Teoria das Categorias para Programadores

Fabrício Olivetti de França

17 de Agosto de 2019

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

Topics

1. Monads
2. Exemplos de Monads
3. Comonads
4. Atividades para Casa

Monads

Monads

there are entire subcultures of young men these days who just hang out online waiting for someone to ask a question about monads

— Monoid Mary (@argumatronic) 4 de março de 2019

Monads

O uso de Monads gerou diversos mitos entre os programadores por conta de seu uso em programação (não necessariamente em Haskell).

Monads

Isso motivou a criação de diversos tutoriais traçando uma analogia de Monads com outros conceitos fora da computação ou com enfoque em uma de suas aplicações práticas.

Monads

De acordo com o Haskell wiki e sumarizado no texto What I wish I knew when learning Haskell, um Monad **não**:

- Define funções impuras
- Cria e gerencia estados de sistema
- Permite sequenciamento imperativo
- Define IO
- É dependente de avaliação preguiçosa
- É uma porta dos fundos para efeito colateral
- É uma linguagem imperativa dentro do Haskell
- Necessita de conhecimento de matemática abstrata para entender
- Exclusivo do Haskell

Monads

A dificuldade em entender Monads se dá por conta do pensamento imperativo que costuma ser nosso primeiro contato com programação.

Monads

Vimos anteriormente o caso do nosso `Writer w a` que alterava a saída de nossas funções com um *embelezamento*, de forma a evitar transformar uma função pura em impura.

Monads

Essa estrutura nos obrigou a criar um operador de composição específico para esses casos, gerando a Categoria Kleisli, que detalharemos em seguida.

Categoria Kleisli

Relembrando nosso tipo `Writer` em Haskell:

```haskell
data Writer w a = Writer (a, w)
```

Categoria Kleisli

Podemos criar uma instância de Functor fazendo:

```
instance Functor (Writer w) where
  fmap f (Writer (a,w)) = Writer (f a, w)
```

Categoria Kleisli

Utilizando esse tipo como saída de nossas funções temos que uma função `f :: a -> b` se torna uma função `f :: a -> Writer w b`.

Categoria Kleisli

Fazendo `Writer w = m`, temos o padrão:

```
f :: a -> m b
```

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

Categoria Kleisli

Podemos então pensar na categoria Kleisli ($\mathbf{K}$) como:

Partindo de uma categoria C e um endofunctor m, a categoria $\mathbf{K}$ possui os mesmos objetos de C, mas com morfismos $a \to b$ sendo mapeados para $a \to mb$ na categoria C.

Categoria Kleisli

Para ser uma categoria precisamos de um operador de composição (já temos o nosso peixe) e um morfismo identidade $a \to a$, que na categoria C é, na verdade, $a \to ma$.

Categoria Kleisli

Com isso, dizemos que m é um Monad se:

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

Categoria Kleisli

E apresenta as propriedades:

```
(f >=> g) >=> h = f >=> (g >=> h) = f >=> g >=> h
f >=> return = return >=> f = f
```

Em outras palavras, um Monad define como compor funções *embelezadas* e como colocar tipos dentro de um contexto.

Categoria Kleisli

Como ficaria a instância completa do nosso Monad `Writer w`?

```haskell
instance Monoid w => Monad (Writer w) where
  f >=> g = \a -> let Writer (b, s)  = f a
                      Writer (c, s') = g b
                  in  Writer (c, s `mappend` s')
  return a = Writer (a, mempty)
```

Indicando que `w` deve ser um Monoid, generalizamos a definição para outros tipos além de `String`.

Dissecando o Peixe

Podemos perceber um padrão dentro do nosso operador >=> que nos ajudará a simplificá-lo:

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \a -> let mb = f a
                in  ???
```

Dissecando o Peixe

O que nos resta fazer é aplicar `g` em `mb` de tal forma a gerar um `m c`:

```
f >=> g = \a -> let mb = f a
                in  h mb g

h :: m b -> (b -> m c) -> m c
```

Dissecando o Peixe

Vamos criar o seguinte operador:

```
(>>=) :: m a -> (a -> m b) -> m b
```

Dissecando o Peixe

Que no caso do Monad `Writer w` fica:

```
(Writer (a,w)) >>= f = let Writer (b, w') = f a
                       in  Writer (b, w `mappend` w')
```

Dissecando o Peixe

Tornando a definição do operador >=> como:

```
f >=> g = \a -> let mb = f a
                in  mb >>= g
```

Dissecando o Peixe

Tornando a definição do operador >=> como:

```
f >=> g = \a -> (f a) >>= g
```

Muito mais simples!

Dissecando o Peixe

O operador >>= é conhecido como `bind` e define outra forma de instanciar um Monad:

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

Dissecando o Peixe

Lembrando que um Monad também é um Functor, ele permite o uso de `fmap`.

Dissecando o Peixe

Se tivermos duas funções:

```
f :: a -> m b
```

```
fmap :: (a -> c) -> m a -> m c
```

Ao aplicar `fmap f ma` sendo `ma` um monad `m a`, o que teremos como saída?

Dissecando o Peixe

```
f :: a -> m b
```

```
fmap f ma
fmap :: (a -> m b) -> m a -> m (m b)
```

Teremos como saída um tipo `m (m b)`.

Dissecando o Peixe

Precisamos de uma função que colapse a estrutura para apenas um Monad:

```
join :: m (m a) -> m a
```

```
ma >>= f = join (fmap f ma)
```

Dissecando o Peixe

Com isso podemos definir um Monad também como:

```
class Monad m where
   join   :: m (m a) -> m a
   return :: a -> m a
```

Dissecando o Peixe

A escolha de qual forma utilizar para instanciar um Monad depende da própria estrutura que queremos instanciar. Escolhemos o que for mais fácil definir e o resto é definido de graça!

Dissecando o Peixe

A função `join` do nosso Monad `Writer w` fica:

```
join (Writer (Writer (a, w), w'))
                = Writer (a, w `mappend` w')
```

Então...

Imagine que temos uma função `a -> b` e queremos transformá-la em `a -> m b`. No `Writer w` podemos fazer:

```
even :: Int -> Bool
even x = x `mod` 2 == 0

tell :: w -> Writer w ()
tell s = Writer ((), s)

evenW x = tell "even" >>= \() -> return (even x)
```

Então...

Reparem que a função `\() -> return x` não faz uso do argumento de entrada, apenas altera o embelezamento da saída da função, podemos definir um operador específico para esses casos:

```
(>>) :: m a -> m b -> m b
ma >> mb = ma >>= \() -> mb
```

Então...

Com esse operador, chamado de *then*, temos:

```
1 evenW x = tell "even" >> return (even x)
```

Esse operador descarta o argumento mas executa o *embelezamento* da função.

Monads em outras linguagens

Monads estão aos poucos aparecendo nas linguagens de programação orientadas a objetos.

A linguagem C++ está introduzindo o conceito de *resumable functions* que o Python já implementa com `yield` e Haskell com *continuation Monad*.

Exemplos de Monads

Monads e Efeitos

Uma linguagem que somente aceita funções puras é útil?

Monads e Efeitos

É possível minimizar o número de funções impuras, mas elas são necessárias para a entrada e saída dos dados.

Monads e Efeitos

Algumas das utilidades de funções impuras conforme listado no artigo de Eugenio Moggi:

- **Não determinismo:** quando a computação pode retornar múltiplos resultados dependendo do caminho da computação.
- **Efeitos colaterais:** quando a computação acessa ou altera um estado como
  - Read-only, leitura do ambiente
  - Write-only, escrita de um log
  - Read/Write
- **Entrada e Saída Interativa**.

Não-determinismo

Não-determinismo

Se uma função pode retornar diferentes resultados, dependendo de certos estados internos, ela é chamada de não-determinística.

Não-determinismo

Uma função que avalia a melhor jogada de um jogo de xadrez deve levar em conta todas as possibilidades de jogadas do seu adversário.

Não-determinismo

Esse tipo de computação pode ser representada como uma lista contendo todas as possibilidades de saída.

Como no Haskell podemos trabalhar com listas infinitas (e a avaliação delas é preguiçosa), podemos usar o Monad `[]` para representar computações não-determinística.

Não-determinismo

A instância Monad para listas é facilmente implementada pela função `join`:

```
1  instance Monad [] where
2    join      = concat
3    return x = [x]
```

Não-determinismo

Essa definição é suficiente para o operador *bind*, que é definido como `as >>= k = concat (fmap k as)`.

Não-determinismo

Em versões futuras do C++ teremos o `range comprehensions` que implementa uma lista preguiçosa similar ao Haskell:

```
1  template<typename T, typename Fun>
2  static generator<typename std::result_of<Fun(T)>::type>
3  bind(generator<T> as, Fun k)
4  {
5      for (auto a : as) {
6          for (auto x : k(a) {
7              __yield_value x;
8          }
9      }
10 }
```

Não-determinismo

E no Python, utilizamos os generators:

```
1  def bind(xs, k):
2      return (y for x in xs for y in k(x))
```

Não-determinismo

Um exemplo da expressividade do Monad lista no Haskell é o cálculo de todas as triplas pitagóricas, que pode ser implementada como:

```
guard :: Bool -> [()]
guard True  = [()]
guard False = []

triples = [1..] >>= \z  -> [1..z]
                >>= \x  -> [x..z]
                >>= \y  -> guard (x^2 + y^2 == z^2)
                >>  return (x, y, z)
```

Não-determinismo

Que pode ser reescrita como:

```
triples = concat (fmap fz [1..])
fz z    = concat (fmap fx [1..z])
fx x    = concat (fmap fy [x..z])
fy y    = concat (fmap fg (guard (x^2 + y^2 == z^2)))
fg ()   = [(x, y, z)]
```

Não-determinismo

```
triples = concat [fz 1, fz 2, fz 3 ..]
fz 1 = concat [fx 1]
fx 1 = concat [fy 1]
fy 1 = concat (fmap fg (guard 1 + 1 == 1))
fy 1 = concat (fmap fg [])
```

Não-determinismo

```
triples = concat [[], [], [], [], fz 5 ..]
fz 5  = concat [fx 1, fx 2, fx 3, fx 4, fx 5]
fx 3  = concat [fy 3, fy 4, fy 5]
fy 4  = concat (fmap fg (guard 9 + 16 == 25))
fy 4  = concat (fmap fg [()])
fg () = [(3,4,5)]

triples = concat [[], [], [], [], [(3,4,5)] ..]
```

Não-determinismo

O Haskell possui um *syntactic sugar* específico para listas, e essa mesma lista pode ser reescrita como:

```
triples = [(x,y,z) | z <- [1..]
                   , x <- [1..z]
                   , y <- [x..z]
                   , x^2 + y^2 == z^2]
```

Não-determinismo

No Python podemos fazer uma construção similar como:

```
def triples(n):
  return ( (x,y,z) for z in range(1, n+1)
                   for x in range(1, z+1)
                   for y in range(x, z+1)
                   if (x**2 + y**2 == z**2))
```

Read-only

Read-only

A leitura de um estado externo de um ambiente genérico `e` é interpretado como uma função que recebe não só o argumento original como um argumento extra codificando o ambiente `e`:

```
f :: (a, e) -> b
```

O embelezamento está no argumento da função.

Read-only

Ao aplicar o *currying* nessa função temos que ela é equivalente a `f :: a -> (e -> b)`, ou seja, `f :: a -> Reader e b`.

Read-only

O Monad `Reader` faz o papel de manipulação de estados somente-leitura e vem equipado com as funções auxiliares `runReader`, que executa o `Reader` para um ambiente `e`, e `ask` que recupera tal ambiente.

Read-only

```haskell
data Reader e a = Reader (e -> a)

runReader :: Reader e a -> e -> a
runReader (Reader f) e = f e

ask :: Reader e e
ask = (Reader id)
```

Read-only

Note que a definição do `Reader` e todas as funções que a utilizam são essencialmente puras, dada uma tabela grande o suficiente poderíamos memoizar todas as entradas e saídas possíveis para todo estado possível do ambiente `e`.

Read-only

Vamos começar a construção da definição de Monad pela função `return`:

```haskell
return a = Reader (\e -> a)
```

Ou seja, dado um valor a qualquer ele simplesmente retorna esse valor para qualquer ambiente.

Read-only

O *bind* do `Reader e` recebe um `Reader e a` e uma função `a -> Reader e b` e deve retornar um `Reader e b`.

Read-only

Para isso deve fazer os seguintes passos:

1. executa `ra` no ambiente atual `e`, capturando o resultado puro `a`.
2. aplica a função `k` em `a` que retorna um `Reader e b`.
3. executa esse `Reader` no ambiente passado como argumento.

Read-only

A definição de Monad para o `Reader e` fica:

```haskell
instance Monad (Reader e) where
  -- (Reader e a) -> (a -> Reader e b) -> (Reader e b)
  -- (Reader f) >>= k = \e -> runReader (k (f e)) e
  ra >>= k = Reader (\e -> let a  = runReader ra e
                               rb = k a
                           in  runReader rb e)

  return a = Reader (\e -> a)
```

Read-only

A definição de nosso operador de composição fica:

```
1 (>=>) :: (a -> Reader e b) -> (b -> Reader e c) -> (a -> Reader
2 f >=> g = \a -> (f a) >>= g
```

Read-only

Imagine que temos um algoritmo que possui uma estrutura de configuração utilizada por uma função principal e funções auxiliares.

```
1  data Config = Conf { numberOfElems :: Int
2                     , filtro        :: (Int -> Bool)
3                     , transform     :: (Int -> Int)
4                     }
```

Read-only

```haskell
recupera :: Config -> String -> [Int] -> [Int]
recupera (Conf nel f t) "filtra" xs      = filtra f t nel xs
recupera (Conf nel f t) "transforma" xs = transforma t nel xs

transforma :: (Int -> Int) -> Int -> [Int] -> [Int]
transforma t nel xs = take nel (fmap t xs)

filtra :: (Int-> Bool) -> (Int -> Int) -> Int -> [Int] -> [Int]
filtra f t nel xs = let fs = filter f xs
                        ts = fmap t fs
                    in  take nel ts
```

Read-only

Para evitar ter que passar o parâmetro de configuração para todas as funções podemos definir `cfg` como uma variável global acessível por todas as funções.

Read-only

Read-only

Porém, se precisarmos carregar essas configurações de um arquivo externo, não podemos deixá-la como global no Haskell.

Read-only

Nas linguagens que permitem o uso de variáveis globais, todas as funções que utilizam a estrutura de configuração em funções se tornariam impuras. Além disso devemos tomar cuidado para não alterar essa variável de nenhuma maneira!

Read-only

Utilizando o `Reader` Monad podemos resolver essa situação da seguinte forma:

```haskell
recupera' :: String -> [Int] -> Reader Config [Int]
recupera' "filtra" xs      = filtra' xs
recupera' "transforma" xs = transforma' xs
```

Tanto a função `filtra'` como a função `transforma'` retornam um `Reader Config [Int]`, ou seja, estão preparadas para ler a configuração externa.

Read-only

```haskell
transforma' :: [Int] -> Reader Config [Int]
transforma' xs = (fmap transform ask) >>= \t ->
                          (fmap numberOfElems ask) >>=
                                \n -> return (take n (fmap t xs))
```

não está nada bonito...

Read-only

Primeiro ponto que podemos melhorar é perceber que o parâmetro `numberOfElems` é utilizado apenas por `take`, e o parâmetro `transform` apenas pelo `fmap`. Podemos então separar essas duas funções utilizando o operador peixe!

Read-only

```haskell
transforma' :: [Int] -> Reader Config [Int]
transforma' xs = (aplicaMap >=> pega) xs

aplicaMap  :: [Int] -> Reader Config [Int]
aplicaMap xs = fmap transform ask >>= \t -> return (fmap t xs)

pega :: [Int] -> Reader Config [Int]
pega xs = fmap numberOfElems ask >>= \n -> return (take n xs)
```

Read-only

Agora vamos eliminar as repetições! O padrão `fmap param ask` pode ser substituído por:

```
askFor p = fmap p ask
```

Read-only

O padrão `\p -> return (f p xs)` pode ser reescrito como:

```
\p -> return ((flip f) xs p)
\p -> (return . ((flip f) xs)) p
return . ((flip f) xs)
```

Read-only

Com isso nosso aplicaMap e pega se tornam:

```haskell
aplicaMap :: [Int] -> Reader Config [Int]
aplicaMap xs = askFor transform >>= return . fmapXS
  where fmapXS = (flip fmap) xs

pega :: [Int] -> Reader Config [Int]
pega xs = askFor numberOfElems >>= return . takeXS
  where takeXS = (flip take) xs
```

Read-only

Podemos também criar uma função `bindTo` que faz:

```
bindTo :: (Config -> a) -> (a -> b)
       -> Reader Config b
bindTo p f = askFor p >>= return . f
```

Read-only

Mas `Config -> a` é equivalente a `Reader Config a`:

```
bindTo :: Reader Config a -> (a -> b)
       -> Reader Config b
bindTo p f = askFor p >>= return . f
```

Read-only

Generalizando `Reader Config` para um Monad `m` qualquer:

```
bindTo :: m a -> (a -> b) -> m b
bindTo p f = askFor p >>= return . f
```

Se invertermos os argumentos, o que obtemos?

