Functors, Applicatives e Monads

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Recapitulando...

Construtores de Tipos

UFABC

- 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:

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

UFABC

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

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

Considerações

Considerações

UFABC

- 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

UFABC

- 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

UFABC

- 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:

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

Maybe como resultados que podem falhar

UFABC

- 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

UFABC

- 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
```

UFABC

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

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

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

UFABC

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

```
1  chrLista     = map chr
2
3  isLowerLista = map isLower
```

UFABC

- 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** 😕

Functors

UFABC

- Functors são morfismos que transformam os morfismos de uma categoria inteira (Tipos) em morfismos de outra (Maybe).
- No Haskell o que temos são endofunctors.

Mais sobre isso no curso de teoria das categorias para programadores!

UFABC

- No Haskell um Functor é definido como uma classe de tipos:

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

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

Functors de lista

UFABC

- Para as listas nós já temos o functor:

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

Functors de Maybe

UFABC

- Para o Maybe definimos da seguinte forma:

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

UFABC

- Agora podemos fazer:

```
1  > fmap chr Nothing
2  Nothing
3  > fmap chr (Just 65)
4  Just 'A'
5  > fmap (+1) (Just 65)
6  Just 66
```

UFABC

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

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

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

UFABC

- 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

UFABC

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

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

UFABC

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

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

- Isso nos ajuda a compor funções que serão mapeadas.

Operador Functor

UFABC

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

```
1  > chr <$> Nothing
2  Nothing
3  > chr <$> (Just 65)
4  Just 'A'
5  > (+1) <$> [1,2,3]
6  [2,3,4]
```

Operador Functor

UFABC

- O operador `(<$>)` nada mais é que a definição de `fmap` infixa

```
1  > (+1) <$> [1,2,3]
2  [2,3,4]
3  > (+1) `fmap` [1,2,3]
4  [2,3,4]
5  -- Para o Functor [] é o mesmo que
6  > (+1) `map` [1,2,3]
7  [2,3,4]
8  > map (+1) [1,2,3]
9  [2,3,4]
```

Exercício 1

UFABC

- 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.:

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

- Escreva a instância de Functor para esse tipo.

Applicative Functors

Functors para funções de múltiplos argumentos

UFABC

- Ok, digamos que eu queira fazer:

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

Família fmap

UFABC

- Idealmente teríamos:

```haskell
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
```

Família fmap

UFABC

- 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...

UFABC

- Podemos resolver isso através do uso de *currying*:

```haskell
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
```

UFABC

- Isso é denominado Applicative Functor, ou simplesmente 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
```

UFABC

- E com isso podemos fazer:

```
1 > pure (+) <*> [1,2] <*> [3,4] -- não dá esse resultado
2 [4,6]
3 > pure (+) <*> (Just 3) <*> (Just 2)
4 Just 5
```

UFABC

- 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

UFABC

- Para o tipo Maybe basta definirmos:

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

UFABC

- 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

UFABC

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

UFABC

- 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]
```

UFABC

- 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

UFABC

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

UFABC

- 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:

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

Operações com Listas

UFABC

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

```
1 -- Tb não dá esse resultado
2 > pure (+) <*> [1,2,3] <*> [4,5]
3 [5,7]
```

UFABC

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

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

UFABC

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

```haskell
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:

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

UFABC

- 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:

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

Sequenciamento

UFABC

```
> 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],[]]
[]
```

UFABC

- 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

UFABC

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

Múltiplas aplicações de funções

UFABC

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

```
1  > g = \x -> map (\f -> f x) [(+1), (*2), (`mod` 3)]
2  > g 1
3  [2,2,1]
4  > map g [1,2,3]
5  [[2,2,1],[3,4,2],[4,6,0]]
```

Múltiplas aplicações de funções

UFABC

- Uma forma mais natural é utilizar Applicatives:

```
1  > g' = sequenceA [(+1), (*2), (`mod` 3)]
2  > g' 1
3  [2,2,1]
4  > map g' [1,2,3]
5  [[2,2,1],[3,4,2],[4,6,0]]
```

Leis do Applicative

UFABC

- 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

UFABC

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

```
1  > v = safeDiv x y
2  > pure id <*> v
3    => Just id <*> v
4    => fmap id v
```

UFABC

- 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:

```
1  > pure (+) <*> pure 2 <*> pure 3 :: [Int]
2    => [(+)] <*> [2] <*> [3]
3    => [g x | g <- [(+)], x <- [2]] <*> [3]
4    => [(+2)] <*> [3]
5    => [g x | g <- [(+2)], x <- [3]]
6    => [5]
7    == pure (2 + 3) :: [Int]
```

