Paradigmas de Programação

Fabrício Olivetti de França

26 de Julho de 2018

Hask: categoria dos tipos

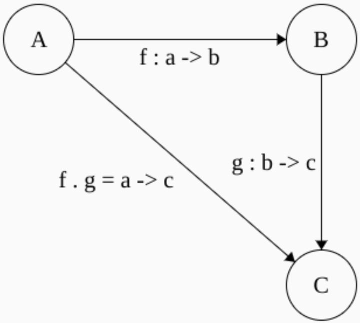
Teoria das Categorias é uma área de estudo da matemática que generaliza o estudo de relações estruturadas. Isso é feito por meio de um grafo direcionado em que os nós são os **objetos** e as arestas são chamadas de **morfismo** e indica uma função que transforma um objeto em outro.

Uma categoria **C** é definida por:

- Um conjunto de objetos **ob(C)**
- Um conjunto de morfismos **hom(C)**
- Um operador binário $\circ$ que define a composição de morfismos

O operador $\circ$ possui as seguintes propriedades:

- **Associativa:** dado que $f : a \to b, g : b \to c, h : c \to d$, então $h \circ (g \circ f) = (h \circ g) \circ f$
- **Identidade:** para todo objeto existe um morfismo de identidade tal que $1 : a \to a$ e $1 \circ f = f = f \circ 1$



Hask

É fácil perceber que observamos essas propriedades anteriormente com o próprio conceito de funções e composição de funções. Com isso temos a categoria **Hask** da linguagem Haskell que define:

- ob(H) = os tipos (Int, Double, Char,...)
- hom(H) = as funções que transformam um tipo em outro

A identidade do morfismo é:

```
id : a -> a
id x = x
```

Hask

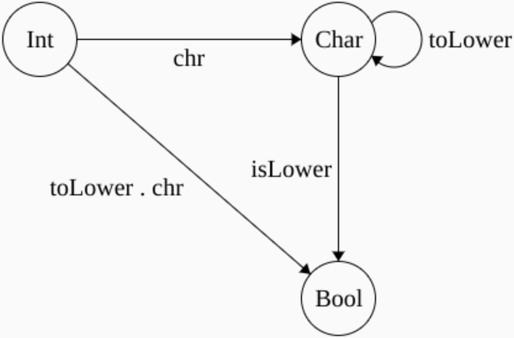


Figure 2: Hask

Construtores de Tipos

Em aulas anteriores vimos o conceito de construtores de tipos, quando criamos novos tipos. Eles recebem um tipo como parâmetro e criam um novo tipo:

```
listaDeDouble :: [Double]
talvezInt     :: Maybe Int
arvoreChar    :: Tree Char
```

Considerações

Tipo paramétrico é todo tipo que possui um parâmetro de tipo:

```
[a], Maybe a, Tree a, ...
```

Considerações

A partir desse momento vamos pensar que os tipos paramétricos definem uma **computação** que produz um valor do tipo **a**. Ao contrário das funções puras, essa computação pode conter efeitos colaterais.

Listas como resultados não-determinísticos

O tipo **Lista** promete entregar um conjunto de valores de resposta, após a computação, que pode representar múltiplos resultados de um algoritmo não-determinístico:

```haskell
naoDeterministico :: Int -> [Int]
naoDeterministico x = [altera x dir | dir <- direcoes]
```

Listas como sequências de operações

Além disso uma lista pode indicar a sequência de operações que devem ser seguidas. Imagine uma função `getChar` que retorna um caractere digitado pelo usuário.

Eu quero garantir que a sequência `getChar, getChar, getChar` seja executada na ordem (ou o resultado poderá ser diferente do esperado). Uma lista pode (mas não necessariamente vai) garantir tal ordem:

```
[getChar, getChar, getChar]
```

Maybe como resultados que podem falhar

O tipo **Maybe** não promete entregar nada, apenas tenta entregar um valor do tipo **a**, mas se algo der errado, ele retorna **Nada**:

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv x y | y /= 0    = Just (x `div` y)
            | otherwise = Nothing
```

Árvore binária como possíveis caminhos

Uma árvore binária promete entregar possíveis desmembramentos de uma computação sequencial.

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

> arvore = Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)
           :: Tree Int
```

Categoria dos Construtores de Tipos

Eles definem categorias próprias!

