CMPT 383

Lecture 8: Monads

Anders Miltner

Monads

- “Just a monoid in the category of endofunctors”
  - Famous phrase in “Categories for the Working Mathematician”
    - The worst book to learn category theory from
    - The book I learned category theory from
  - Fictionally attributed to Philip Wadler in Brief, Incomplete and Mostly Wrong History of Programming Languages
- Actually hard to really understand — we’ll do a LOT of examples, in addition to mathematical definitions and intuitive descriptions

Review of Functors and Applicatives

- Functors

  `fmap` `a->b` + $\boxed{\texttt{a}}$ → $\boxed{\texttt{b}}$

- Applicatives

  `pure` `a` → $\boxed{\texttt{a}}$

  `<*>` $\boxed{\texttt{a->b}}$ + $\boxed{\texttt{a}}$ → $\boxed{\texttt{b}}$

- The square denotes a type constructor (functor or applicative)
- Intuitively, the square means a box, container, structure, or context, …

Motivating Example: SafeDiv

- Given the following definition of Expr

```
data Expr = Val Int | Div Expr Expr deriving (Show)
```

- To handle the potential failure properly, define a safediv function

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

- Write a function to evaluate an expression

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
                   Nothing -> Nothing
                   Just n  -> case eval y of
                                Nothing -> Nothing
                                Just m  -> safediv n m
```

Motivating Example: SafeDiv

- The eval function resolves the “divide by zero” issue, but in a verbose way
- Let’s try to improve it using applicatives (because Maybe is an applicative)

```
eval :: Expr -> Maybe Int
eval (Val n)   = pure n
eval (Div x y) = pure safediv <*> eval x <*> eval y
```

- The above code does not type check
  - Because it requires safediv to have type Int -> Int -> Int
  - But then safediv cannot indicate failure …
- The eval function does not fit the pattern captured by applicatives

Motivating Example: SafeDiv

- Observe the common pattern in the verbose eval where we need a case analysis on a Maybe value (Nothing to Nothing, Just x to sth about x)

- Let's define a function (>>=) for this pattern

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x  -> f x
```

- Then the eval function becomes

```
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = eval x >>= \n ->
                 eval y >>= \m ->
                 safediv n m
```

Monad Definition

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

  return = pure
```

- To be a Monad, you must be an Applicative
  - (To be an Applicative, you must be a Functor)
- return is the same as pure — turns a value into a monadic value
- (>>=) or *bind* takes a monadic value and feeds it into a function that takes a normal value, but returns a monadic value. This ultimately returns a monadic value

Motivating Example: SafeDiv

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

  return = pure
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x  -> f x
```

```
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = eval x >>= \n ->
                 eval y >>= \m ->
                 safediv n m
```

Motivating Example: SafeDiv

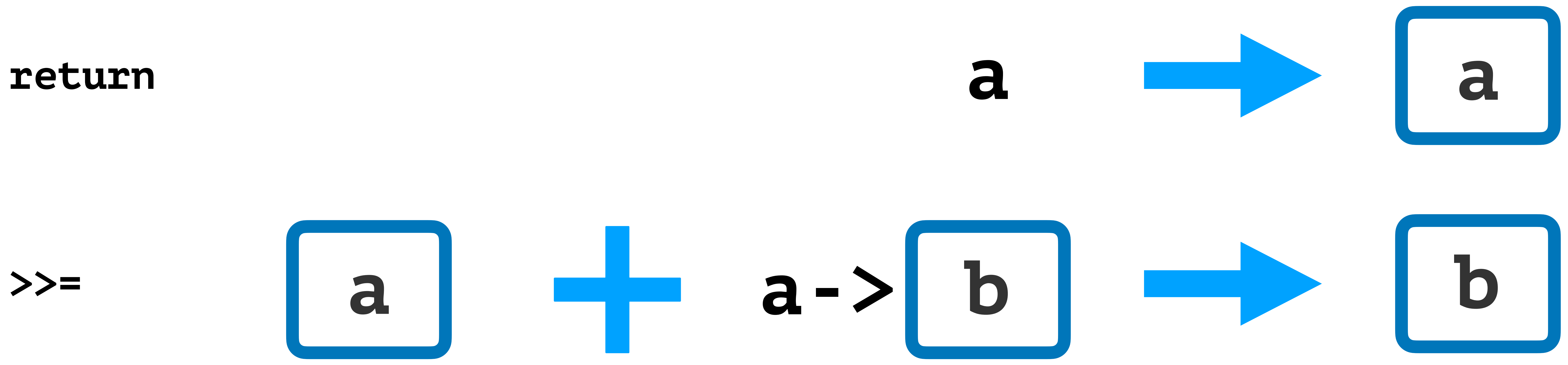
```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

  return = pure
```

```
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = bind
                   (eval x)
                   (\n ->
                       bind
                         (eval y)
                         (\m -> safediv n m))
```

