the key concept of the transformer abstraction is lift***
- ***lift adds the effect of a blending monad to a base monad***
  lift :: baseMonad a -> MaybeT baseMonad a

The Monad Transformer Abstraction

```
class MonadTrans blendingTransformer where
   lift :: Monad baseMonad =>
              baseMonad a -> blendingTransformer baseMonad a
```

[Control.Monad.Trans.Class]

a monad transformer as a polymorphic type has 2 type parameters

- *a* ***baseMonad*** *- the monad being transformed*
- *a* ***substrate*** *- the type of slots in the base monad and in the transformer itself as a monad*

About the MonadTrans Abstraction

lift vs return

- **return adds an effect to a substrate value**
- **lift adds another effect to a base monad value**

***MonadT* needs these instances**

- ***instance MonadTrans MonadT***
- ***instance Monad baseMonad***
  => instance Monad (MonadT baseMonad)

lift via hBind

lift :: (Monad m, MonadBlender blender blendingMonad) =>
m a -> blender m a

lift m_a *= hBind* m_a *return*$_{blendingMonad}$

lift m_a *= hJoin $ return*$_{blendingMonad}$ *<$>* m_a

Food for Thought
an Unrealized Dream

hCompose **as defined here**

- ***does not smoothly subsume homogeneous Kleisli composition***
- ***does not produce a category - with identity and associativity properties***

unclear how to model such a uniform categorical model in Haskell

even if we started from scratch with freedom

- ***to choose our own data structures other than transformers***
- ***to use the power of GHC extensions***

MonadTrans Laws

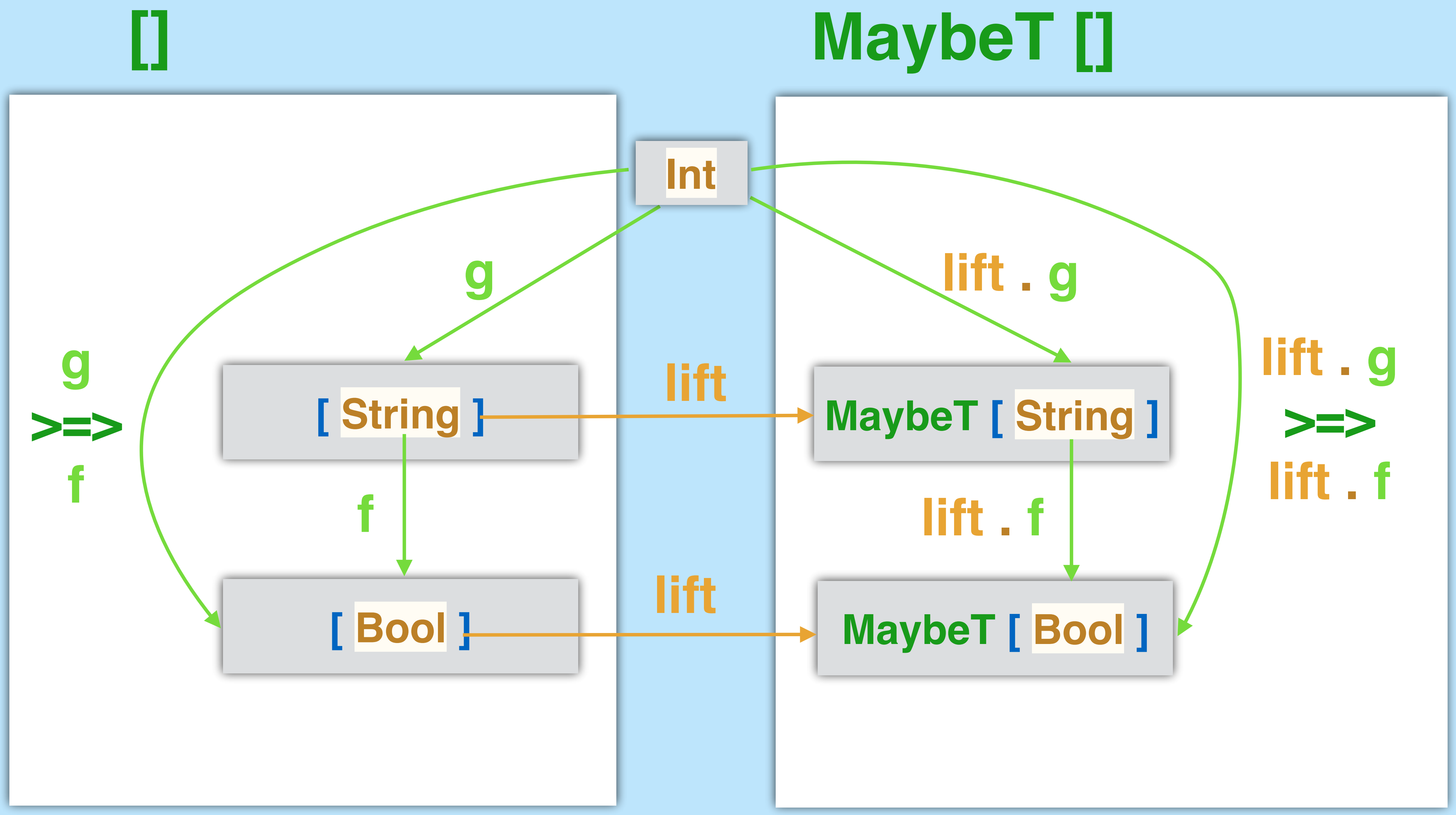
return law

$\text{lift} \,.\, \text{return}_{\text{baseMonad}} = \text{return}_{\text{MonadT}}$

