

Teoria das Categorias para Programadores

Fabrício Olivetti de França

10 de Agosto de 2019



Topics

1. Functors
2. Bifunctors
3. Tipo Função
4. Monoids Livres
5. Transformação Natural
6. Functors Representáveis
7. Lema de Yoneda

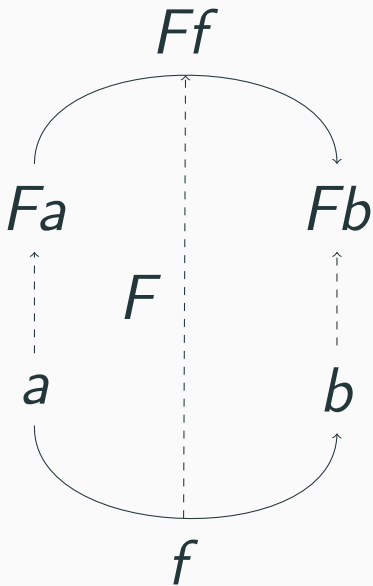
Functors

Um outro conceito de Teoria das Categorias que pode ser diretamente relacionado com programação é o **Functor**.

Functor é um mapa de objetos e morfismos de uma categoria C para uma categoria D :

- $F : C \rightarrow D$ é um functor de C para D
 - se $a, b, f \in C$, sendo $f : a \rightarrow b$
 - então $Fa, Fb, Ff \in D$ e $Ff : Fa \rightarrow Fb$.

Functors



Functors

Esse mapa não apenas transforma uma categoria em outra, mas também preserva sua estrutura, ou seja, tanto os morfismos identidades como as composições são mantidas intactas:

$$Fid_a = id_{F_a}$$

e

$$h = g.f \implies Fh = Fg.Ff$$

Functors em Linguagem de Programação

Pensando na categoria dos tipos, temos na verdade **endofunctors** que mapeiam a categoria dos tipos para ela mesma.

Functors em Linguagem de Programação

Podemos pensar em um Functor F como um tipo paramétrico:

- Dado um tipo a eu crio um tipo Fa que **contém** valores de a .

Em outra palavra é um **container**.

Containers

Um exemplo de container é uma lista, podemos ter uma lista de Int, lista de Char, etc.

Quais outros containers vocês conhecem?

Functor Lista

Vamos revisar a definição de uma lista em Haskell:

```
data List a = Empty | a : (List a)
```

Uma lista do tipo `a` ou é vazia (`Empty`) ou tem um elemento do tipo `a` seguido por outra lista do mesmo tipo.

Functor Lista

Pensando que um Functor é um container, então poderíamos dizer que $F = List$.

Porém, um Functor deve manter toda a estrutura do tipo contido na lista.

Ou seja, para qualquer $f : a \rightarrow b$, devo ter um $Ff : Fa \rightarrow Fb$.

Functor Lista

Para termos um Functor precisamos ter um mapa de morfismos. A definição de um Functor em Haskell evidencia isso:

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

fmap recebe uma função de a para b e retorna uma função de F a para F b. Isso é chamado de **lift**.

Functor Lista

Para simplificar, podemos remover o segundo par de parênteses e ler de outra forma:

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

Dada uma função de a para b e um Functor de a, eu retorno um Functor de b.

Functor Lista

Diante dessa segunda leitura, como você implementaria a função `fmap` para listas? (em qualquer linguagem)

Functor Lista

Em Haskell temos:

```
instance Functor List where
    fmap f []      = []
    fmap f (x:xs) = f x : fmap f xs
```


Functor Lista

Aplicando no seguinte exemplo temos:

```
let xs = 1 : 2 : 3 : []
```

```
fmap show xs
```

```
-- fmap show (1:xs) = show 1 : fmap show xs
```

```
-- = show 1 : fmap show (2:xs)
```

```
-- = show 1 : show 2 : fmap show xs
```

```
-- = show 1 : show 2 : show 3 : fmap show []
```

```
-- = show 1 : show 2 : show 3 : []
```

Tudo é um Functor

Discutimos alguns exemplos de containers. . . mas quais os containers mais simples que vocês conseguem imaginar?

Const Functor

O mais simples é aquele que não armazena nada! Ele é conhecido como Const Functor e simplesmente guarda um mesmo valor **sempre**:

```
data Const b a = Const b
```

Como você implementaria fmap para esse Functor?

Const Functor

```
instance Functor (Const b) where  
    fmap _ (Const x) = Const x
```

Ele será útil para automatizarmos a tarefa de construir um Functor!