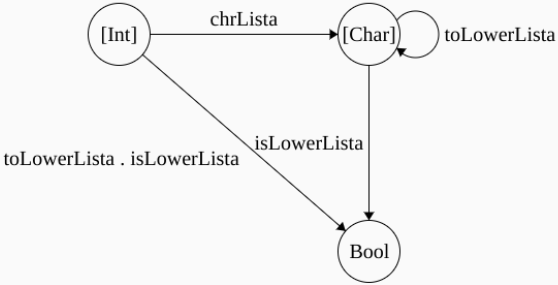


Figure 3: Categoria dos Tipos

(na verdade é a mesma categoria)

Dado que eu já tenho as funções `chr`, `toLower`, `isLower`, devo escreve-las novamente ao definir um novo tipo paramétrico?

```
chrLista :: [Int] -> [Char]
chrLista []     = []
chrLista (n:ns) = chr n : chrLista ns

isLowerLista :: [Char] -> [Bool]
isLowerLista []     = []
isLowerLista (c:cs) = isLower c : isLowerLista cs
```

Mapas entre categorias

Esse padrão nós já conhecemos! É o `map`:

```
chrLista     = map chr

isLowerLista = map isLower
```

Mapas entre categorias

E se estivermos trabalhando com `Maybe`?

```haskell
chrMaybe :: Maybe Int -> Maybe Char
chrMaybe Nothing  = Nothing
chrMaybe (Just n) = Just (chr n)
```

Eu só queria aplicar a função `chr` 😟

Se pensarmos na categoria das funções em que as funções são objetos e os morfismos seriam funções que mapeiam função de um tipo para outro teremos o que é chamado de **Functors**:

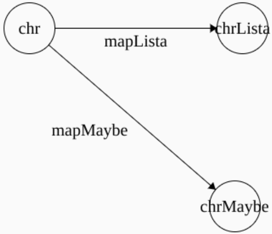


Figure 4: Functors

Functors

Functors são morfismos que transformam os morfismos de uma categoria inteira (Tipos) em morfismos de outra (Maybe).

No Haskell o que temos são **endofunctors**.

Functors

No Haskell um Functor é definido como uma classe de tipo com a seguinte definição:

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Ou seja, se eu já tenho uma função $g : a \rightarrow b$, e tenho um tipo paramétrico f, eu posso aplicar a função g em fa para obter fb.

Functors de lista

Para as listas nós já temos o functor:

```haskell
instance Functor [] where
  fmap = map
```

Functors de Maybe

Para o Maybe definimos da seguinte forma:

```haskell
instance Functor Maybe where
  fmap _ Nothing =  Nothing
  fmap g (Just x) = Just (g x)
```

Functors de Maybe

Agora podemos fazer:

```
> fmap chr Nothing
Nothing
> fmap chr (Just 65)
Just 'A'
> fmap (+1) (Just 65)
Just 66
```

Functors de Maybe

Reforçando a ideia de promessa computacional, imagine que eu esteja aplicando a função `chr` em um valor proveniente de uma computação que pode falhar:

```
> x = (n + 36) `mod` y
> fmap chr x
```

Nesse caso, se a computação de **x** falhar, a função não será aplicada e o programa não termina com erro.

Definimos um Functor de Árvores como:

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

Propriedades

Ao definir um Functor, o desenvolvedor deve garantir as seguintes propriedades:

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

Propriedades

Ou seja, ao mapear a função `id` em uma estrutura o resultado deve ser a estrutura original, a composição de dois mapeamentos é o mapeamento da composição das funções. Ou seja:

```haskell
(fmap isLower) . (fmap chr) = fmap (isLower . chr)
```

Isso nos ajuda a compor funções que serão mapeadas.

Operador Functor

Podemos utilizar o operador `(<$>)` no lugar do `fmap`:

```
> chr <$> Nothing
Nothing
> chr <$> (Just 65)
Just 'A'
> (+1) <$> [1,2,3]
[2,3,4]
```

Exercício

Considere um tipo descrevendo Pokémons que só podem atacar ou defender, o ataque/defesa pode ser descrito por diversos tipos: numérico descrevendo a força, string descrevendo o efeito, tuplas descrevendo ambos, etc.:

```
data Pokemon a = ATK a | DEF a | AD a
                 deriving (Show, Eq)
```

Escreva a instância de Functor para esse tipo.