$\text{lift} \,.\, \text{Kleisliid}_{\text{baseMonad}} = \text{Kleisliid}_{\text{MonadT}}$

composition law: for functions: f :: b -> baseMonad c, g :: a -> baseMonad b
and monads: m :: baseMonad b

lift (m >>= f)	= lift m >>= (lift . f)	bind version
lift (g a >>= f)	= (lift . g) a >>= (lift . f)	let m = g a
lift ((g >=> f) a)	= ((lift . g) >=> (lift . f)) a	definition of Kleisli
lift . (g >=> f)	= (lift . g) >=> (lift . f)	Kleisli version



MonadTrans Composition Law (Kleisli version)

lift . (g >=> f) = (lift . g) >=> (lift . f)

MonadTrans Laws =>
Monad Laws for lifted Functions

```
liftFunction :: (a -> baseMonad b) -> a -> MaybeT baseMonad b
liftFunction f =  lift . f
```

baseMonad instance	MaybeT baseMonad instance
lift . (return >=> f) = lift . f	return >=> lift . f = lift . f
lift . (f >=> return) = lift . f	lift . f >=> return = lift . f
lift . ((h >=> g) >=> f)	(lift . h >=> lift . g) >=> lift . f
lift . (h >=> (g >=> f))	lift . h >=> (lift . g >=> lift . f)

MaybeT Transformer

hJoin :: baseMonad (Maybe a) -> MaybeT baseMonad a

- ***join*** **would have *flattened* the nested structure**
- ***hJoin cannot flatten***
- ***hJoin*** **can only package up the nested monads**

MaybeT - the Monad Transformer for Maybe

```
newtype MaybeT baseMonad a = MaybeT {
   runMaybeT :: baseMonad (Maybe a)
}
```

[Control.Monad.Trans.Maybe]

```
hJoin = MaybeT
```

note: the runner is a getter for the core

```
runMaybeT :: MaybeT baseMonad a -> baseMonad (Maybe a)
```

Using MaybeT

MaybeT baseMonad **- must be an** ***instance*** **of both**

- ***Monad***
- ***MonadTrans***

assume ***it is [stay tuned for the evidence]***

```
findPersonByName :: String -> MaybeT IO Person - - given

findPersonAddress :: String -> MaybeT IO Address

findPersonAndPrintAddress name = do
    person <- findPersonByName name
    return $ address person
```

Exercise

Working with combined effects.

In this exercise we use Maybe and Either to represent different exceptional conditions:

- ***Maybe is used to represent the existence or non-existence of an entity***
- ***Either String is used to represent a validation error***

The combined effect is represented by *MaybeT Either.*

Given two entity sets, *customers*, and *products*, create a function that takes a customer name and a product name and computes the discounted price of the product for the customer, by working in the *MaybeT Either* monad.

See the skeletal program: *CalcDiscount.hs*.

Where are We

1. Motivation

2. >> Implementation

   - >> *MaybeT* as *MonadTrans*

   - *MaybeT* as *Monad [and Applicative and Functor]*

   - special notes

3. Usage

Shorthand Notation

MaybeT.monad_{a} :: MaybeT monad a

$\text{MaybeT.[]}_{\text{Int}}$ :: MaybeT [] Int

$\text{fmap}_{\text{MaybeT.m}}$ :: (a -> b) -> MaybeT m a -> MaybeT m b

monad.Maybe_{a} :: monad (Maybe a)

$\text{[].Maybe}_{\text{Int}}$ :: [Maybe Int]

$\text{fmap}_{\text{monad.Maybe}}$ :: (a -> b) -> monad.Maybe_{a} -> monad.Maybe_{b}

$\text{fmap}_{\text{[].Maybe}}$ :: (Int -> Bool) -> [Maybe Int] -> [Maybe Bool]

MaybeT as a MonadTrans

```
instance MonadTrans MaybeT where
    lift_MaybeT = MaybeT . liftM_baseMonad Just - - Control.Monad.Trans.Maybe

[liftM same as fmap - liftM :: (Monad m) => (a -> b) -> m a -> m b]
[lift_MaybeT :: (Monad m) => m a -> MaybeT m a]
```

example

```
    lift_MaybeT [1, 2] = MaybeT $ Just <$> [1, 2]
                       = MaybeT [Just 1, Just 2]
```

MaybeT as a Monad

since Monad is a subclass of Functor and Applicative

MaybeT must be an instance of

- **Functor - evidence derivation follows**
- **Applicative - see Appendix B for summary**
- **Monad - evidence derivation follows**

MaybeT as a Functor

instance (Functor functor) => Functor (MaybeT functor) where

fmap :: (a -> b) -> MaybeT functor a -> MaybeT functor b = ???