Identity Functor

O segundo Functor mais simples é aquele que guarda um único valor do tipo `a`:

```
data Identity a = Identity a
```

Como você implementaria `fmap` para esse Functor?

Identity Functor

```
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
```

O Functor Identity a é isomorfo ao tipo a. Podemos então dizer que todo tipo é um Functor!

Construindo novos Functors

Temos um Functor que descarta informação (`Const`) e outro que guarda um único valor (`Identity`). Como construímos um container que **ou** guarda nada ou guarda apenas um valor?

Construindo novos Functors

```
data NadaOuUm a = Either (Const () a) (Identity a)
```

Como você definiria a função `fmap`?

Construindo novos Functors

```
instance Functor NadaOuUm where
  fmap _ (Left Const ()) = Left Const ()
  fmap f (Right Identity x) = Right Identity (f x)
```

Construindo novos Functors

Esse tipo `NadaOuUm` é isomorfo a qual outro tipo que vimos anteriormente?

Construindo novos Functors

```
1 f :: Maybe a -> NadaOuUm a
2 f Nothing = Const ()
3 f (Just x) = Identity x
4
5 g :: NadaOuUm a -> Maybe a
6 g (Left (Const ())) = Nothing
7 g (Right (Identity x)) = Just x
8
9 f . g = id
10 g . f = id
```

Functor Maybe

```
instance Functor Maybe where
  -- fmap _ (Const ()) = Const ()
  fmap _ Nothing = Nothing

  -- fmap f (Identity x) = Identity (f x)
  fmap f (Just x) = Just (f x)
```

É realmente um Functor?

Precisamos verificar se nossa definição obedece as propriedades de um Functor:

```
fmap id = id
```

```
fmap (g . f) = fmap g . fmap f
```

É realmente um Functor?

```
fmap id Nothing = id Nothing  
Nothing = Nothing
```

```
fmap id (Just x) = id (Just x)  
Just (id x) = Just x  
Just x = Just x
```

É realmente um Functor?

```
fmap (g . f) Nothing = (fmap g . fmap f) Nothing
Nothing = fmap g (fmap f Nothing)
Nothing = fmap g Nothing
Nothing = Nothing
```

É realmente um Functor?

```
fmap (g . f) (Just x) = (fmap g . fmap f) (Just x)
Just ((g . f) x) = fmap g (fmap f (Just x))
Just (g (f x)) = fmap g (Just (f x))
Just (g (f x)) = Just (g (f x))
```


Functor Writer

Relembrando a definição de `Writer` (um pouco diferente da aula anterior):

```
data Writer s a = Writer a s
```

Como reescrever utilizando `Const` e `Identity`?

Functor Writer

```
type Writer s a = (Identiy a, Const s a)
```

Como escrevemos a definição de fmap para esse tipo?

Functor Writer

```
instance Functor (Writer s) where
    fmap f (Writer x s) = Writer (f x) s
```

Functor via compilador

A construção de um Functor é um processo mecânico, podemos derivar automaticamente pelo compilador. No compilador ghc do Haskell podemos fazer:

```
{-# LANGUAGE DeriveFunctor #-}
```

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

Functor via compilador

Essa construção funciona pois os Functors podem ser compostos:

```
instance Functor Maybe where
```

```
  fmap f (Left x)  = Left (fmap f x)
```

```
  fmap f (Right y) = Right (fmap f y)
```

Ou seja, definimos o `fmap` em função de `fmap` de outros Functors.

Functor Reader

Um outro container interessante é o Reader, que é representado por uma função:

```
type Reader r a = r -> a
```

Functor Reader

Dada uma função $a \rightarrow b$ e um `Reader r a`, `fmap` deve criar um `Reader r b`.

Functor Reader

Dada uma função $a \rightarrow b$ e um $r \rightarrow a$, `fmap` deve criar um $r \rightarrow b$.

Functor Reader

```
instance Functor (Reader r) where
  -- fmap :: (a -> b) -> (Reader r a)
    -> (Reader r b)
  -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
  fmap = ???
```

Functor Reader

```
instance Functor (Reader r) where
  -- fmap :: (a -> b) -> (Reader r a)
    -> (Reader r b)
  -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
  fmap = (.)
```

Functor Reader

Se funções são Functors, funções podem ser interpretadas como containers!

Concordam??

Functor Reader

Funções puras podem ser memoizadas, ou seja, ter seus resultados armazenados em um container.

O inverso também é válido, um container pode ser representado como uma função.

