Functional Programming

Lecture 12: Monads

Twan van Laarhoven

6 December 2021

Outline

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

Handling failure

The limits of Applicative

Structure of applicative computation

Remember: list instance of `Applicative` computes all combinations

What is the length of

```
flurbify <$> [1..10] <*> "hello" <*> replicate 2 Banana
```

Structure of applicative computation

Remember: list instance of Applicative computes all combinations

What is the length of

```
flurbify <$> [1..10] <*> "hello" <*> replicate 2 Banana
```

Answer:

$10 * 5 * 5 = 2500$

This doesn't depend on the values or the function!

In general:

- the 'shape' of the output can not depend on values in the container
- the arguments to `<*>` are independent

Last week: Evaluator

Evaluator in an applicative style

```
evalA :: (Applicative f) ⇒ Expr → f Integer
evalA (Lit i)       = pure i
evalA (Add e1 e2) = pure (+) <*> evalA e1 <*> evalA e2
evalA (Mul e1 e2) = pure (*) <*> evalA e1 <*> evalA e2
evalA (Div e1 e2) = pure div <*> evalA e1 <*> evalA e2
```

- Counter for counting steps.
- Maybe for exception handling?

Maybe for handling failure

Recall:

```
data Maybe a = Nothing | Just a
```

- Just x is a successful computation (with result x)
- Nothing denotes failure.

Make a safe version of `div` that fails if its second argument is zero

```
safediv :: Integer → Integer → Maybe Integer
safediv x y
  | y == 0    = Nothing
  | otherwise = Just (x `div` y)
```

Exception handling evaluator

Safe version of `div` that fails if its second argument is zero

```
safediv :: Integer → Integer → Maybe Integer
safediv x y
  | y == 0    = Nothing
  | otherwise = Just (x `div` y)
```

How to modify the interpreter to use `safediv`?

```
evalA :: Integer → Maybe Integer
...
evalA (Div e1 e2) = pure div <*> evalA e1 <*> evalA e2 -- ??
```

Exception handling evaluator

Safe version of `div` that fails if its second argument is zero

```
safediv :: Integer → Integer → Maybe Integer
safediv x y
  | y == 0    = Nothing
  | otherwise = Just (x `div` y)
```

How to modify the interpreter to use `safediv`?

```
evalA :: Integer → Maybe Integer
...
evalA (Div e₁ e₂) = pure safediv <*> evalA e₁ <*> evalA e₂ -- ??
```

But: `pure safediv <*> evalA e₁ <*> evalA e₂` has type `Maybe (Maybe Integer)` and not `Maybe Integer`.

Exception handling evaluator

Problem: `pure safediv <*> evalA e1 <*> evalA e2` has type `Maybe (Maybe Integer)` and not `Maybe Integer`.

Solution: introduce a combinator that allows us to join/concatenate a whole container of containers together.
In other words: join a value of type `f (f a)` into a value of type `f a`

```
class (Applicative f) => Joinable f where
  join :: f (f a) -> f a
```

Maybe instance

The instance for the Maybe type is

```
instance Joinable Maybe where
  -- join :: Maybe (Maybe a) → Maybe a
  join Nothing  = Nothing
  join (Just x) = x
```

join peels off the outermost Just

Exception handling evaluator

```
evalA :: Expr → Maybe Integer
evalA (Lit i)      = pure i
evalA (Add e1 e2) = pure (+) <*> evalA e1 <*> evalA e2
evalA (Mul e1 e2) = pure (*) <*> evalA e1 <*> evalA e2
evalA (Div e1 e2) = join (pure safediv <*> evalA e1 <*> evalA e2)
```

Monads

in Haskell

Haskell's solution: Monad

`Joinable` does not exist in Haskell

Instead Haskell introduces

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

$\gg=$ (pronounced as "bind") allows you to generate an impure computation on the base of the pure value of another impure computation.
The second computation can depend on the result of the first.

Have you seen these combinators before?

What is the difference between `return` and `pure`?