MaybeT functor x :: MaybeT { runMaybeT :: functor.Maybe x }

core

fmap f_{ab} maybeT

1. unbox the core: *runMaybeT maybeT*
2. map f_{ab} onto the core: *$fmap_{functor.Maybe}$ f_{ab} (runMaybeT maybeT)*
3. box the mapped value:

   MaybeT ($fmap_{functor.Maybe}$ f_{ab} (runMaybeT maybeT))

fmap for MaybeT

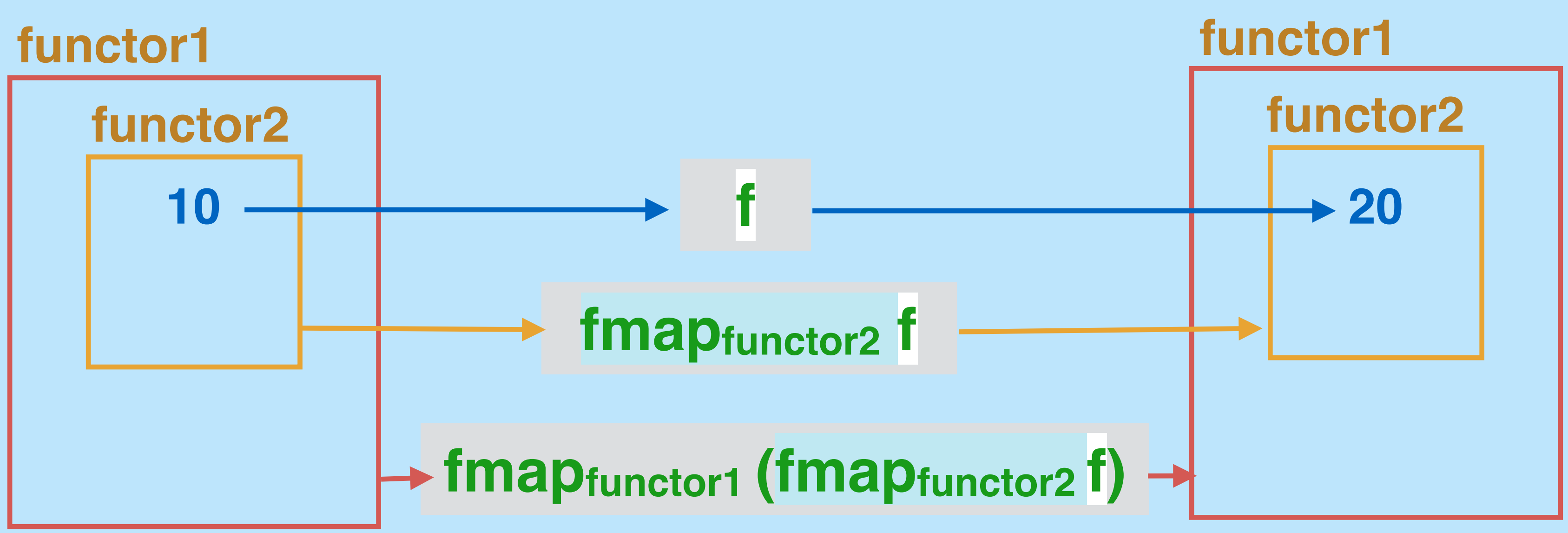
[library function in Control.Monad.Trans.Maybe]

mapMaybeT f = MaybeT . f . runMaybeT - -

fmap**$_{\textit{MaybeT.functor}}$ **f_{ab} maybeT =
MaybeT (fmap**$_{\textit{functor.Maybe}}$ **f_{ab} (runMaybeT maybeT))

fmap$_{\text{MaybeT.functor}}$
= mapMaybeT fmap$_{\text{functor.Maybe}}$
= mapMaybeT (fmap$_{\text{functor}}$ **. fmap**$_{\text{Maybe}}$**) - -** ***stay tuned***

fmap for Nested Functors



f = (*2)

MaybeT as a Monad

```
instance Monad m => Monad (MaybeT m) where
      return = ???
      maybeT >>= maybeTFactory = ???
```

$\text{return}_{\text{MaybeT}}$ :: a -> MaybeT { m (Maybe a) }

$(\text{>>=})_{\text{MaybeT}}$ ::

MaybeT { m (Maybe a) } -> (a -> MaybeT { m (Maybe b) })

