Functional Programming

Lecture 12: Advanced Monads

Twan van Laarhoven

5 December 2022

Outline

- Monads for effects
- Laws
- Type classes for effects
- Example: probabilistic programming
- Pure state
- Summary

Monads, Functors, Applicatives

recap from last week

Functor

Last week we introduced

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

fmap applies a pure function to all elements of a container.

Applicative

Last week we introduced

```
class (Functor f) ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

<*>> (pronounced as "apply") applies a container of functions to a container of arguments.

pure wraps a pure value into a container

Monad

Last week we introduced

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

`>>=` (pronounced as "bind") allows you to generate an impure computation based on the pure value that comes out of another impure computation.
The second computation can depend on the result of the first.

`return`/`pure` can 'warp' a pure value into a monadic one.

Maybe instance

```
instance Applicative Maybe where
  pure :: a → Maybe a
  pure x = Just x
  (<*>) :: Maybe (a → b) → Maybe a → Maybe b
  Just g <*> Just x = Just (g x)
  _      <*> _      = Nothing
instance Monad Maybe where
  (>>=) :: Maybe a → (a → Maybe b) → Maybe b
  Nothing >>= _ = Nothing
  Just x  >>= k = k x
```

Exception handling: Nothing represents failure

Kinds: types for types

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

What about types with multiple arguments?

```
data Either a b = Left a | Right b
```

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

What about types with multiple arguments?

```
data Either a b = Left a | Right b

instance Functor Either where
  fmap :: (a → b) → (Either a → Either b) -- WRONG
```

Which types can be an instance of Functor?

We can't make an instance of Functor for ordinary types

```
instance Functor String where
  fmap :: (a → b) → (String a → String b) -- WRONG
```

What about types with multiple arguments?

```
data Either a b = Left a | Right b

instance Functor Either where
  fmap :: (a → b) → (Either a → Either b) -- WRONG
```

The type-checker should disallow these.

Types of types

The type of a type is called its *kind*.

- Normal types have kind $\star$
- Maybe is a type constructor: a function from types to types.
  Its kind is $\star \rightarrow \star$

Types of types

The type of a type is called its *kind*.

- Normal types have kind $\star$
- Maybe is a type constructor: a function from types to types.
  Its kind is $\star \rightarrow \star$

Examples:

```
Int        :: ⋆
[]         :: ⋆ → ⋆
Either     :: ⋆ → ⋆ → ⋆
Either a   :: ⋆ → ⋆
Either a b :: ⋆
```

Note that type constructors can be partially applied.

Constructor classes

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

We know that f a is a type, f a :: ⋆
We know that a is a type, a :: ⋆
So the argument f has kind ⋆ → ⋆.

Constructor classes

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

We know that f a is a type, f a :: ⋆
We know that a is a type, a :: ⋆
So the argument f has kind ⋆ → ⋆.

Type constructors with this kind are:

- []
- Maybe
- Tree
- Either a

Kinds for classes

The kind of classes is Constraint,

$$
\begin{aligned}
\mathsf{Num} &:: \star \to \mathsf{Constraint} \\
\mathsf{Eq} &:: \star \to \mathsf{Constraint} \\
\mathsf{Functor} &:: (\star \to \star) \to \mathsf{Constraint} \\
\mathsf{Applicative} &:: (\star \to \star) \to \mathsf{Constraint} \\
\mathsf{Monad} &:: (\star \to \star) \to \mathsf{Constraint}
\end{aligned}
$$

Reasoning with Monads

and Functors and Applicatives

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Functor laws

Functors are required to satisfy two equational laws:

- Identity
  ```
  fmap id      = id
  ```
- Composition
  ```
  fmap (g . f) = fmap g . fmap f
  ```

Intuition:

- Do not change the structure, only the values.

Applicative laws

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

Instances of Applicative must satisfy the laws

- Identity:
  ```
  pure id <*> xs      = xs
  ```
- Composition
  ```
  fs <*> (gs <*> xs) = (pure (.) <*> fs <*> gs) <*> xs
  ```
- Homomorphism
  ```
  pure f <*> pure x  = pure (f x)
  ```
- Interchange
  ```
  fs <*> pure x      = pure (\f -> f x) <*> fs
  ```

Intuition:

- Like the Functor laws, remember: `fmap f xs = pure f <*> xs`
- `pure` doesn't intervere with `<*>`

Monadic function composition

How to compose functions that have a container/computation/monadic type

$$
\begin{aligned}
f &:: a \rightarrow m\ b \\
g &:: b \rightarrow m\ c
\end{aligned}
$$

For simple pure functions there is

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

For monadic computations

$$
\begin{aligned}
&(>=>) :: \mathrm{Monad}\ m \Rightarrow (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c) \\
&(f >=> g) = \backslash x \rightarrow f\ x \gg= \backslash y \rightarrow g\ y \\
&(<=<) :: \mathrm{Monad}\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c) \\
&(g <=< f)\ x = \mathbf{do}\ \{\ y \leftarrow f\ x;\ g\ y\ \}
\end{aligned}
$$

Monad composition laws

Instances of Monad must satisfy the monad laws

- Left identity

  $$\mathsf{pure} >\!=\!> f = f$$

- Right idententy

  $$f >\!=\!> \mathsf{pure} = f$$

- Associativity

  $$(f >\!=\!> g) >\!=\!> h = f >\!=\!> (g >\!=\!> h)$$

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

Monad laws

Instances of Monad must satisfy the monad laws

- Left identity