Applicatives

Functors para funções de múltiplos argumentos

Ok, digamos que eu queira fazer:

```
> [1,2] + [3,4]
[4,5]
> (Just 3) + (Just 2)
Just 5
```

Família `fmap`

Idealmente teríamos:

```
fmap0 :: a -> f a
fmap1 :: (a -> b) -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

Com isso poderíamos:

```
> fmap2 (+) [1,2] [3,4]
[4,5]
> fmap2 (+) (Just 3) (Just 2)
Just 5
```

Mas definir todas essas funções é um trabalho tedioso...

Applicative

Podemos resolver isso através do uso de *currying*:

```
pure   :: a -> f a
aplica :: f (a -> b) -> f a -> f b

fmap0 :: a -> fa
fmap0 = pure

fmap1 :: (a -> b) -> (f a -> f b)
fmap1 g x = aplica (pure g) x

fmap2 :: (a -> (b -> c)) -> (f a -> (f b -> f c))
fmap2 g x y = aplica (aplica (pure g) x) y
```

Applicative

Isso é denominado **Applicative** cuja classe de tipo é definida como:

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

E com isso podemos fazer:

```
> pure (+) <*> [1,2] <*> [3,4]   -- não dá esse resultado
[4,5]
> pure (+) <*> (Just 3) <*> (Just 2)
Just 5
```

Applicative

O significado de `pure` nesse contexto é a de que estamos transformando uma função **pura** em um determinado contexto computacional (de computação não determinística, de computação que pode falhar, etc.)

Applicative Maybe

Para o tipo Maybe basta definirmos:

```haskell
instance Applicative Maybe where
  pure              = Just
  Nothing  <*> _    = Nothing
  (Just g) <*> mx   = fmap g mx
```

Maybe - Tratamento de Exceções

Essas definições nos ajudam a definir um modelo de programação em que funções puras podem ser aplicadas a argumentos que podem falhar, sem precisar gerenciar a propagação do erro:

```haskell
r1 = safeDiv x y
r2 = safeDiv y x

-- Se alguma divisão falhar, retorna Nothing
-- Não precisamos criar um safeAdd!
somaResultados = pure (+) <*> r1 <*> r2
```

Maybe - Tratamento de Exceções

```
> pure (+) <*> safeDiv 1 0 <*> safeDiv 0 1
Nothing
```

Applicative List

Para as listas, o uso de applicative define como aplicar um operador em todas as combinações de elementos de duas listas:

```haskell
instance Applicative [] where
  pure x    = [x]
  gs <*> xs = [g x | g <- gs, x <- xs]
```

Applicative List

Com isso temos:

```
> pure (+1) <*> [1,2,3]
[2,3,4]
> pure (+) <*> [1] <*> [2]
[3]
> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

Applicative List

```
> pure (++) <*> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh."
 ,"hmm?","hmm!","hmm."]
```

Imagine que queremos fazer a operação `x * y`, mas tanto x quanto y são não-determinísticos, ou seja, podem assumir uma lista de possíveis valores. Uma forma de tratar esse problema é através do Applicative listas que retorna todas as possibilidades:

```
> pure (*) <*> [1,2,3] <*> [2,3]
[2,3,4,6,6,9]
> pure (*) <*> [1,2,3] <*> []
[]
```

Operações com Listas

Uma outra interpretação para o Applicative de listas é a operação element-a-elemento pareados. Ou seja:

```
> pure (+) <*> [1,2,3] <*> [4,5]
[5,7]
```

Como só pode existir uma única instância para cada tipo, criaram a **ZipList** que é uma lista que terá essa propriedade na classe Applicative:

```
> import Control.Applicative
> pure (+) <*> ZipList [1,2,3] <*> ZipList [4,5]
ZipList [5,7]
```

Sequenciamento

Imagine que temos uma sequência de aplicações de uma função g a ser aplicada na ordem:

```
g :: a -> Maybe a

[g x1, g x2, g x3]
```

Na avaliação preguiçosa, quando avaliarmos uma lista cada elemento será avaliado em ordem (dependendo da função sendo avaliada).

Sequenciamento

Como a sequência é importante, não queremos continuar computando no caso de falhas. Podemos construir uma lista de Applicative da seguinte forma:

```
pure (:) <*> g x1 <*>
   (pure (:) <*> g x2 <*>
      (pure (:) <*> g x3 <*> pure []))
```

Sequenciamento

Se uma aplicação falhar, não temos motivos para continuar computando, caso a aplicação `g x2` falhe, podemos retornar `Nothing` imediatamente. É possível generalizar essa função com:

```
-- sequencia de Applicatives
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA []     = pure []
sequenceA (x:xs) = pure (:) <*> x <*> sequenceA xs
```

Sequenciamento

```
> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
> sequenceA [Just 3, Nothing, Just 1]
Nothing
> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

Sequenciamento

Sequenciamento é útil quando queremos ter controle da ordem das operações e tais operações podem gerar efeitos colaterais ou falhar. Ex.:

- Capturar caracteres do teclado
- Backtracking

Múltiplas aplicações de funções

Considere que queremos criar uma função que recebe um argumento e retorna uma lista de operações sobre esse argumento:

Múltiplas aplicações de funções

Considere que queremos criar uma função que recebe um argumento e retorna uma lista de operações sobre esse argumento:

```
> g = \x -> map (\f -> f x) [(+1), (*2), (`mod` 3)]
> g 1
[2,2,1]
> map g [1,2,3]
[[2,2,1],[3,4,2],[4,6,0]]
```

Múltiplas aplicações de funções

Uma forma mais natural é utilizar Applicatives:

```
> g' = sequenceA [(+1), (*2), (`mod` 3)]
> g' 1
[2,2,1]
> map g' [1,2,3]
[[2,2,1],[3,4,2],[4,6,0]]
```

Leis do Applicative

Toda definição de Applicative deve seguir as seguintes leis:

- `pure id <*> v = v`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`
- `u <*> (v <*> w) = pure (.) <*> u <*> v <*> w`

Isso garante que toda sequência de aplicações pode ser reescrita de tal forma que exista apenas uma função pura (que pode ser composição de várias funções puras) e ela será a primeira a ser executada, tendo sequência das funções de tipo paramétricos.

Lei de identidade

A lei da identidade fala que aplicar a função `id` em um contexto computacional retorna o próprio contexto inalterado:

```
> v = safeDiv x y
> pure id <*> v
  => Just id <*> v
  => fmap id v
```

Homomorfismo

O homomorfismo nos diz que aplicar uma função pura em um contexto puro, é o mesmo que aplicar a função no valor e envolver no contexto:

```
> pure (+) <*> pure 2 <*> pure 3 :: [Int]
  => [(+)] <*> [2] <*> [3]
  => [g x | g <- [(+)], x <- [2]] <*> [3]
  => [(+2)] <*> [3]
  => [g x | g <- [(+2)], x <- [3]]
  => [5]
  == pure (2 + 3) :: [Int]
```

Inversão

A lei da inversão fala que se temos uma expressão pura a direita, podemos inverter a ordem utilizando a função de aplicação ($):

```
> [(+3)] <*> pure 2
  => pure ($ 2) <*> [(+3)]
  => [($ 2)] <*> [(+3)]
  => [g x | g <- [($ 2)], x <- [(+3)]]
  => [($ 2) (+3)]
  => (+3) ($ 2)
  => (+3) 2
  == 5
```

Composição

Com a lei da composição, podemos transformar uma expressão associativa a direita em uma expressão associativa a esquerda:

```
> [dobra] <*> ([triplica] <*> [2,3])
  => [dobra] <*> [6,9]
  => [12,18]

> pure (.) <*> [dobra] <*> [triplica] <*> [2,3]
  => [(dobra .)] <*> [triplica] <*> [2,3]
  => [dobra . triplica] <*> [2,3]
  => [12,18]
```

Exercício

Escreva a instância de Applicative para o tipo Pokémon:

```haskell
data Pokemon a = ATK a | DEF a | AD a
               deriving (Show, Eq)
```

Monads

Vamos definir um tipo de dado que representa expressões matemáticas:

```haskell
data Expr = Val Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
```

Para avaliar essa expressão podemos definir:

```
eval :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = (eval x) + (eval y)
eval (Sub x y) = (eval x) - (eval y)
eval (Mul x y) = (eval x) * (eval y)
eval (Div x y) = (eval x) `div` (eval y)
```

Monads

Porém, se fizermos:

```
> eval (Div (Val 1) (Val 0))
*** Exception: divide by zero
```

Podemos resolver isso usando `safeDiv` e `Maybe` (vamos focar apenas na divisão):

```
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
```

Agora temos:

```
> eval (Div (Val 1) (Val 0))
Nothing
```

Mas nosso código está confuso...

Applicative?

O uso de Applicative pode resolver muitos problemas de encadeamento de funções com efeito, seria legal poder fazer:

```
> pure safeDiv <*> eval x <*> eval y
```

Mas `safeDiv` tem tipo `Int -> Int -> Maybe Int` e deveria ser `Int -> Int -> Int` para o uso de applicativo.

O problema aqui é que o uso de Applicative é para sequências de computações que podem ter efeitos mas que são independentes entre si.

Queremos agora uma sequência de computações com efeito mas que uma computação dependa da anterior.

Monads

Precisamos de uma função que capture nosso padrão de `case of`:

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

O nome significa que estamos vinculando o resultado da computação de `mx` ao argumento da função `g`.

Monads

No Haskell esse operador é conhecido como `bind` e definido como:

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Com isso podemos reescrever eval como:

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

Monads