-> MaybeT { m (Maybe b) }

MaybeT as a Monad: return

$\text{return}_{\text{MaybeT.monad}}$ :: a -> MaybeT { monad (Maybe a) }

$\text{return}_{\text{MaybeT.monad}}$ = $\text{lift}_{\text{MaybeT}}$. $\text{return}_{\text{monad}}$

[by the return law of MonadTrans]

$\text{return}_{\text{MaybeT.[]}}$ 1
= $\text{lift}_{\text{MaybeT}}$ ($\text{return}_{[]}$ 1)
= $\text{lift}_{\text{MaybeT}}$ [1]
= MaybeT [Just 1]

Bind for MaybeT

```
(>>=)MaybeT ::
        MaybeT { monad (Maybe a) }
        -> (a -> MaybeT { monad (Maybe b) } )
        -> MaybeT { monad (Maybe b) }

maybeT >>= factoryMaybeT.monad = MaybeT $
    domonad
        maybe <- runMaybeT maybeT
        case maybe of
            Nothing -> return Nothing
            Just value -> runMaybeT (factoryMaybeT.monad value)
```

Checking the Monad, Applicative, and Functor Laws for MaybeT

Must be done, of course.

Omitted in the interest of time.

Where are We

1. Motivation
2. >> Implementation
    - *MaybeT* as *MonadTrans*
    - *MaybeT* as *Monad [and Applicative and Functor]*
    - >> special notes
3. Usage

Nesting of Functions

- **structure of the core of transformers so far:**
  - ***baseMonad (BlendingMonad)***
- **some primitive monads are functions**
  - ***((->) r), s -> (a, s)***
- **the nesting formula creates nested functions - inconvenient**
  - **multiple functions one in each slot of base monad**
- **more convenient to have a composition model with**
  - **a single function**
  - **a single base monad**

Special Transformer Types for Monads that are Functions

```
newtype ReaderT env monad a
  = ReaderT { runReaderT :: env -> monad a }
```

in the heterogeneous composition model:

```
hJoin :: monad ( env -> a ) ->
  ReaderT { runReaderT :: env -> monad a }
```

in this case hJoin performs its version of *flattening*

see Appendix A for details

see ReaderBlender.hs in workshop project for worked out example

Monads Defined by Using Monad Transformers

in *transformers* package: *Control.Monad.Trans*
some monads are defined as: *MndT Identity*

Identity has no effect - so

MndT Identity is isomorphic to *Mnd*

example

newtype ExceptT err monad a = ExceptT (monad (Either err a))
type Except err = ExceptT err Identity

allows consistent and convenient use of functions for the derived monad based on the corresponding transformer functions

Monad Transformer Packages

transformers package

includes

Control.Monad.Trans.Class (MonadTrans, …)

Control.Monad.Trans.Identity (IdentityT, …)

Control.Monad.Trans.Maybe (MaybeT, …)

Control.Monad.Trans.Except (ExceptT, …)

etc.

primitive monad [m2]	the ??? = transformer	transformed Identity (m1 = Identity)	derived monad
Maybe a	MaybeT m_1 a		
Either e a	ExceptT e m_1 a	ExceptT e Identity	Except e
((->) r)	ReaderT r m_1 a	ReaderT r Identity	Reader r
((,) w)	WriterT w m_1 a	WriterT w Identity	Writer w
s -> (a, s)	StateT s m_1 a	StateT s Identity	State s

Monad Transformer Packages

***mtl package** - Monad Transformer Library*

Control.Monad.Writer, **etc.**

adds *elevator type classes and instances*

- **e.g., *MonadWriter*, *MonadReader*, etc.**
- **to allow monads produced in lower level transformers**
- **to be automatically lifted (*elevated*) to the top of the stack**
- ***stay tuned***

Specific Functionality of Common Monads

specific transformers have specific behavior
over and above monadic behavior

various convenience functions

getters and setters of related values

- **ReaderT - environment value**
- **WriterT - log value**
- **StateT - state value**

Primitive Constructors

turn a primitive monad to a corresponding transformer

- ***reader :: (Monad monad) => (r -> a) -> ReaderT r monad a***
  reader f_{ra} = ReaderT (return$_{monad}$. f_{ra})

- ***state :: (Monad monad) => (s -> (a, s)) -> StateT s monad a***
  state f_{sas} = StateT (return$_{monad}$. f_{sas})

- ***writer :: (Monad monad) => (a, log) -> WriterT log monad a***
  writer = WriterT . return$_{monad}$

Sampling of Specific Functions

- *ReaderT r m a*
  - *ask :: Monad monad => ReaderT r m r*
  - *local :: (r -> r) -> ReaderT r m a -> ReaderT r m a*

- *WriterT w m a*
  - *tell :: Monad m => w -> WriterT w m ()*
  - *listen :: Monad m => WriterT w m a -> WriterT w m (a, w)*

- *StateT s m a*
  - *get :: Monad m => StateT s m s*
  - *put :: Monad m => s -> StateT s m ()*

Exercise

Getting familiar with the transformer library.