Maybe instance

```haskell
instance Monad Maybe where
  return :: a → Maybe a
  return x = Just x
  (>>=) :: Maybe a → (a → Maybe b) → Maybe b
  Nothing >>= _ = Nothing
  Just x  >>= k = k x
```

Exception handling evaluator

Evaluator using the Monad class

```
evalA :: Expr → Maybe Integer
evalA (Lit i)       = pure i
evalA (Add e1 e2) = pure (+) <*> evalA e1 <*> evalA e2
evalA (Mul e1 e2) = pure (*) <*> evalA e1 <*> evalA e2
evalA (Div e1 e2) = pure div <*> evalA e1 <*> (evalA e2 >>= guarded (/= 0))

guarded :: (a → Bool) → a → Maybe a
guarded pred x
  | pred x      = Just x
  | otherwise = Nothing
```

Original evaluator, monadically

We can also write the interpreter in a monadic style

```
evalM :: (Monad m) ⇒ Expr → m Integer
evalM (Lit i) = return i
evalM (Add e1 e2) =
  evalM e1 >>= \x1 →
  evalM e2 >>= \x2 →
  return (x1 + x2)
evalM (Div e1 e2) =
  evalM e1 >>= \x1 →
  evalM e2 >>= \x2 →
  safediv x1 x2
```

(other cases omitted for reasons of space)

do notation

Haskell provides a special notation for monadic expressions

$$\begin{array}{l} \textbf{do} \\ \quad x_1 \leftarrow m_1 \\ \quad x_2 \leftarrow m_2 \\ \quad \quad \ldots \\ \quad x_n \leftarrow m_n \\ \quad f\ x_1\ x_2\ \ldots\ x_n \end{array} \quad = \quad \begin{array}{l} m_1 \gg\!= \backslash x_1 \rightarrow \\ m_2 \gg\!= \backslash x_2 \rightarrow \\ \quad \ldots \\ m_n \gg\!= \backslash x_n \rightarrow \\ f\ x_1\ x_2\ \ldots\ x_n \end{array}$$

Note: do is layout sensitive.

See also lecture 10

do notation (continued)

We can ignore the result with

$$(\gg) :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b$$
$$a \gg b = a \gg\!= \backslash\_ \rightarrow b$$

Make local declarations with **let** statements

$$\begin{array}{l} \textbf{do} \\ \quad x_1 \leftarrow m_1 \\ \quad \textbf{let } x_2 = \text{nonMonadicCode } x_1 \\ \quad x_3 \leftarrow m_3 \\ \quad m_4 \\ \quad f\ x_1\ x_2\ x_3 \end{array} = \begin{array}{l} m_1 \gg\!= \backslash x_1 \rightarrow \\ \textbf{let } x_2 = \text{nonMonadicCode } x_1 \textbf{ in} \\ m_3 \gg\!= \backslash x_3 \rightarrow \\ m_4 \gg \\ f\ x_1\ x_2\ x_3 \end{array}$$

Have we been here before?

```
mapM       :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_      :: Monad m => (a -> m b) -> [a] -> m ()
foldM      :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
filterM    :: Monad m => (a -> m Bool) -> [a] -> m [a]
replicateM :: Monad m => Int -> m a -> m [a]
```

Have we been here before?

$$\mathsf{mapM} :: \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b]$$
$$\mathsf{mapM\_} :: \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to [a] \to m\ ()$$
$$\mathsf{foldM} :: \mathsf{Monad}\ m \Rightarrow (b \to a \to m\ b) \to b \to [a] \to m\ b$$
$$\mathsf{filterM} :: \mathsf{Monad}\ m \Rightarrow (a \to m\ \mathsf{Bool}) \to [a] \to m\ [a]$$

$$\begin{aligned}
&\mathsf{replicateM} :: \mathsf{Monad}\ m \Rightarrow \mathsf{Int} \to m\ a \to m\ [a] \\
&\mathsf{replicateM}\ 0\ m = \mathsf{return}\ [] \\
&\mathsf{replicateM}\ n\ m = (:) \mathbin{<\$>} m \mathbin{<*>} \mathsf{replicateM}\ (n-1)\ m
\end{aligned}$$

Have we been here before?

```
mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
replicateM :: Monad m => Int -> m a -> m [a]
replicateM 0 m = return []
replicateM n m = (:) <$> m <*> replicateM (n-1) m

join :: Monad m => m (m a) -> m a
join mmx = do { mx <- mmx; mx }
```

Monadic function composition

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

```
f :: a → m b
g :: b → m c
```

For simple pure functions there is `(.) :: (b → c) → (a → b) → (a → c)`.