Inversão

UFABC

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

```
1  > [(+3)] <*> pure 2
2    => pure ($ 2) <*> [(+3)]
3    => [($ 2)] <*> [(+3)]
4    => [g x | g <- [($ 2)], x <- [(+3)]]
5    => [($ 2) (+3)]
6    => (+3) ($ 2)
7    => (+3) 2
8    == 5
```

UFABC

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

```
1  > [dobra] <*> ([triplica] <*> [2,3])
2    => [dobra] <*> [6,9]
3    => [12,18]
4  > pure (.) <*> [dobra] <*> [triplica] <*> [2,3]
5    => [(dobra .)] <*> [triplica] <*> [2,3]
6    => [dobra . triplica] <*> [2,3]
7    => [12,18]
```

Exercício 2

UFABC

- Escreva a instância de Applicative para o tipo Pokémon:

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

Monads

UFABC

- 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
```

UFABC

- Para avaliar essa expressão podemos definir:

```haskell
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)
```

UFABC

- Porém, se fizermos:

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

UFABC

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

```haskell
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
```

UFABC

- Agora temos:

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

- Mas nosso código está confuso...

Applicative?

UFABC

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

```
1  > 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.

UFABC

- 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`.

UFABC

- No Haskell esse operador é conhecido como **bind** e definido como:

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

- Qualquer semelhança com o logo de Haskell, é mera coincidência 😄

UFABC

- Com isso podemos reescrever `eval` como:

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

UFABC

```
> 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
```

UFABC

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

UFABC

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

UFABC

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

```
1  m1 >>= \x1 ->
2  m2 >>= \x2 ->
3  ...
4  mn >>= \xn ->
5  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.)

UFABC

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

```
1  do x1 <- m1
2     x2 <- m2
3     ...
4     xn <- mn
5     f x1 x2 ... xn
```

UFABC

- 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? 

UFABC

- 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:

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

Monad Lists

UFABC

- 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]
```

UFABC

- 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)]
```

UFABC

- Ou em *do-notation*:

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

Monad Lists

UFABC

```
> 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)]
```

UFABC

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

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

Exercício 3

UFABC

- Escreva a instância de Monads para o tipo Pokémon:

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

UFABC

- A definição de um Monad deve seguir três leis:

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

Leis do Monad

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

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

UFABC

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

```
1 mx      :: m a
2 return :: a -> ma
3 mx >>= return :: m a
```

UFABC

- 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))
```

UFABC

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

```haskell
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)
```

UFABC

- Digamos que tenho a seguinte função:

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

UFABC

- Podemos aplicar **mapM** para obter:

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

UFABC

- Também temos a versão monádica de `filter`:

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

UFABC

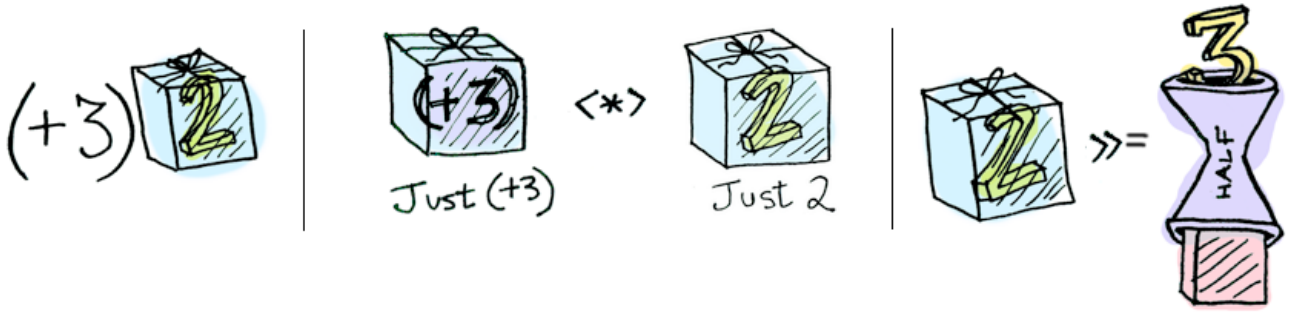
- Podemos gerar o conjunto das partes com essa função e o Monad List:

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

Veja também...

Outras referências interessantes

- Functors, Applicatives, And Monads In Pictures



- Abstraction, intuition, and the “monad tutorial fallacy”
- Monad tutorials timeline