1. ***Check and justify the implementations of bind, return, and lift for ReaderT and WriterT.***
2. ***Compare Reader and Writer with their counterparts in GHC Base.***
3. ***Write a simple do block that shows the effects of tell and listen.***
4. ***Write a simple do block that shows the effects of put and get.***
5. ***Check and justify the implementations of ask, and put.***

The sources for the transformer library are under: https://hackage.haskell.org/package/transformers-0.5.2.0/docs/

Where are We

1. Motivation
2. Implementation
3. >> Usage
   - Transformer stacks
   - Intro to Step-by-Step Tutorial
   - Patterns/idioms

Stacking of Monad Transformers

a monad that was lifted by one transformer Monad1T may be lifted again by another transformer Monad2T, etc.

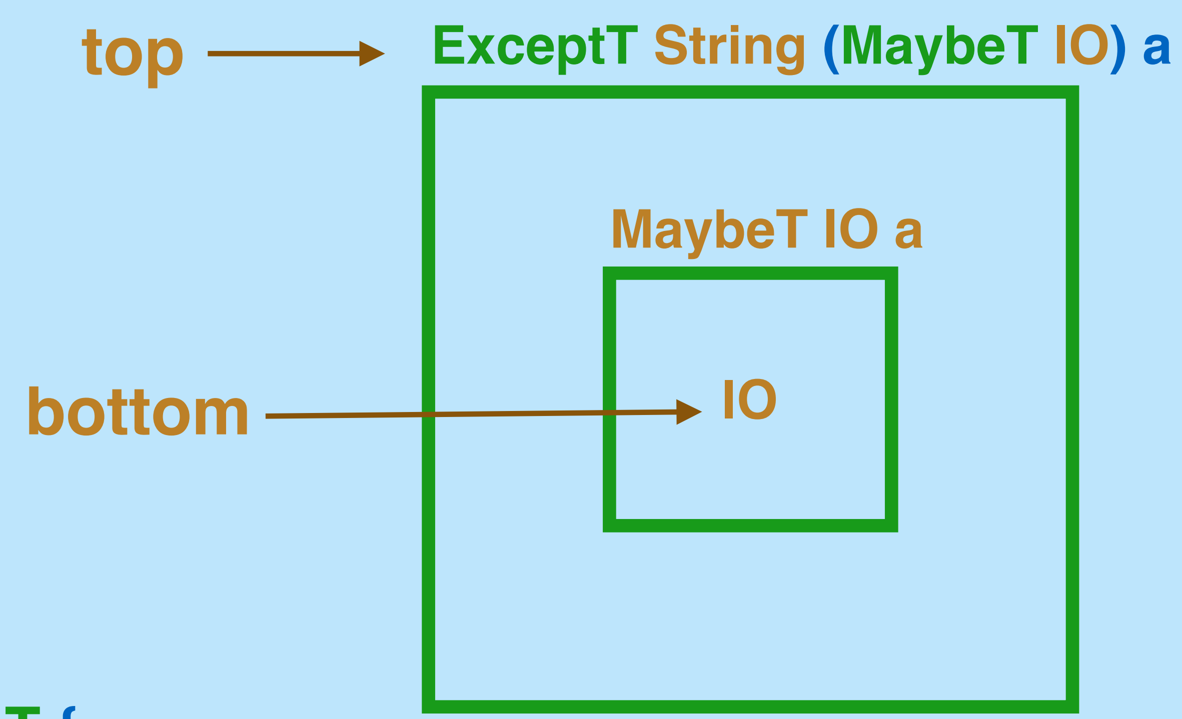
repeated lifting gives rise to a *monad transformer stack*

Terminology

The base monad is *lifted into* the monad transformer.
The monad transformer is the *top of the stack*.

Top refers to the transformer data structure.

Stacking of Monad Transformers



stack: [ExceptT, MaybeT, IO]

data structure

ExceptT String { runExceptT :: MaybeT {
runMaybeT :: IO (Maybe (Either String a)) }

nesting order is reversed in the core

Nesting Reversal

```
Monad1T baseMonad1 a =
   Monad1T { runMonad1T :: baseMonad1 (Monad1 a) }

Monad2T baseMonad2 a =
   Monad2T { runMonad2T :: baseMonad2 (Monad2 a) }

let baseMonad1 a = Monad2T baseMonad2 a
   in Monad1T { runMonad1T :: baseMonad1 (Monad1 a) }

Monad1T { runMonad1T :: (Monad2T baseMonad2) (Monad1 a) }

Monad1T {
   runMonad1T :: Monad2T {
      runMonad2T :: baseMonad2 (Monad2 (Monad1 a)) } }
```