Monad Laws

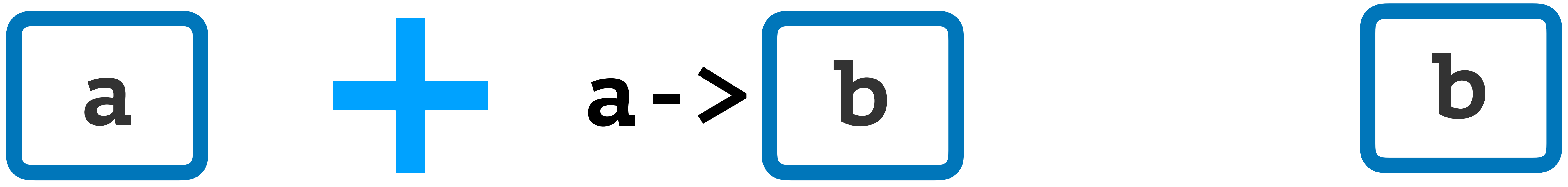
- Monads

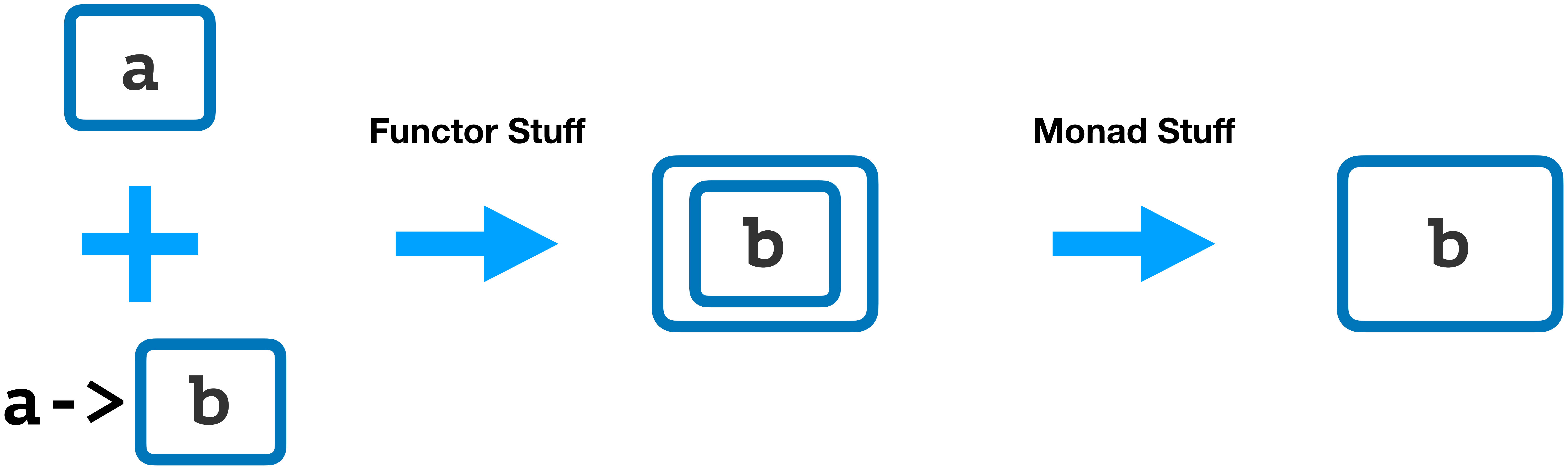


- A type constructor is a monad if it is an instance of the Monad type class that satisfies the following laws
  - (Left identity) return x >>= f = f x
  - (Right identity) mx >>= return = mx
  - (Associativity) (mx >>= f) >>= g = mx >>= (\x -> (f x >>= g))

Monad Laws

>>=





```haskell
eval :: Expr -> Maybe (Maybe (Maybe Int)))
eval (Val n)   = Just n
eval (Div x y) = fmap
                   (\n ->
                       fmap
                         (\m -> safediv n m)
                         (eval y))
                   (eval x)
```

Maybe Monad Instance

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  Fmap f (Just x) = Just (f x)
```

```
instance Monad Maybe where
  return x = Just x

  (>>=) Nothing f = Nothing
  (>>=) (Just x) f = f x
```

List Monad

```haskell
instance Functor [] where
  fmap f [] = []
  fmap f (h:t) = h:(fmap f t)
```

```haskell
instance Monad [] where
  return x = Just x

  (>>=) Nothing f = []
  (>>=) (h:t) f = (f h) ++ (t >>= f)

--(>>=) l f = concat (fmap f l)
```

List Monad Use

```
instance Monad [] where
  return x = Just x

  (>>=) Nothing f = []
  (>>=) (h:t) f = (f h) ++ (t >>= f)
```

```
data Expr = Val Int | OneOf (Expr,Expr) | Plus (Expr,Expr)
```

```
evaluate :: Expr -> [Int]
evaluate (Val i) = return I
evaluate (OneOf (e1,e2)) =
  (evaluate e1) ++ (evaluate e2)
evaluate (Plus (e1,e2)) =
  evaluate e1 >>= \i1 ->
  evaluate e2 >>= \i2 ->