```
pure x >>= k              = k x
```

- Right identity

```
mx >>= pure               = mx
```

- Associativity

```
mx >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

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

Monad laws

Instances of Monad must satisfy the monad laws

- Left identity
  `do { y ← pure x; k y .. }` = `do { let y = x; k y .. }`
- Right identity
  `do { x ← mx; pure x }` = `do { mx }`
- Associativity
  `do { y ← do { x ← mx; k x }; h y }` = `do { x ← mx; y ← k x; h y }`

Intuition:

- pure has no effect
- You can substitute nested computations

More effects

back to the evaluator example

Applicative Evaluator

Recall:

$$
\begin{aligned}
&\mathsf{eval} :: (\mathsf{Applicative}\ f) \Rightarrow \mathsf{Expr} \rightarrow f\ \mathsf{Integer} \\
&\mathsf{eval}\ (\mathsf{Lit}\ i) \qquad\quad = \mathsf{pure}\ i \\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{pure}\ (+) <\!\!*\!\!> \mathsf{eval}\ e_1 <\!\!*\!\!> \mathsf{eval}\ e_2 \\
&\mathsf{eval}\ (\mathsf{Mul}\ e_1\ e_2) = \mathsf{pure}\ (*) <\!\!*\!\!> \mathsf{eval}\ e_1 <\!\!*\!\!> \mathsf{eval}\ e_2
\end{aligned}
$$

Recovering the vanilla evaluator

Can we recover

```
evalPure :: Expr → Integer
```

from

```
eval :: (Applicative f) ⇒ Expr → f Integer
```

What should `f` be?

Recovering the vanilla evaluator

Can we recover

```
evalPure :: Expr → Integer
```

from

```
eval :: (Applicative f) ⇒ Expr → f Integer
```

What should f be?

```
type Id a = a
```

Recovering the vanilla evaluator

Can we recover

```
evalPure :: Expr → Integer
```

from

```
eval :: (Applicative f) ⇒ Expr → f Integer
```

What should f be?

```
type Id a = a
```

Type synonyms are not allowed in instances.
Use the *newtype trick*:

```
newtype Id a = Id { fromId :: a }
```

Recovering the vanilla evaluator

Type synonyms are not allowed in instances.
Use the *newtype trick*:

```
newtype Id a = Id { fromId :: a }
```

We have to manually wrap and unwrap

```
Id     :: a → Id a
fromId :: Id a → a
```

Note: **newtype** is like **data**. There is a small technical difference in strictness, but it doesn't matter here.