The Running Example

extended sample nicely developed in the paper

> ***Monad Transformers Step by Step***
>
> ***http://catamorph.de/documents/Transformers.pdf***

defines an expression evaluator

successively adds new effects to the evaluator
by extending the monad transformer stack
to propagate useful effects
as an expression is evaluated

Expression Evaluator Model

```
type VarName = String

data Exp = Lit Integer
         | Var VarName
         | Plus Exp Exp
         | Lambda VarName Exp
         | Apply Exp Exp

type Env = Map.Map VarName Value

data Value = IntVal Integer | FunVal Env VarName Exp

eval :: Env -> Exp -> Value
```

[see EvaluatorTypes.hs in the workshop exercises]

Steps in Building the Evaluator Transformer Stack

eval :: Env -> Exp -> Value

progression effects added to the transformer stack

- **no effect**
- **generic effect: convert to evaluate inside a monad**
- **capture and propagate errors - add *ExceptT***
- **abstract out the evaluation environment - add *ReaderT***
- **others - add *WriterT*, *StateT***

Effectless Expression Evaluation

```
eval :: Env -> Exp -> Value

eval env (Lit i) = IntVal i

eval env (Var var) =
  fromJust $ Map.lookup var env

eval env (Plus expr1 expr2) =
  let IntVal val1 = eval env expr1
      IntVal val2 = eval env expr2
  in IntVal (val1 + val2 )
```

[see BasicEvaluator.hs in workshop exercises]

Idioms/Patterns

- **monadic renderer**
  refactor an effectless computation into ***generic monadic form***
- **concrete framer**
  turn a generic monadic value to a specific monadic value by framing it in a concrete context - applying a concrete function to it
- **core runner/accessor**
  reach inside a transformer stack to access the core:
  the combined effectful value
- **elevator**
  automatically lift [elevate] effectful functions from a stack's lower level transformers to the top of the stack
- **flipped reader**
  readably provide an environment to an inlined reader block

Concrete Framer Idiom

```
class ExampleClass where …

genericFactory :: (ExampleClass example) => args -> example

instance ExampleClass Example1 where …
instance ExampleClass Example2 where …
```

the application of *genericFactory* can be specialized to an instance by *composition with a framing function, e.g.,*

```
framer :: Example1 -> Result

(framer . genericFactory) :: args -> Result
```

the composition forces *genericFactory* to be run with *Example1*'s dictionary

Framing the Generic Monad

```
eval :: (Monad monad) => Env -> Exp -> monad Value

let expr = Lit 12 `Plus` Var "x"
    env = singleton "x" (IntVal 2)

eval env expr :: monad Value

runIdentity :: Identity a -> a
runIdentity $ eval env expr
```

Exercise: Framing

create framing functions

runInList :: [a] -> [a]

runInMaybe :: Maybe a -> Maybe a

use them to frame an expression value produced by the basic evaluator in a list and in a Maybe

Core Accessor Idiom Framing in Identity

```
newtype ExceptT err monad a =
   ExceptT { runExceptT :: monad (Either err a) }

type TransformerStack monad val = ExceptT String monad val
type StackCore val =  Either String val

eval :: (Monad monad) => Env -> Exp -> TransformerStack monad Value


runTransformerStack :: TransformerStack Identity val -> StackCore val
runTransformerStack = runIdentity . runExceptT

runTransformerStack $ eval env (Var "x")
```

Effectful Evaluation with a Nested Transformer Stack

```
newtype Monad1T monad a = { runMonad1T :: monad (Monad1 a) }
newtype Monad2T monad a = { runMonad2T :: monad (Monad2 a) }

type TransformerStack monad val = Monad1T (Monad2T monad val)

   = Monad1T { runMonad1T :: (Monad2T monad) (Monad1 val) }

   = Monad1T { runMonad1T :: Monad2T {
                    runMonad2T :: monad (Monad2 (Monad1 val)) } }

eval :: (Monad monad) => Env -> Exp -> TransformerStack monad Value
```

Nested Core Accessor

```
type TransformerStack monad val = Monad1T (Monad2T monad) val

   = Monad1T { runMonad1T :: Monad2T {
                            runMonad2T :: monad (Monad2 (Monad1 val)) } }


runTransformerStack :: TransformerStack monad val ->
         Monad2 (Monad1 val)

runTransformerStack = runIdentity . runMonad2 . runMonad1
```

Nested Core Accessor: Example

```
type TransformerStack monad val

    = WriterT [String] (ExceptT String monad val)
    = WriterT { runWriterT :: ExceptT {
                    runExceptT :: monad (Either String (val, [String])) } }


eval :: (Monad monad) => Env -> Exp -> TransformerStack monad Value


runTransformerStack :: TransformerStack Identity val -> Either String (val, [String])
runTransformerStack = runIdentity . runExceptT . runWriterT
```