  i1 + i2
```

A Common Monad Pattern

```
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = eval x >>= \n ->
                 eval y >>= \m ->
                 safediv n m
```

```
evaluate :: Expr -> [Int]
evaluate (Val i) = return I
evaluate (OneOf (e1,e2)) =
  (evaluate e1) ++ (evaluate e2)
evaluate (Plus (e1,e2)) =
  evaluate e1 >>= (\i1 ->
  evaluate e2 >>= (\i2 ->
  i1 + i2))
```

Do Notation

```haskell
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = eval x >>= \n ->
                 eval y >>= \m ->
                 safediv n m
```

```haskell
evaluate :: Expr -> [Int]
evaluate (Val i) = return I
evaluate (OneOf (e1,e2)) =
  (evaluate e1) ++ (evaluate e2)
evaluate (Plus (e1,e2)) =
  evaluate e1 >>= (\i1 ->
  evaluate e2 >>= (\i2 ->
  i1 + i2))
```

```haskell
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = do
    n <- eval x
    m <- eval y
    safediv n m
```

```haskell
evaluate :: Expr -> [Int]
evaluate (Val i) = return I
evaluate (OneOf (e1,e2)) =
  (evaluate e1) ++ (evaluate e2)
evaluate (Plus (e1,e2)) = do
  i1 <- evaluate e1
  i2 <- evaluate e2
  i1 + i2
```

ErrJst Monad Instance

```
data ErrJst e a =
    Err e
  | Jst a
```

```
instance Monad ErrJst where
  return x = Jst x

  (>>=) (Err e) _ = Err e
  (>>=) (Jst x) f = f x
```

ErrJst Usage

```
eval :: Expr -> Maybe Int
eval (Val n)   = Jst n
eval (Div x y) = do
    n <- eval x
    m <- eval y
    case safediv n m of
      Nothing -> Err (printf “Divided %d by zero” n)
      Just v -> Jst v
```