For monadic computations

```
(>=>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
(f >=> g) = \x → f x >>= \y → g y
(<=<) :: Monad m ⇒ (b → m c) → (a → m b) → (a → m c)
(g <=< f) x = do { y ← f x; g y }
```

Reasoning with Monads

and Functors and Applicatives

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

Instances of Applicative must satisfy the laws

- Identity:

  ```
  pure id <*> v = v
  ```

- Composition

  ```
  u <*> (v <*> w) = (pure (.) <*> u <*> v) <*> w
  ```

- Homomorphism

  ```
  pure f <*> pure x = pure (f x)
  ```

- Interchange

  ```
  u <*> pure y = pure (\g -> g y) <*> u
  ```

Intuition:

- pure doesn't intervere with <*>
- <*> is associative (except for the types)

Monad laws

Instances of Monad must satisfy the monad laws

- Left identity

  ```
  return a >>= k = k a
  ```

- Right identty

  ```
  m >>= return = m
  ```

- Associativity

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

Monad laws

Instances of Monad must satisfy the monad laws

- Left identity
  `do { x ← return a; k x .. } = do { let x = a; k x .. }`
- Right identty
  `do { x ← m; return x } = do { m }`
- Associativity
  `do { y ← do { x ← m; k x }; h y } = do { x ← m; y ← k x; k x }`

Intuition:

- `return` has no effect
- You can substitute nested computations

Combining effects

Counting + failure

Counting uses

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

Exception handling uses

```
Nothing  :: Maybe a
safediv  :: Integer → Integer → Maybe Integer
guarded  :: (a → Bool) → a → Maybe a
```

How to do both?

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 *> pure (+) <*> evalC e1 <*> evalC e2
evalC (Mul e1 e2) = tick *> pure (*) <*> evalC e1 <*> evalC e2
evalC (Div e1 e2) = tick *> pure div <*> 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 = return (x `div` y)
```

Multiple effects

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

What is the difference?

Multiple effects

```haskell
data MCounter1 a = MC (Maybe (a, Int))
data 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

```
data MCounter₁ a = MC { unMC :: Maybe (a, Int) }
instance Monad MCounter₁ where
  return :: a → MCounter₁ a
  return x = MC (Just (x,0))
  (>>=) :: 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)
    return (y, n₁ + n₂)
```

MCounter_2 is left as an exercise.

Multiple effects

```haskell
data 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
```

Effectful interpreter

```
evalMC :: (MonadCount m, MonadFail m) ⇒ Expr → m Integer
evalMC (Lit i)       = tick *> pure i
evalMC (Add e₁ e₂) = tick *> pure (+) <*> evalMC e₁ <*> evalMC e₂
evalMC (Mul e₁ e₂) = tick *> pure (*) <*> evalMC e₁ <*> evalMC e₂
evalMC (Div e₁ e₂) = do
  tick
  x₁ ← evalMC e₁
  x₂ ← evalMC e₂
  safediv x₁ x₂
```

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 a = uniform [a]
  (<*>) :: 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 ; return (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

```haskell
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

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

Host teases you by opening one of the doors

```haskell
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
  return (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 = return 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

```haskell
concatMap :: (a → [b]) → [a] → [b]
concatMap f xs = concat (map f xs)
```

List instance

```haskell
instance Monad [] where
  return :: a → [a]
  return x = [x]
  (>>=) :: [a] → (a → [b]) → [b]
  (>>=) = flip concatMap
```

Compare

```haskell
do x ← xs
   y ← ys
   return (x + y)
```

```haskell
[ x + y | x ← xs, y ← ys]
```

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
nameMyChild :: MonadState m ⇒ m String
nameMyChild = do
  n:ns ← getState
  putState ns
  return 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)
```

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

- Applicative functors and monads allow you to abstract over patterns of computations (effects)
- A small hierarchy in order of expressiveness:
  - functor
  - applicative functor
  - monad
- 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)