Functor Reader

Com essa intuição, podemos definir tipos infinitos (ex.: *stream* de dados):

```
-- lista infinita com os números naturais
```

```
nat = [1..]
```

```
nat = 1 : fmap (+1) nat
```

```
-- 1 : (+1) 1 : (+1) (+1) 1 : ...
```

Composição de Functors

Podemos compor dois ou mais Functors criando estruturas mais complexas de forma simplificada graças as propriedades do Functor:

```
1 maybeTail :: [a] -> Maybe [a]
2 maybeTail [] = Nothing
3 maybeTail (x:xs) = Just xs
4
5 square :: Integer -> Integer
6 square x = x*x
7
8 xs :: [Integer]
9 xs = [1 .. 10]
10
11 fmap (fmap square) (maybeTail xs)
12 =
13 (fmap . fmap) square (maybeTail xs)
```

Functors em outras linguagens

A definição de `fmap` em C++ para o tipo `optional` pode ser escrita como:

```
1 template<class A, class B>
2 std::optional<B> fmap(std::function<B(A)> f,
3                       std::optional<A> opt)
4 {
5     if (!opt.has_value())
6         return std::optional<B>{};
7     else
8         return std::optional<B>{ f(*opt) };
9 }
```

Functors em outras linguagens

E para o tipo vector:

```
1 template<class A, class B>
2 std::vector<B> fmap(std::function<B(A)> f, std::vector<A> v)
3 {
4     std::vector<B> w;
5     std::transform(std::begin(v), std::end(v),
6                     std::back_inserter(w) , f);
7     return w;
8 }
```

Functors em outras linguagens

```
1  int dobra(int x) {
2      return 2*x;
3  }
4
5  int main()
6  {
7      std::optional<int> o1, o2;
8      std::function<int(int)> f = dobra;
9
10     std::vector<int> v{ 1, 2, 3, 4 };
11
12     o1 = {3};
13     o2 = {};
14
15     std::cout << fmap(f, o1).value_or(-1) << std::endl;
16     std::cout << fmap(f, o2).value_or(-1) << std::endl;
17     for (auto const& c : fmap(f, v))
18         std::cout << c << ' ';
19     std::cout << std::endl;
20
21     return 0;
22 }
```

Functors em outras linguagens

Em Python temos que usar `singledispatch` com os parâmetros invertidos, pois o tipo paramétrico deve ser o primeiro parâmetro:

```
1 class Maybe:
2     def __init__(self, x = None):
3         self.val = x
4
5 @singledispatch
6 def fmap(a, f):
7     print("Not implemented for" + str(type(a)))
```

Functors em outras linguagens

```
1 @fmap.register(list)
2 def _(l, f):
3     return list(map(f, l))
4
5 @fmap.register(Maybe)
6 def _(m, f):
7     if m.val is None:
8         m.val = None
9     else:
10        m.val = f(m.val)
11    return m
```

Functors em outras linguagens

```
1 f = lambda x: x*2
2
3 l = [1,2,3]
4 m1 = Maybe(2)
5 m2 = Maybe()
6
7 print(fmap(l, f))
8 print(fmap(m1, f).val)
9 print(fmap(m2, f).val)
```

Bifunctors

Bifunctors

Vimos anteriormente que muitos Functors são aplicados em tipos paramétricos com **dois** parâmetros. Por exemplo, temos os dois tipos algébricos fundamentais: `Either a b` e `Pair a b`.

Bifunctors

Nesses casos devemos decidir qual parâmetro fica fixo e em qual aplicamos a função. Convencionamos de fixar o primeiro dos tipos paramétricos.

Por que não criar um Functor que permite aplicar funções em ambos os parâmetros?

Bifunctors

```
class Bifunctor f where
  bimap :: (a -> c) -> (b -> d) -> (f a b -> f c d)
```


Bifunctors

Podemos definir um Bifunctor para os tipos `Either` e para tuplas como:

```
1 instance Bifunctor Either where
2     bimap f _ (Left x) = Left (f x)
3     bimap _ g (Right y) = Right (g y)
4
5 instance Bifunctor (,) where
6     bimap f g (x, y) = (f x, g y)
```

Bifunctors

No Haskell o Bifunctor também pode ser definido através das funções `first` e `second` com implementações padrão:

```
1 class Bifunctor f where
2     bimap :: (a -> c) -> (b -> d) -> (f a b -> f c d)
3     bimap f g = first f . second g
4
5     first :: (a -> c) -> (f a b -> f c b)
6     first f = bimap f id
7
8     second :: (b -> d) -> (f a b -> f a d)
9     second g = bimap id g
```