Elevator Pattern: the Problem

type StackT monad val = Monad1T monad val

typical function in a *Monad1T do* block

f :: (Monad monad) => arg₁ -> … -> StackT monad val

type StackT' monad val = Monad2T (Monad1T monad) val

to upgrade to *StackT'* all our f's have to change

f' :: (Monad monad) => arg₁ -> … -> StackT' monad val

f' = lift . f

code overly dependent on the particular stack

Elevator: Example Problem

using only the *transformer* package

```
newtype WriterT log m a = WriterT { runWriterT :: m (a, log) }
tell :: (Monad m) => log -> WriterT log m () - - append log

type StackT' m val = MaybeT (WriterT [String] m) val

eval :: (Monad m) => Env -> Exp -> StackT' m Value
eval env (Var var) =
    lift $ tell [var]
    return $ fromJust $ Map.lookup var env
```

```
type StackT'' m val =
   ReaderT String MaybeT (WriterT [String] m) val

eval :: (Monad m) => Env -> Exp -> StackT'' m Value
eval env (Var var) =
   lift $ lift $ tell [var]
   return $ fromJust $ Map.lookup var env
```

Can access to lower-level functions be made generic?

Elevator: in mtl Library

mtl

```
class MonadWriter log monad where
      tell :: log -> monad ()
```

```
instance MonadWriter w m => MonadWriter w (MaybeT m) where
       tell = lift . tell
```

basis of recursion is the actual implementation of tell for WriterT

transformers WriterT

```
tell :: (Monad m) => log -> WriterT log m ()
tell w = WriterT $ return ((), w)
```

Elevator: in mtl Library

example usage

```
eval :: (Monad monad) =>
   Env -> Exp -> MaybeT (WriterT [String]) monad val


eval env (Var var) = do
   tell [var]
   ….
```

Elevator: in mtl

if every transformer above WriterT on the stack is a MonadWriter
then tell is elevated to the top of the stack

the monads in mtl are all friends
each is an instance of the others' elevator classes

Working with IO In Monad Transformers

there is no IO transformer

IO is always at the bottom of a transformer stack

- **MaybeT (ExceptT String IO) a**
- **ReaderT r (MaybeT IO) a**
- **Monad1T (Monad2T … (MonadnT IO) …)**

Lifting an IO Action into a Transformer Stack

- **MaybeT (ExceptT String IO) a**
- **ReaderT r (MaybeT IO) a**
- **Monad1T (Monad2T … (MonadnT IO) …)**

to lift an IO action into a transformer stack
requires composition of lifts for all the transformers on the stack

- $\mathbf{lift_{MaybeT} \,.\, lift_{ExceptT}\ ioAction}$
- $\mathbf{lift_{ReaderT} \,.\, lift_{MayT}\ ioAction}$
- $\mathbf{lift_{Monad1T} \,.\, lift_{Monad2T} \,.\,.\,.\, lift_{MonadnT}\ ioAction}$

convenient to have an abstraction for multi-level IO lifts - *liftIO*

special elevator for IO

Special Elevator for IO

```
class Monad m => MonadIO m where liftIO :: IO a -> m a
instance MonadIO IO where liftIO = id -- basis
```

recursive instance definitions - generic form

instance (MonadIO stackTail) => MonadIO (HeadMonadT stackTail)
    where $\text{liftIO}_{\text{HeadMonadT}} = \text{lift}_{\text{HeadMonadT}} \, . \, \text{liftIO}_{\text{stackTail}}$

example

instance (MonadIO stackTail) => MonadIO (MaybeT stackTail)
    where $\text{liftIO}_{\text{MaybeT}} = \text{lift}_{\text{MaybeT}} \, . \, \text{liftIO}_{\text{stackTail}}$

Flipped Reader Pattern

```
type Env = String
type StackT a = ReaderT Env (MaybeT IO) a
runFlipped :: Env -> StackT a -> IO (Maybe a)
runFlipped env stack =  runMaybeT $ (runReaderT stack) env

action1 :: String -> StackT String
action2 :: String -> StackT String

main :: IO ()
main = do
    env <- getLine
    result <- runFlipped env $ do
        s1 <- action1 "1"
        s2 <- action2 "2"
        return (s1 ++ s2)
    print result
```

References

- **Monad Transformers Step by Step - Martin Grabmüller**
  http://catamorph.de/documents/Transformers.pdf

- **Functional Programming with Overloading and Higher-Order Polymorphism - Mark P. Jones**
  http://web.cecs.pdx.edu/~mpj/pubs/springschool95.pdf

- **Building Monad Transformers - Part 1 - Jakub Arnold**
  http://blog.jakubarnold.cz/2014/07/22/building-monad-transformers-part-1.html

- **Gentle Introduction to Monad Transformers - Lim H.**
  https://github.com/kqr/gists/blob/master/articles/gentle-introduction-monad-transformers.md

Appendix A

Derivation of ReaderT in the Heterogeneous Composition Model

Structure of ReaderT
Review of ((->) r)