```
?? :: (a -> b) -> m a -> m b
?? f p = askFor p >>= return . f
```

Read-only

Com isso nosso `aplicaMap` e `pega` se tornam:

```haskell
aplicaMap :: [Int] -> Reader Config [Int]
aplicaMap xs = fmap fmapXS (askFor transform)
  where fmapXS = (flip fmap) xs

pega :: [Int] -> Reader Config [Int]
pega xs = fmap takeXS (askFor numberOfElems)
  where takeXS = (flip take) xs
```

Read-only

Para executar o algoritmo precisamos fazer `runReader (algorithm xs) c`, sendo `c` a variável contendo a configuração.

Read-only

Notem que a variável contendo a configuração não é passada diretamente para nenhum das funções do algoritmo, qualquer alteração que seja feita nessa estrutura ou no uso dela, não criará um efeito cascata de alterações no código.

Read-only

Em Python podemos fazer algo muito similar:

```
1  Conf = namedtuple('Conf', ['numberOfElems', 'filtro', 'transform
2
3  def numberOfElems(c):
4    return c.numberOfElems
5  def filtro(c):
6    return c.filtro
7  def transform(c):
8    return c.transform
```

Read-only

A classe Reader:

```
1  class Reader():
2    # Reader e a
3    def __init__(self, fun = None):
4      self.r = fun
5
6    def run(self, e):
7      return self.r(e)
8
9    # (a -> b) -> Reader e a -> Reader e b
10   def fmap(self, f):
11     return Reader(lambda e: f(self.r(e)))
12
13   def unit(self, x):
14     return Reader(lambda e: x)
```

Read-only

A classe Reader (cont.):

```
1   # Reader e a -> (a -> Reader e b) -> Reader e b
2   def bind(self, fab):
3     def f(e):
4       a = self.r(e)
5       rb = fab(a)
6       return rb.run(e)
7     return Reader(f)
```

Read-only

E as funções auxiliares:

```
1  def composeMonad(f, g):
2    def k(a):
3      return f(a).bind(g)
4    return k
5
6  def ask(e):
7    return e
8
9  def askFor(f):
10   return Reader(ask).fmap(f)
11
12 def bindTo(p, f):
13     return askFor(p).bind(lambda x: Reader().unit(f(x)))
```

Read-only

```
1  def recupera(tipo, xs):
2    if tipo == "filtra":
3      return filtra(xs)
4    else:
5      return transforma(xs)
6
7  def filtra(xs):
8    f = composeMonad( composeMonad(aplicaFiltro, pega),
9                      aplicaMap
10                   )
11   return f(xs)
12
13 def transforma(xs):
14   f = composeMonad(pega, aplicaMap)
15   return f(xs)
```

Read-only

```
1  def aplicaFiltro(xs):
2    filtroXS = lambda f: filter(f, xs)
3    return askFor(filtro).fmap(filtroXS)
4
5  def aplicaMap(xs):
6    mapXS = lambda f: map(f, xs)
7    return askFor(transform).fmap(mapXS)
8
9  def pega(xs):
10   pegaXS = lambda n: list(xs)[:n]
11   return askFor(numberOfElems).fmap(pegaXS)
```

Unread-only

A versão sem o uso de Monad possui menos funções, porém maiores. Vale a pena? ¯_(ツ)_/¯

```
1  def recupera(tipo, config, xs):
2    if tipo == "filtra":
3      return filtra(config, xs)
4    else:
5      return transforma(config, xs)
6
7  def filtra(config, xs):
8    n = config.numberOfElems
9    f, t = config.filtro, config.transform
10   ys = map(t, filter(f, xs))
11   return list(ys)[:n]
12
13 def transforma(xs):
14   n = config.numberOfElems
15   t = config.transform
16   ys = map(t, xs)
17   return list(ys)[:n]
```

State

State

Um estado simplesmente é um ambiente e que permite leitura e escrita, ou seja, é a combinação dos Monads `Reader` e `Writer` (chamados de **functores adjuntos**).

State

Uma função `f :: a -> b` é embelezada para
`f :: (a, s) -> (b, s)`, e utilizando *currying* temos
`f :: a -> (s -> (b, s))`.

State

Assim como o `Reader` ele é equipado com funções auxiliares:

```haskell
data State s a = State (s -> (a, s))
               = Reader s (Writer s a)

runState :: State s a -> s -> (a, s)
runState (State f) s = f s

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

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

State

Com isso temos a capacidade de executar, ler ou alterar um estado.

A instância de Monad nesse caso fica muito parecida com o Monad `Reader`, exceto que tomamos o cuidado de passar o novo estado para o próximo `runState`.