```
> eval (Div (Val (Just 4)) (Val (Just 2)))
  => (Just 4) >>= \n ->
            (Just 2) >>= \m -> safeDiv n m
  => (Just 2) >>= \m -> safeDiv 4 m
  => safeDiv 4 2
```

Monads

```
> eval (Div (Val (Nothing)) (Val (Just 2)))
  => Nothing >>= \n ->
              (Just 2) >>= \m -> safeDiv n m
  => Nothing
```

Monads

```
> eval (Div (Val (Just 4)) (Val (Nothing))
  => (Just 4) >>= \n ->
            (Just 2) >>= \m -> safeDiv n m
  => Nothing >>= \m -> safeDiv 4 m
  => Nothing
```

Monads

Generalizando, uma expressão construída com o operador `(>>=)` tem a seguinte estrutura:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
...
mn >>= \xn ->
f x1 x2 ... xn
```

Indicando um encadeamento de computação sequencial para chegar a uma aplicação de função. Esse operador garante que se uma computação falhar, ela para imediatamente e reporta a falha (em forma de `Nothing`, `[]`, etc.)

Essa mesma expressão pode ser escrita com a notação chamada **do-notation**:

```
do x1 <- m1
   x2 <- m2
   ...
   xn <- mn
   f x1 x2 ... xn
```

Monads: Syntactic Sugar

Com isso podemos reescrever `eval` novamente como:

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

Que captura uma sequência de computações que devem respeitar a ordem, são dependentes e podem falhar. Uma notação imperativa?

Monads

Esse tipo de operação forma uma nova classe de tipos denominada **Monads**:

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

   return = pure
```

Além do operador `bind` ela redefine a função `pure` com o nome de `return`.

Monad Maybe

Já escrevemos a definição de `Monad Maybe` mas podemos deixá-la mais clara utilizando Pattern Matching:

```
instance Monad Maybe where
   Nothing  >>= _ = Nothing
   (Just x) >>= f = f x
```

Monad Lists

Listas também fazem parte da classe Monad, inclusive já fizemos uso de *bind* para listas anteriormente:

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

Monad Lists

Por exemplo, para gerar todas as combinações de elementos de duas listas pode ser escrito como:

```haskell
pares :: [a] -> [b] -> [(a,b)]
pares xs ys = xs >>= \x ->
              ys >>= \y ->
              return (x,y) -- [(x,y)]
```

Monad Lists

Ou em *do-notation*:

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

Monad Lists

```
> pares [1,2] [3,4]
  => [1,2] >>= \x ->
           [3,4] >>= \y ->
                [(x,y)]
  => [x' | x  <- [1,2],
           x' <- \x -> [3,4] >>= \y -> [(x,y)]]
  => [x' | x  <- [1,2],
           x' <- \x -> [y' | y <- [3,4], y' <- [(x,y)]]
  => [x' | x  <- [1,2],
           x' <- \x -> [y' | y' <- [(x,3), (x,4)]]
  => [x' | x  <- [1,2],
           x' <- \x -> [(x,3), (x,4)]]
  => [x' | x' <- [(1,3),(1,4),(2,3),(2,4)]]
  => [(1,3),(1,4),(2,3),(2,4)]
```

A compreensão de listas surgiu a partir da notação *do*:

```
pares xs ys = [(x,y) | x <- xs, y <- ys]
  == do x <- xs
        y <- ys
        return (x,y)
```

Exercício

Escreva a instância de Monads para o tipo Pokémon:

```haskell
data Pokemon a = ATK a | DEF a | AD a
               deriving (Show, Eq)
```

Leis do Monad

A definição de um Monad deve seguir três leis:

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

As duas primeiras leis indicam que `return` é a identidade do Monad:

```
f :: a -> m b
x :: a
return x :: m a
return x >>= f = f x
```

Leis do Monad

As duas primeiras leis indicam que `return` é a identidade do Monad:

```
mx     :: m a
return :: a -> ma
mx >>= return :: m a
```

Leis do Monad

A última lei mostra como deve ser feito a associatividade do operador *bind*:

```haskell
mx :: m a
f  :: a -> m b
g  :: b -> m c

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

Funções de alta ordem para Monads

As funções de alta ordem possuem versões para Monads na biblioteca `Control.Monad`:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

mapM

Digamos que tenho a seguinte função:

```haskell
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing
```

mapM

Podemos aplicar `mapM` para obter:

```
> mapM conv ”1234”
Just [1,2,3,4]
> mapM conv ”12a4”
Nothing
```

filterM

Também temos a versão monádica de `filter`:

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filter M p xs
                      return (if b then x:ys else ys)
```

filterM

Podemos gerar o conjunto das partes com essa função e o Monad List:

```
> filterM (\x -> [True, False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```