Recovering the vanilla evaluator

Meet the identity functor

```haskell
newtype Id a = Id { fromId :: a }

instance Functor Id where
  fmap :: (a -> b) -> Id a -> Id b
  fmap f (Id x) = Id (f x)

instance Applicative Id where
  pure :: a -> Id a
  pure x = Id x
  (<*>) :: Id (a -> b) -> Id a -> Id b
  Id f <*> Id x = Id (f x)
```

pure is the identity and <*> is function application

Example evaluation:

```haskell
>>> (+) <$> Id 1 <*>
Id 2
  Id 3
>>> fromId (eval good)
3
```

Recovering the vanilla evaluator

Meet the identity functor

```
newtype Id a = Id { fromId :: a }
instance Functor Id where
  fmap :: (a → b) → Id a → Id b
  fmap f (Id x) = Id (f x)
instance Monad Id where
  (>>=) :: Id a → (a → Id b) → Id b
  Id x >>= mf = mf x
```

pure is the identity and <*> is function application

Example evaluation:

```
>>> (+) <$> Id 1 <*> Id 2
Id 3
>>> fromId (eval good)
3
```

The counter instance

Tracking a value:

```haskell
data Counter a = C a Int    -- a value and a count
  deriving (Show)

instance Functor Counter where
  fmap :: (a -> b) -> Counter a -> Counter b
  fmap f (C x n) = C (f x) n

instance Applicative Counter where
  pure :: a -> Counter a
  pure x = C x 0
  (<*>) :: Counter (a -> b) -> Counter a -> Counter b
  C f n1 <*> C x n2 = C (f x) (n1 + n2)
```

The counter instance

Tracking a value:

```
data Counter a = C a Int      -- a value and a count
  deriving (Show)

instance Monad Counter where
  (>>=) :: Counter a -> (a -> Counter b) -> Counter b
  C x n1 >>= f = let (C y n2) = f x in C y (n1 + n2)
```

Conting as an effect

```haskell
data Counter a = C a Int    -- a value and a count
  deriving (Show)
```

Increment the count:

```haskell
tick :: Counter ()
tick = C () 1
```

`tick` is only called for its effect, not its value.

Example:

```haskell
>>> do{ tick; tick; tick }
(C () 3)
>>> pure "tock"
(C "tock" 0)
```

Counting evaluator, applicative style

to integrate tick we use $\gg$

```
evalC :: Expr → Counter Integer
evalC (Lit i)       = tick ≫ pure i
evalC (Add e₁ e₂) = tick ≫ pure (+) <*> evalC e₁ <*> evalC e₂
evalC (Mul e₁ e₂) = tick ≫ pure (*) <*> evalC e₁ <*> evalC e₂
```

Example evaluation:

```
⋙ evalC (Add (Lit 7) (Add (Lit 4) (Lit 2)))
C 13 5
```

Combining effects

Counting + failure

Counting uses

```
tick :: Counter ()
```

Exception handling uses

```
Nothing :: Maybe a
safediv :: Integer → Integer → Maybe Integer
```

Nondeterminism uses

```
choice :: [a] → [a] → [a]
```

How to combine effects?

A type class for counting

```
class Monad m ⇒ MonadCount m where
  tick :: m ()

instance MonadCount Counter where
  tick = tickCounter

evalC :: MonadCount m ⇒ Expr → m Integer
evalC (Lit i)       = tick >> pure i
evalC (Add e1 e2) = tick >> (+) <$> evalC e1 <*> evalC e2
evalC (Mul e1 e2) = tick >> (*) <$> evalC e1 <*> evalC e2
```

A type class for failure

```haskell
class Monad m => MonadFail m where
  fail :: String -> m a  -- error message on failure
instance MonadFail Maybe where
  fail _ = Nothing

safediv :: MonadFail m => Integer -> Integer -> m Integer
safediv x y
  | y == 0    = fail "division by zero"
  | otherwise = pure (x `div` y)
```

Using multiple effects

```
eval :: (MonadFail m, MonadCount m) ⇒ Expr → m Integer
eval (Lit i)       = tick ≫ pure i
eval (Add e₁ e₂) = tick ≫ (+) <$> eval e₁ <*> eval e₂
eval (Mul e₁ e₂) = tick ≫ (*) <$> eval e₁ <*> eval e₂
eval (Div e₁ e₂) = do
  tick
  v₁ ← eval e₁
  v₂ ← eval e₂
  safediv v₁ v₂
```

Multiple effects

$$\textbf{newtype}\ \mathsf{MCounter}_1\ a = \mathsf{MC}\ (\mathsf{Maybe}\ (a, \mathsf{Int}))$$
$$\textbf{newtype}\ \mathsf{MCounter}_2\ a = \mathsf{MC}\ (\mathsf{Maybe}\ a)\ \mathsf{Int}$$

What is the difference?