State

Relembrando a instância de Reader:

```haskell
instance Monad (Reader e) where
  -- (Reader e a) -> (a -> Reader e b) -> (Reader e b)
  ra >>= k = Reader (\e -> let a  = runReader ra e
                              rb = k a
                          in  runReader rb e)

  return a = Reader (\e -> a)
```

State

Agora trabalhamos com um estado s:

```haskell
instance Monad (State s) where
  -- (State s a) -> (a -> State s b) -> (State s b)
  ra >>= k = State (\s -> let a  = runState ra s
                              rb = k a
                          in runState rb s)

  return a = State (\s -> a)
```

State

Porém, agora as funções retornam um novo estado:

```haskell
instance Monad (State s) where
  -- (State s a) -> (a -> State s b) -> (State s b)
  sa >>= k = State (\s -> let (a, s') = runState sa s
                              rb      = k a
                          in  runState rb s')

  return a = State (\s -> (a, s))
```

State

Equivalente em Python:

```
1  class State:
2    # State s -> (a, s)
3    def __init__(self, f = None):
4      self.r = f
5
6    def run(self, s):
7      return self.r(s)
8
9    def unit(self, x):
10     return State(lambda s: (x, s))
11
12   def bind(self, k):
13     def f(s):
14       (a, sn) = self.run(s)
15       return k(a).run(sn)
16     return State(f)
17
18   def fmap(self, f):
19     def applySt(f, st):
20       return (f(st[0]), st[1])
21     return State(lambda s: applySt(f,self.r(s)))
```

State

Uma aplicação desse Monad é na manipulação de números aleatórios em que queremos que o estado do gerador seja atualizado a cada chamada da função `random`.

State

Vamos exemplificar com uma função que alterar uma lista de `Bool`, invertendo cada um de seus elementos, caso um certo valor aleatório seja `< 0.3`.

State

```haskell
import Control.Monad
import Control.Monad.State
import System.Random

randomSt :: (RandomGen g) => State g Double
randomSt = state (randomR (0.0, 1.0))

change :: Bool -> Double -> Bool
change b p = if p < 0.3 then not b else b

booleanos :: [Bool] -> State StdGen [Bool]
booleanos bs = ???
```

State

Vamos pensar em como alterar um único booleano aleatoriamente:

```haskell
import Control.Monad
import Control.Monad.State
import System.Random

randomSt :: (RandomGen g) => State g Double
randomSt = state (randomR (0.0, 1.0))

change :: Bool -> Double -> Bool
change b p = if p < 0.3 then not b else b

changeAtRandom :: Bool -> State g Bool
changeAtRandom b = fmap (change b) randomSt
```

State

Agora precisamos criar uma lista desses booleanos:

```haskell
booleanos :: [Bool] -> [State g Bool]
booleanos bs = [changeAtRandom b | b <- bs]
```

hmmm, mas eu queria na verdade um `State g [Bool]`

State

Preciso de uma função que execute as ações de gerar um número aleatório na sequência da lista. Não queremos executar usando o mesmo estado para todos os elementos...

State

Se a lista está vazia é fácil resolver!

```
1  sequence :: [State g Bool] -> State g [Bool]
2  sequence []     = return []
```

State

```haskell
sequence :: [State g Bool] -> State g [Bool]
sequence []     = return []
-- b' :: Bool
sequence (b:bs) = b >>= \b' -> ??
```

State

```haskell
sequence :: [State g Bool] -> State g [Bool]
sequence []     = return []
-- sequence bs :: State g [Bool]
sequence (b:bs) = b >>= \b' -> sequence bs
```

State

```haskell
sequence :: [State g Bool] -> State g [Bool]
sequence []     = return []
-- bs' :: [Bool]
sequence (b:bs) = b >>= \b' -> sequence bs >>= \bs' -> ???
```

State

```haskell
sequence :: [State g Bool] -> State g [Bool]
sequence []     = return []
-- (b':bs') :: [Bool]
sequence (b:bs) = b >>= \b' -> sequence bs
                        >>= \bs' -> ?? (b':bs')
```

State

```haskell
sequence :: [State g Bool] -> State g [Bool]
sequence []     = return []
-- return (b':bs') :: State g [Bool]
sequence (b:bs) = b >>= \b' -> sequence bs
                        >>= \bs' -> return (b':bs')
```

State

Com isso nossa função `booleanos` se torna:

```haskell
booleanos :: [Bool] -> State g [Bool]
booleanos bs = sequence [changeAtRandom b | b <- bs]

main = do
  g <- getStdGen
  let bs = replicate 20 True
  print (fst $ runState (booleanos bs) g)
```

State

Imagine que agora também temos as seguintes funções:

```haskell
select b p = if (p < 0.3 && b) then [b] else []

selection :: [Bool] -> State StdGen [Bool]
selection bs = fmap concat bs'
  where bs' = sequence rbs
        rbs = [fmap (select b) randomSt | b <- bs]
```

E ela deve ser aplicada na saída da função `booleanos`.

State

```
1  (booleanos bs) :: State StdGen [Bool]
2  selection :: [Bool] -> State StdGen [Bool]
```

O que fazemos?