(->) r - monads are functions from a given domain - r

fmap for reader

$monad_x :: r \to x$
$f_{xy} :: x \to y$
$monad_y :: r \to y = fmap\ f_{xy}\ monad_x = f_{xy}\ .\ monad_x$

bind for reader

(>>=) :: monad x -> (x -> monad y) -> monad y
$(>>=)_{reader} :: (r \to x) \to (x \to r \to y) \to (r \to y)$

$(rx >>= xry)\ r_1 = xry\ (rx\ r_1)\ r_1$

join for reader

join :: monad (monad x) -> monad x
$join_{reader} :: (r \to (r \to x)) \to (r \to x)$

$join\ rrx\ r_1 = rrx\ r_1\ r_1$

Review of the ((->) r) Monad

bind monad factory = join $ fmap factory monad

for the monad ((->) r) - fmap = (.)

$factory_{xry}$. $monad_{rx}$ = \\r_1 -> $factory_{xry}$ ($monad_{rx}$ r_1) :: r -> r -> y

reader's join flattens (r_1 -> (r_2 -> y)) to (r_1 -> y) by applying the nested function to the outer r_1

for each (r_1, ry) pair of the nested function structure
join (r_1, ry) = (r_1, ry r_1)

Bind via Join for ((->) r)

example

```
ax = double, xay = (+)
ax >>= xay
= join $ (+) <$> double = join $ (+) <$> { (1 -> 2), (2 -> 4), … }
= join $ { (1 -> (2 +), (2 -> (4 +)) … }
= { (1, 3), (2, 6), (3, 9), … }
```

Structure of ReaderT: Recap

hBind** $baseMonad_a$ $blendingMonadFactory_{ab}$ **=
    ***hJoin $** $blendingMonadFactory_{ab}$ **<$>** $baseMonad_a$*

Blending with Functions as Monads

functionFactory :: *x* -> (*a* -> *y*)

nester *baseMonad*$_x$ *functionFactory*$_{xy}$ =
functionFactory$_{xy}$ <$> *baseMonad*$_x$

hBind *baseMonad*$_x$ *functionFactory*$_{xy}$ =
hJoin $ *nester* *monad*$_x$ *maybeFactory*$_{xy}$

example - baseMonad = []

nester [1, 2, 3] (+) = [(1 +) (2 +), (3 +)]

Blending with Functions as Monads

nester* *[1, 2, 3] (+) = [(1 +) (2 +), (3 +)]

goal - the final result of blending needs to capture both the ideas of list and reader a = a function from type a

issue

nesting captures the idea of a list
but not of a single function from type a
each slot has its own function

hJoin

should restore "reader-ness"
convert [(1 +) (2 +), (3 +)] so it represents a single function
while retaining list-ness

Blending with Functions as Monads

nester [1, 2, 3] (+) = [(1 +) (2 +), (3 +)]

hJoin

> **restores *simple reader-ness***
> **converts [(1 +) (2 +), (3 +)] so it represents a single function while retaining list-ness**

```
flatten :: [Int -> Int] -> (Int -> [Int])
flatten :: monad (r -> a) -> r -> monad a
flatten monadOfFunction = \r -> monadOfFunctions <*> (return r)

hJoin monadOfFunctions = ReaderT { flatten monadOfFunction }

newtype ReaderT env monad a
   = ReaderT { runReaderT :: env -> monad a }
```

Appendix B

<*> for MaybeT

MaybeT as Applicative

instance (Applicative appl) =>

Applicative (MaybeT appl) where

$\text{pure}_{\text{MaybeT}}$ a = MaybeT \$ $\text{pure}_{\text{appl}}$ \$ Just a

$\text{maybeT}_{\text{ab}}$ $<*>_{\text{MaybeT}}$ maybeT_{a} =

let core_{ab} = runMaybeT $\text{maybeT}_{\text{ab}}$

core_{a} = runMaybeT maybeT_{a}

in MaybeT \$ $\text{liftA2}_{\text{appl}}$ $(<*>)_{\text{Maybe}}$ core_{ab} core_{a}

[liftA2 "applies" a 2-arg function to two applicative args]

[derivation of <*> skipped for brevity]

check: <*> for [Maybe a]

(<*>)$_{\text{[].Maybe}}$ = liftA2$_{\text{[]}}$ (<*>)$_{\text{Maybe}}$

liftA2 f a b = fmap f a <*> b

((return f) <*>) = (f <$>) hence ((Just f) <*>) = (f <$>)

[Just f1, Just f2] <*>$_{\text{[].Maybe}}$ maybes = liftA2 (<*>) [Just f1, Just f2] maybes

= fmap (<*>) [Just f1, Just f2] <*> maybes

= [((Just f1) (<*>)), ((Just f2) (<*>))] <*> maybes

= [(f1 <$>), (f2 <$>)] <*> maybes

[Just (* 2), Just (+ 100)] <*>$_{\text{[].Maybe}}$ [Nothing, Just 1]
= [Nothing, Just 2, Nothing, Just 101]