Multiple effects

```
newtype MCounter1 a = MC (Maybe (a, Int))
newtype MCounter2 a = MC (Maybe a) Int
```

What is the difference?

- $\mathsf{MCounter}_1$ only contains a count if the computation succeeds
- $\mathsf{MCounter}_2$ always contains a count

Multiple effects

```haskell
newtype MCounter1 a = MC { unMC :: Maybe (a, Int) }
instance Applicative MCounter1 where
  pure :: a → MCounter1 a
  pure x = MC (Just (x,0))
instance Monad MCounter1 where
  (>>=) :: MCounter1 a → (a → MCounter1 b) → MCounter1 b
  mx >>= k = MC $ do -- working in the Maybe monad
    (x, n1) ← unMC mx
    (y, n2) ← unMC (k x)
    pure (y, n1 + n2)
```

$MCounter_2$ is left as an exercise.

Multiple effects

```
newtype MCounter1 a = MC { unMC :: Maybe (a, Int) }
instance MonadCount MCounter1 where
  tick :: MCounter1 ()
  tick = MC $ Just ((), 1)
instance MonadFail MCounter1 where
  fail :: String -> MCounter1 a
  fail _msg = MC Nothing
```

Case study

the Monty Hall problem

Monty Hall problem

Suppose you are on a game show, and you are given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2 instead?" Is it to your advantage to switch your choice?

- Probabilistic programming: computing with probabilities
- Two strategies: stick to original choice, or switch choice
- Strategies as programs

Representing probabilities

Discrete probability distribution (probability mass function)

```
type Prob = Rational
newtype Dist a = D { fromD :: [(a,Prob)] }
```

Invariant: probabilities of a distribution dist sum up to 1

```
sum [p | (e,p) ← fromD dist] == 1
```

(ideally, each event occurs exactly once, exercise: define

```
norm :: Ord a ⇒ Dist a → Dist a)
```

Uniform distribution (events have the same probability)

```
uniform :: [a] → Dist a
uniform xs = D [(x, 1 % n) | x ← xs]
  where n = genericLength xs
```

Events

Sum of probabilities

```
probability :: (a → Bool) → Dist a → Prob
probability ev dist = sum [ p | (x,p) ← fromD dist, ev x]
```

for example, the probability of getting at least 5 when throwing 1 die is

```
⋙ probability (≥ 5) die
1 % 3
```

where

```
die = uniform [1..6]
```

The probability distribution monad

```
instance Functor Dist where
  fmap :: (a → b) → Dist a → Dist b
  fmap f (D d) = D [(f x, p) | (x,p) ← d]
instance Applicative Dist where
  pure :: a → Dist a
  pure x = uniform [x]
  (<*>) :: Dist (a → b) → Dist a → Dist b
  D fd <*> D xd = D [(f x, p1*p2) | (f,p1) ← fd, (x,p2) ← xd]
instance Monad Dist where
  (>>=) :: Dist a → (a → Dist b) → Dist b
  D xd >>= k = D [(y, p1*p2) | (x,p1) ← xd, (y,p2) ← fromD (k x)]
```

(exercise: is the invariant always satisfied?)

