More monads

Lecture 7 of CSE 3100
Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

Lecture plan

- The `Either` monad
- The list monad
- The `State` monad
- The `Parser` monad
- The monad laws

Recap: the `Monad` type class

`Monad` is a type class with functions `return` and `(>>=)` ('bind'):

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

A monadic action `x :: m a` can be seen as a computation that can perform some side effects before returning a value of type `a`.

Examples of monads: `Maybe`, `IO`.

The `Either` monad

We can see `Either` as a generalization of `Maybe`:

- `Right` takes the role of `Just`
- `Left err` is a more informative version of `Nothing`

```
instance Monad (Either e) where
  return x = Right x
  Left err >>= f = Left err
  Right x  >>= f = f x
```

The list monad

We can see `[]` as another generalization of `Maybe`:

- `[x]` takes the role of `Just x`
- `[]` takes the role of `Nothing`
- Functions can have multiple outputs

```
instance Monad [] where
  return x = [x]
  xs >>= f = [y | x <- xs, y <- f x]
```

Using the list monad

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x, y)
```

Compare this with list comprehensions:

```
pairs xs ys =
  [ (x , y) | x <- xs
            , y <- ys ]
```

You've been using the list monad all this time!

The State monad

Example: Generating random numbers

How to generate random numbers in Haskell?

```haskell
-- chosen by fair dice roll
-- guaranteed to be random.
randomNumber :: Int
randomNumber = 4
```

…not like that!

Example: Generating random numbers

Solution. Write a pure function that takes a random seed as input.

```
import System.Random

randomNumber :: StdGen -> (Int, StdGen)
randomNumber = random

> randomNumber (mkStdGen 100)
(9216477508314497915, StdGen{...})
```

Example: Generating random numbers

Exercise. Roll 3 six-sided dice & add results.

Example: Generating random numbers

Exercise. Roll 3 six-sided dice & add results.

```
roll :: StdGen -> (Int, StdGen)
roll gen = let (x, newGen) = random gen
           in  (x `mod` 6, newGen)

roll3 :: StdGen -> (Int, StdGen)
roll3 gen0 =
  let (die1, gen1) = roll gen0
      (die2, gen2) = roll gen1
      (die3, gen3) = roll gen2
  in  (die1+die2+die3, gen3)
```

Can't we do better??

The State monad

```haskell
newtype State s a = State (s -> (a,s))

get :: State s s
get = State (\st -> (st,st))

put :: s -> State s ()
put st = State (\_ -> ((),st))
```

Rolling dice with `State`

```haskell
randomInt :: State StdGen Int
randomInt = State random

roll :: State StdGen Int
roll = do
  x <- randomInt
  return (x `mod` 6)

roll3 :: State StdGen Int
roll3 = do
  die1 <- roll
  die2 <- roll
  die3 <- roll
  return (die1+die2+die3)
```

Functor and applicative for State

```
instance Functor (State s) where
  fmap f (State h) =
    State (\oldSt ->
      let (x, newSt) = h oldSt
      in  (f x, newSt)          )

instance Applicative (State s) where
  pure x = State (\st -> (x,st))
  State g <*> State h =
    State (\oldSt ->
      let (f, newSt1) = g oldSt
          (x, newSt2) = h newSt1
      in  (f x, newSt2)          )
```

Binding the state

```haskell
runState :: State s a -> s -> (a,s)
runState (State h) = h

instance Monad (State s) where
  return x = pure x

  State h >>= f =
    State (\oldSt ->
      let (x, newSt) = h oldSt
      in  runState (f x) newSt)
```

Reader and Writer monads

The reader monad gives access to an extra input of some type `r`:

```
newtype Reader r a = Reader (r -> a)
```

The writer monad allows writing some output of type `w`:

```
newtype Writer w a = Writer (w, a)
```

See exercises on Weblab!

Monadic parsing

What is a parser?

At a basic level, a parser turns strings into objects of some type:

```
type Parser a = String -> a

item :: Parser Char
item (x:[]) = x
item _      = error "Parse failed!"
```

Problems:

- No option for graceful failure
- Hard to compose parsers

A better parser

```
type Parser a =
  String -> [(a,String)]

item :: Parser Char
item (x:xs) = [(x,xs)]
item []     = []
```

- Parsing returns a *list* of possible parses
- Each parse comes with a ‘remainder’ of the string for further parsing

A parser of things

is a function from strings

to lists of pairs

of things and strings!

A monadic parser

We wrap `Parser` in a `newtype` to make it into a monad:

```
newtype Parser a =
  Parser (String -> [(a,String)])
parse :: Parser a -> String -> [(a,String)]
parse (Parser f) = f

instance Functor Parser where ...
instance Applicative Parser where ...
instance Monad Parser where ...
```

See book for implementation of instances.

Writing monadic parsers

We can now make use of `do` notation to write parsers:

```
three :: Parser (Char, Char)
three = do
  c1 <- item
  c2 <- item
  c3 <- item
  return (c1,c3)
```

Writing monadic parsers

We can now make use of `do` notation to write parsers:

```
word :: Parser String
word = do
  c <- item
  if (isSpace c) then
    return ""
  else do
    cs <- word
    return (c:cs)
```

Picky parsers

We can define a parser `empty` that always fails:

```
empty :: Parser a
empty = Parser []
```

This is useful to write parsers that only succeed when some property is satisfied:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- char
           if p x then return x else empty

digit :: Parser Char
digit = sat isDigit
```

Choosing between parses

We can combine two parsers by using the second one if the first one fails:

```
(<|>) :: Parser a -> Parser a -> Parser a
(Parser f <|> Parser g) = Parser
  (\inp -> case f inp of
    []     -> g inp
    result -> result)

-- Parsing an optional thing
maybeP :: Parser a -> Parser (Maybe a)
maybeP p = fmap Just p <|> pure Nothing
```

Parsing several things

`some` repeats a parser one or more times.

`many` repeats a parser zero or more times.

```
some many :: Parser a -> Parser [a]
some x = pure (:) <*> x <*> many x
many x = some x <|> pure []

nat :: Parser Int
nat = do xs <- some digit
         return (read xs)
```

Live coding: parsing boolean expressions

Assignment. Develop a parser for the following grammar:

$$
\begin{aligned}
\mathit{expr} ::=\; & \mathit{atom} \\
\mid\; & \texttt{true} \\
\mid\; & \texttt{false} \\
\mid\; & \mathit{expr} \mathbin{\&\&} \mathit{expr} \\
\mid\; & \mathit{expr} \mathbin{||} \mathit{expr} \\
\mid\; & \sim \mathit{expr} \\
\mid\; & (\mathit{expr})
\end{aligned}
$$

where *atom* can be any string of letters.

The Monad laws

Type class laws

Most Haskell type classes have one or more laws that instances should satisfy.

These laws are not checked by the compiler, but they form a contract between Haskell programmers.

So **check**[1] **that your implementation satisfies the laws when implementing an instance!**

---

[1] for example, using a QuickCheck property

Example: Laws of `Eq`

- Reflexivity: `x == x = True`
- Symmetry: `(x == y) = (y == x)`
- Transitivity: If `(x == y && y == z) = True` then `x == z = True`
- Substitutivity: If `x == y = True` then `f x == f y = True`[2]
- Negation: `x /= y = not (x == y)`

[2] where `f :: a -> b` and `a` and `b` are both instances of `Eq`.

The functor laws

`fmap f` applies `f` to each value stored in the container, but should *leave the structure of the container unchanged*.

This is expressed formally by the functor laws:

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

A bogus instance of `Functor`

```
instance Functor Tree where
  fmap f (Leaf x)   = Leaf (f x)
  fmap f (Node l r) =
    Node (fmap f r) (fmap f l)
```

A bogus instance of `Functor`

```
instance Functor Tree where
  fmap f (Leaf x)     = Leaf (f x)
  fmap f (Node l r) =
    Node (fmap f r) (fmap f l)
```

This does not satisfy the law `fmap id = id`:

`fmap id (Node (Leaf 1) (Leaf 2))`

= `Node (Leaf 2) (Leaf 1)`

≠ `id (Node (Leaf 1) (Leaf 2))`

The four laws of `Applicative`

```
pure id <*> x = x

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

mf <*> pure y
  = pure (\g -> g y) <*> mf

x <*> (y <*> z)
  = (pure (.) <*> x <*> y) <*> z
```

The Monad Laws

Monad law #1: Left identity

```
return x >>= f  =  f x
```

Intuition. We can remove `return` statements in the middle of a `do`-block.

```
do
   ...
   y <- return x
   foo y
```

```
do
   ...
   f x
```

Monad law #2: Right identity

```haskell
mx >>= (\x -> return x)   =   mx
```

Intuition. We can eliminate `return` at the end of a `do`-block.

```haskell
do
   ...
   x <- mx
   return x
```

```haskell
do
   ...
   mx
```

Monad law #3: The associativity law

```haskell
(mx >>= f) >>= g
=
mx >>= (\x -> (f x >>= g))
```

Intuition. We can 'flatten' nested `do`-blocks.

```haskell
do
  y <- do x <- mx
          f x
  g y
```

```haskell
do
  x <- mx
  y <- f x
  g y
```

What's next?

Next lecture: Laziness and infinite data

To do:

- Read the book:
    - Today: section 13.1-13.8
    - Next lecture: 15.1-15.5, 15.7
- Start on week 4 exercises on Weblab