State

```
1  booleanos bs >>= selection
```

State

Em Python podemos escrever:

```
1  import random
2  from functools import partial
3
4  def change(b, p):
5    if p < 0.3:
6      return not b
7    return b
8
9  def mutation(bs):
10   return sequence([myRandST.fmap(partial(change,b))
11                                for b in bs])
```

State

Em Python podemos escrever:

```
1  import random
2  from functools import partial
3
4  def select(b, p):
5    if p < 0.3 and b:
6      return [b]
7    return []
8
9  def concat(xss):
10   return [x for xs in xss
11               for x in xs]
12
13 def selection(bs):
14   return sequence([myRandST.fmap(partial(select,b))
15                            for b in bs]).fmap(concat)
```

State

A função sequence é escrita em Python como:

```
1  # [State bool] -> State [bool]
2  def sequence(ss):
3    def f(s):
4      xs = []
5      for si in ss:
6        (x, s) = si.run(s)
7        xs.append(x)
8      return (xs, s)
9    return State(f)
```

State

E o restante do programa fica:

```
1  def myRand(s):
2    random.setstate(s)
3    x = random.random()
4    return x, random.getstate()
5
6  myRandST = State(myRand)
7
8  bs = [True]*10
9  bs1 = (mutation(bs)
10          .bind(selection)
11          .run(random.getstate())
12        )
13
14 print( bs, bs1[0] )
```

State

```
1  def mutation(bs):
2    return [change(b, random()) for b in bs]
3
4  def select(b, p):
5    if p < 0.3 and b:
6      return True
7    return False
8
9  def selection(bs):
10   return [b for b in bs if select(b, random())]
11
12 bs1 = selection(mutation(bs))
```

State

Poderíamos fazer uma observação similar ao `Reader` monad, porém notem que a função `random()` altera um estado global do programa.

State

Pensando no mesmo programa com a passagem de estado como parâmetro, teríamos algo como:

```
1  def mutation(bs, s):
2    nbs = []
3    for b in bs:
4      (p, s) = random(s)
5      nbs.append(change(b, p))
6    return nbs, s
7
8  def selection(bs, s):
9    nbs = []
10   for b in bs:
11     (p, s) = random(s)
12     nbs.append(select(b, p))
13   return nbs, s
14
15 (bs1, s) = mutation(bs, s)
16 (bs2, s) = selection(b, s)
```

IO

IO

Quando dizemos que Haskell é uma linguagem de programação puramente funcional e **todas** suas funções são puras, a primeira questão que vem na mente é de como as funções de entrada e saída são implementadas.

Como as funções `getChar`, `putChar` podem ser puras se elas dependem do efeito colateral? Como é possível compor funções puras com a saída de `getChar` se a saída é, teoricamente, indeterminada?

IO

O segredo das funções de manipulação de `IO` é que elas tem seus valores guardados dentro de um container (o `IO` Monad) que nunca pode ser aberto.

Ou seja, criamos funções que lidam com `Char` sem saber exatamente quem é esse caracter.

IO

Podemos imaginar o Monad `IO` como uma caixa quântica contendo uma superposição de todos os valores possíveis de um tipo.

IO

Toda chamada de função para esse tipo é **jogada lá dentro** e executada pelo sistema operacional quando apropriado.

IO

As assinaturas de getChar e putChar são:

```
getChar :: IO Char -- () ->  IO Char

putChar :: Char :: IO ()
```

IO

Note que a implementação da instância de Functor e Monad para `IO` é implementada internamente no Sistema Operacional e não temos um `runIO` que nos devolve um valor contido no container.

Ao fazer `fmap f getChar` a função será executada no retorno de `getChar` mas não poderemos ver seu resultado.

IO

Uma outra forma de pensar no `IO` é como um tipo `State`:

```
data IO a = Mundo -> (a, Mundo) = State Mundo a
```

IO

A sequência:

```
do putStr "Hello"
   putStr "World"
```

Causa uma dependência funcional entre as duas funções de tal forma que elas serão executadas na sequência.

Uma palavra final sobre Monads

Em resumo, Monads nos permite colocar um valor dentro de um contexto e criar funções que colocam valores dentro desse contexto sem perder a propriedade de composição:

```
a -> m b
```

mas ele não nos garante que podemos retirar o conteúdo do contexto...

Comonads

Comonads

A categoria oposta da Kleisli, denominada **co-Kleisli** leva ao conceito de **Comonads**.

Agora temos endofunctors `w` e morfismos do tipo `w a -> b`.

Comonads

Queremos definir um operador de composição para eles, da mesma forma que definimos o operador >=>:

```
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
```

Comonads

Nosso morfismo identidade é similar ao `return` mas com a seta invertida:

```
extract :: w a -> a
```

`extract` permite extrair um conteúdo do functor `w` (nosso container).

Comonads

O oposto de nosso operador `bind`, chamado de `extend` (=>>) tem assinatura assinatura:

```
(=>>) :: (w a -> b) -> w a -> w b
```

Seja `f` uma função que retira um `a` do container `w` e transforma em `b`, dado um `w a`, me retorne um `w b`.

Comonads

Finalmente, o oposto de `join` é o `duplicate`:

```
duplicate :: w a -> w (w a)
```

ela insere um container dentro de outro container

Comonads

Nesse ponto, podemos perceber a dualidade entre Monad e Comonad.

Comonads

O Monad nos dá uma forma de colocar valores dentro de containers, mas sem garantias de que podermos retirá-los.

O valor fica envolvido em um contexto que pode ficar escondido até o fim.

Comonads

O Comonad dá uma forma de retirar um valor de um container, sem prover uma forma de colocá-lo de volta.

Podemos focar em um elemento e manter todo o contexto em volta dele (e já vimos isso nos tipos buracos!).

Comonads

Então nossa classe Comonad é definida por:

```haskell
class Functor w => Comonad w where
    extract   :: w a -> a

    (=>=)     :: (w a -> b) -> (w b -> c) -> (w a -> c)
    f =>= g = g . (f =>>)

    (=>>)     :: (w a -> b) -> w a -> w b
    f =>> wa = fmap f . duplicate

    duplicate :: w a -> w (w a)
    duplicate = (id =>>)
```

Definições padrão

A ideia de um Comonad é que você possui um container com um ou mais valores de a e que existe uma noção de *valor atual* ou *foco* em um dos elementos.