Rolling dice

A pair of dice, sum of pips (applicative and monadic style)

```
rollA , rollM :: Dist Int
rollA = pure (+) <*> die <*> die
rollM = do { a ← die ; b ← die ; pure (a + b) }
```

Rolling a pair of dice,

```
>>> rollA
[(2,1 % 36),(3,1 % 36),(4,1 % 36),(5,1 % 36),...,(11,1 % 36),(12,1 % 36)]
>>> norm it
[(2,1 % 36),(3,1 % 18),(4,1 % 12),(5,1 % 9),(6,5 % 36),(7,1 % 6),
 (8,5 % 36),(9,1 % 9),(10,1 % 12),(11,1 % 18),(12,1 % 36)]
```

Rolling dice

Multiple dice, collecting all possibilities

```
dice :: Int → Dist [Int]
dice n = replicateM n die
```

Example

```
>>> dice 2
[([1,1],1 % 36),([1,2],1 % 36),...,([6,5],1 % 36),([6,6],1 % 36)]
>>> dice 4
[([1,1,1,1],1 % 1296),([1,1,1,2],1 % 1296),...,([6,6,6,6],1 % 1296)
```

probability of rolling Yahtzee

```
>>> probability (\(x:xs) → all (==x) xs) (dice 5)
1 % 1296
```

Back to Monty Hall

We model the game show as follows

```
data Outcome = Win | Lose  deriving (Eq, Ord, Show)
data Door = No1 | No2 | No3 deriving (Eq, Enum)
doors = [No1 .. No3]
```

Host hides the car behind one of the doors; you pick one

```
hide, pick :: Dist Door
hide = uniform doors
pick = uniform doors
```

Host teases you by opening one of the doors

```
tease h p = uniform (doors \\ [h, p])
```

Back to Monty Hall

Whole game parametrized by strategy

```
play :: (Door → Door → Dist Door) → Dist Outcome
play strategy = do
  h ← hide           -- host hides the car behind door h
  p ← pick           -- you pick door p
  t ← tease h p      -- host teases you with door t (/= h, p)
  s ← strategy p t   -- you choose, based on p and t
  pure (if s == h then Win else Lose)
```

You win iff your choice s equals h

Back to Monty Hall

The two strategies

```
stick, switch :: Door → Door → Dist Door
stick  p t = pure p
switch p t = uniform (doors \\ [p, t])
```

Which is better?

```
>>> norm (play stick)
D [(Win; 1 % 3); (Lose; 2 % 3)]
>>> norm (play switch)
D [(Win; 2 % 3); (Lose; 1 % 3)]
```

Switching doubles (!) your chance of winning

More effects

Global state

```haskell
type GlobalState -- whatever is needed for your program
class Monad m ⇒ MonadState m where
  getState :: m GlobalState
  putState :: GlobalState → m ()
```

Usage:

```haskell
type GlobalState = [String]
-- give all your children a unique name from a big list
freshName :: MonadState m ⇒ m String
freshName = do
  n:ns ← getState
  putState ns
  pure n
```

Mutable state without IO

A pure computation that manipulates the global state

- takes state as extra input,
- produces state as extra output.

So we need a type like

```haskell
type StateFull a = GlobalState → (a, GlobalState)
```

The State monad

```
newtype State a = St { runSt :: GlobalState → (a,GlobalState) }
instance Functor State where
  fmap f sx = St $ \s₁ → let (x,s₂) = runSt sx s₁ in (f x, s₂)
instance Monad State where
  return x = St $ \s → (x, s)
  sx >>= k = St $ \s₁ → let (x, s₂) = runSt sx s₁
                             (y, s₃) = runSt (k x) s₂
                         in  (y, s₃)
instance MonadState State where
  getState = St $ \s → (s, s)
  putState newState = St $ \_ → ((),newState)
```

Using the state monad

```
newtype State a = St { runSt :: GlobalState → (a, GlobalState) }
type GlobalState = [String]
freshName :: MonadState m ⇒ m String
```

Usage:

```
>>> let tree = Bin (Tip ()) (Tip ())
>>> let names = ["x","y","z"]
>>> runSt names (mapM (const freshName) tree)
Bin (Tip "x") (Tip "y")
```

Restricting IO

IO actions can do many (evil) things.

```haskell
class Monad m => MonadConfig m where
  readConfigFile :: m String
instance MonadConfig IO where
  readConfigFile = readFile "config.json"
untrusted :: MonadConfig m => Int -> m Int
```

Can `untrusted` do arbitrary IO things?

Take away

Summary

- Haskell allows you to implement your own computational effect or combination of effects (how cool is this?)
- Use type classes to specify the allowed effects (separate interface from implementation)