Esse valor atual é acessado pela função `extract`.

Definições padrão

O operador *co-peixe* (=>=) faz alguma computação no valor atual porém tendo acesso a tudo que está em volta dele.

Definições padrão

A função `duplicate` cria diversas versões de `w a` com os diferentes focos.

Definições padrão

A função `extend` (=>>) aplica uma função em todas as possibilidades de foco.

Stream de Dados

Podemos definir um Stream de dados como uma lista infinita não-vazia:

```haskell
data Stream a = Cons a (Stream a)

instance Functor Stream where
  fmap f (Cons a as) = Cons (f a) (fmap f as)
```

Stream de Dados

Essa estrutura possui um foco em seu primeiro elemento.

Podemos definir a função `extract` simplesmente como:

```
1  extract (Cons a _) = a
```

Stream de Dados

A função `duplicate` deve gerar uma Stream de Streams, cada uma com um foco diferentes:

```
1  duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

Stream de Dados

Com isso temos as funções necessárias para criar uma instância de Comonad para Streams:

```haskell
instance Comonad Stream where
  extract (Cons a _)    = a
  duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

Stream de Dados

Como exemplo de aplicação vamos criar uma função que calcula a média móvel de um stream de dados.

Stream de Dados

Começamos com a definição da média entre os *n* próximos elementos:

```haskell
sumN :: Num a => Int -> Stream a -> a
sumN n (Cons a as) | n <= 0     = 0
                   | otherwise = a + sumN (n-1) as

avgN :: Fractional a => Int -> Stream a -> a
avgN n as = (sumN n as) / (fromIntegral n)
```

Stream de Dados

Notem que `avgN` tem assinatura `Int -> (w a -> a)`, para gerarmos uma Stream de médias móveis queremos algo como `Int -> (w a -> a) -> w a -> w a`, que remete a assinatura de `=>>`:

```
1 movAvg :: Fractional a => Int -> Stream a -> Stream a
2 movAvg n as = (avgN n) =>> as
```

Stream de Dados

Com isso, a função `avgN n` será aplicada em cada foco de `as` gerando uma nova Stream contendo apenas os valores das médias.

Store Comonad

Relembrando o `State` Monad visto anteriormente, ele foi definido como a composição `(Reader s) . (Writer s)`:

```haskell
data State s a = State (s -> (a, s))
               = Reader s (Writer s a)
```

Isso foi possível pois os Functors `Reader` e `Writer` são adjuntos.

Store Comonad

De forma análoga podemos fazer a composição complementar para criarmos um Comonad:

```haskell
data Store s a = Store (s -> a) s
               = Writer s (Reader s a)
```

Store Comonad

A instância de Functor para esse Comonad é simplesmente a composição da função definida por `Reader s a`:

```haskell
instance Functor (Store s) where
  fmap g (Store f s) = Store (g . f) s
```

Store Comonad

A assinatura de `extract` deve ser `Store s a -> a`, sendo que o tipo `Store` armazena uma função `s -> a` e um `s`, basta aplicar a função no estado `s` que ele armazena.

Store Comonad

Por outro lado, a função `duplicate` pode se aproveitar da aplicação parcial na construção de um valor definindo:

```
1  instance Comonad (Store s) where
2    extract (Store f s)   = f s
3    duplicate (Store f s) = Store (Store f) s
```

Store Comonad

O Comonad `Store` é um par que contém um container (a função **f**) que armazena diversos elementos do tipo **a** indexados pelo tipo **s** e um **s** que indica o foco atual da estrutura.

Store Comonad

Nessa interpretação temos que `extract` retorna o elemento `a` na posição atual `s` e `duplicate` simplesmente cria infinitas cópias desse container de tal forma que cada cópia está deslocada em n posições para direita ou para a esquerda.

Store Comonad em Python

```python
class Store:
  def __init__(self, f, s):
    self.f, self.s = f, s

  def extract(self):
    return self.f(self.s)

  def duplicate(self):
    return Store(lambda s: Store(self.f, s), self.s)
```

Store Comonad em Python (cont.)

```
1     def fmap(self, g):
2       f = lambda s: g(self.f(s))
3       return Store(f, self.s)
4
5     def extend(self, g):
6       return self.duplicate().fmap(g)
```

Store Comonad

Como exemplo, vamos implementar o automato celular 1D conforme descrito por Wolfram.

Store Comonad

Esse automato inicia com uma lista infinita indexada por valores inteiros (positivos e negativos) e centralizada em 0.

A lista contém inicialmente o valor 1 na posição central e 0 em todas as outras posições.

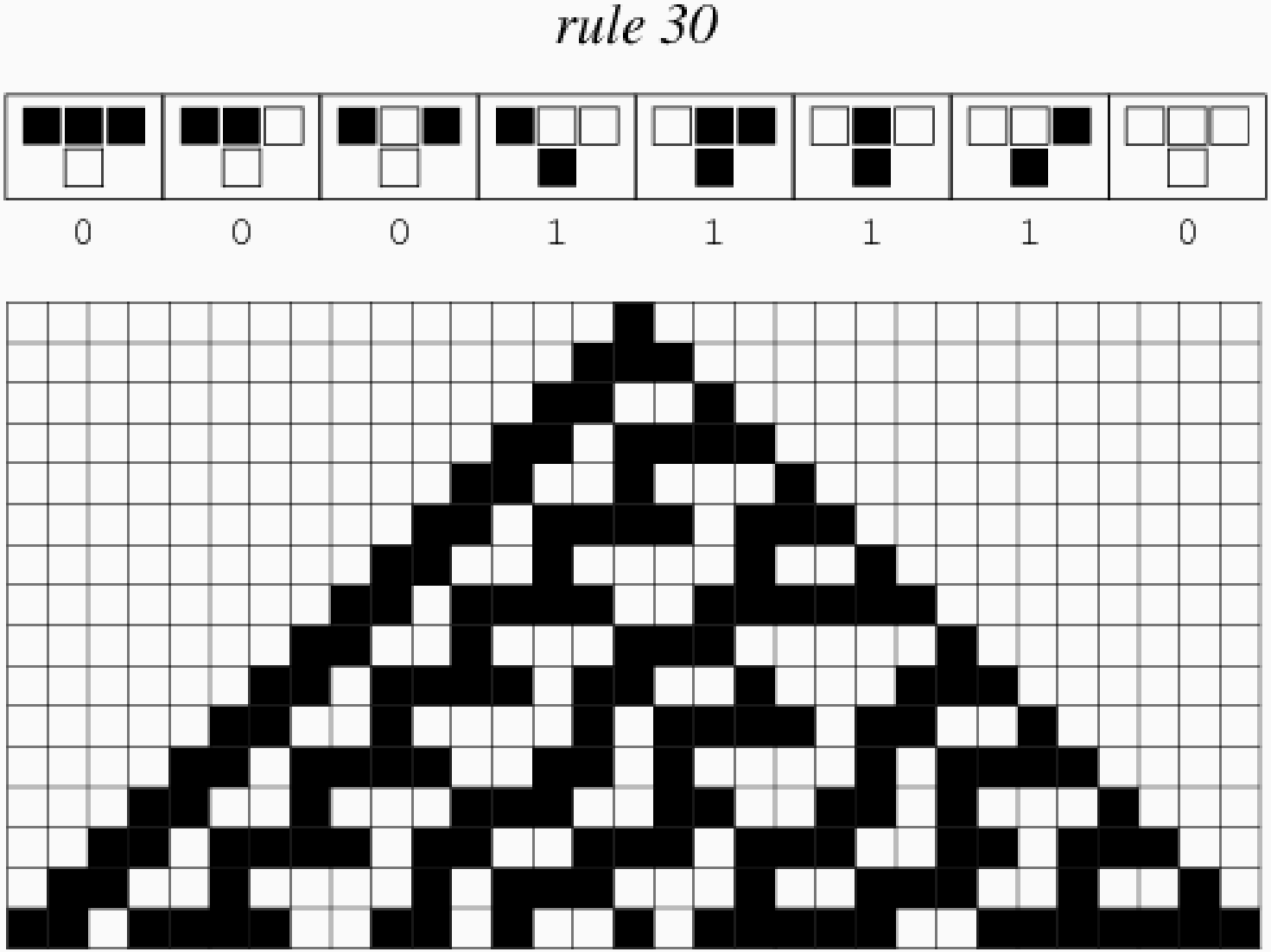
Store Comonad

A cada passo da iteração, a lista é atualizada através das `regras n` em que $0 \leq n \leq 255$.

A numeração da regra codificam um mapa de substituição para o número binário formado pela subslita composta do valor atual e de seus dois vizinhos.

Store Comonad

Por exemplo, a regra 30 codifica:



Store Comonad

Podemos implementar esse autômato utilizando um `Store Int Int`, primeiro definindo a função que aplica a regra:

```
# Int -> Store Int Int -> Int
def rule(x, fs):
  revBin = bin(x)[2:][::-1]
  bit = 4*fs.f(fs.s-1) + 2*fs.f(fs.s) + fs.f(fs.s+1)

  if bit >= len(revBin):
    return 0
  return revBin[bit]
```

Store Comonad

Fazendo uma aplicação parcial do número da regra, a assinatura da função fica: `Store Int Int -> Int` que deve ser aplicada em um `Store Int Int` para gerar a próxima função de indexação.

Store Comonad

Isso sugere o uso de `extend` (=>>):

```
1 def nextStep(rl, fs):
2   return fs.extend(rl)
```

Store Comonad

Mas queremos aplicar sucessivamente essa regra:

```
1  def wolfram(rl, fs):
2    while True:
3      yield fs
4      fs = fs.extend(rl)
```

Store Comonad

A representação inicial de nosso ambiente é feita por:

```
1  def f0(x):
2    if x==0:
3      return 1
4    return 0
5
6  fs = Store(f0, 0)
```

Store Comonad

E, então, criamos nosso automato com a regra 30 fazendo:

```
1  wolf30 = wolfram(partial(rule, 30), fs)
2  top5 = itertools.islice(wolf30, 6)
3
4  for w in top5:
5    print(w)
```

Store Comonad

O Comonad Store pode ser aplicado com funções de n-variáveis, então não estamos limitados a containers unidimensionais.

Store Comonad

Por exemplo, podemos representar uma imagem bitmap como um Store e aplicar uma sequência de filtros da seguinte forma:

```
(img.extend(sharpen)
    .extend(emboss)
)
```

```
img =>> sharpen =>> emboss
```

Store Comonad

Os filtros precisam podem ser facilmente construídos através apenas da matriz de convolução!

Atividades para Casa

Atividades para Casa

1. Alguns Functors possuem instâncias de Monad e Comonad, escreva essas instâncias (na linguagem que preferir) para:

a. Functor Identity
b. O tipo lista não-vazia `data FullList a = Single a | Cons a (FullList a)`

2. Complete o código *ConvEx.py* para criar a aplicação de kernel de convolução utilizando o Comonad Store.