Bifunctors

Para os tipos `Either` e `Pair` essas funções ficariam:

```
1 instance Bifunctor Either where
2     first f (Left x)  = Left (f x)
3     first _ (Right y) = Right y
4
5     second _ (Left x)  = Left x
6     second g (Right y) = Right (g y)
7
8 instance Bifunctor (,) where
9     first f (x, y) = (f x, y)
10    second g (x, y) = (x, g y)
```

Functors Covariantes e Contravariantes

Os Functors que vimos até então tem um nome mais específico, eles são chamados de **Covariantes**. Lembrando do Functor Reader definido como:

```
type Reader r a = r -> a
```

```
instance Functor (Reader r) where  
  fmap f g = f . g
```

Functors Covariantes e Contravariantes

Se quiséssemos definir um Bifunctor para o tipo função, teríamos que definir primeiro um Functor para o tipo `Op`:

```
type Op r a = a -> r
```

```
instance Functor (Op r) where
```

```
  fmap :: (a -> b) -> (Op r a) -> (Op r b)
```

```
=
```

```
  fmap :: (a -> b) -> (a -> r) -> (b -> r)
```

Não tem como definirmos uma função `fmap` com essa assinatura!

Functors Covariantes e Contravariantes

Precisamos de um argumento do tipo $b \rightarrow a$. Isso é definido pelo Functor **Contravariante** da categoria oposta ao Covariante:

```
class Contravariant f where
  contramap :: (b -> a) -> (f a -> f b)

instance Contravariant (Op r) where
  -- (b -> a) -> Op r a -> Op r b
  contramap f g = g . f
```

Exemplo de aplicação: Composição de Comparadores

Imagine que temos as seguintes estruturas de dados:

```
1 data Person = Person {name :: String
2                       , age  :: Int
3                       }
4
5 data Employee = Employee {tag :: Int
6                          , person :: Person
7                          , salary :: Double
8                          }
```

Exemplo de aplicação: Composição de Comparadores

Dada a função `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`, podemos fazer:

```
1 cmpAge :: Int -> Int -> Ordering
2 cmpAge x y | x < y = LT
3             | x > y = GT
4             | x == y = EQ
5
6 cmpPerson :: Person -> Person -> Ordering
7 cmpPerson x y = cmpAge (age x) (age y)
8
9 cmpEmployee :: Employee -> Employee -> Ordering
10 cmpEmployee x y = cmpAge ((age. person) x) ((age.person) y)
```

Exemplo de aplicação: Composição de Comparadores

Observando essas funções, percebemos um padrão de repetição em nossos códigos. Idealmente poderíamos ter uma função f que aplica uma função em nossos registros antes de aplicar a função de comparação:

```
1 cmpPerson    = f age cmpAge
2 cmpEmployee = f (age.person) cmpAge
```

Exemplo de aplicação: Composição de Comparadores

Qual deve ser a assinatura dessa função?

```
f :: (?? -> ??) -> (Int -> Int -> Ordering)
    -> (?? -> ?? -> Ordering)
```

Exemplo de aplicação: Composição de Comparadores

Vamos generalizar o tipo `Int` em um tipo genérico `a`:

```
f :: (?? -> ??) -> (a -> a -> Ordering)
    -> (?? -> ?? -> Ordering)
```

Exemplo de aplicação: Composição de Comparadores

O primeiro argumento recebe tipo b e transforma em um tipo a , pois sabemos ordenar o tipo a :

```
f :: (b -> a) -> (a -> a -> Ordering)
   -> (?? -> ?? -> Ordering)
```

Exemplo de aplicação: Composição de Comparadores

Finalmente, podemos criar uma função que sabe ordenar o tipo `b`:

```
f :: (b -> a) -> (a -> a -> Ordering)
   -> (b -> b -> Ordering)
```

Exemplo de aplicação: Composição de Comparadores

Digamos que a função de comparação é um Functor do tipo `Order a`, com isso temos:

```
f :: (b -> a) -> Order a -> Order b
```

Já vimos algo parecido com isso...

Exemplo de aplicação: Composição de Comparadores

`contramap :: (b -> a) -> F a -> F b`

Exemplo de aplicação: Composição de Comparadores

```
type Order a = a -> a -> Ordering
```

```
instance Contravariant Order where
```

```
  --contramap :: (b -> a) -> (a -> a -> c)  
  -> (b -> b-> c)
```

```
  contramap f c = \x y -> c (f x) (f y)
```


Exemplo de aplicação: Composição de Comparadores

Com isso nossa ordenação fica:

```
cmpPerson' = contramap age cmpAge
```

```
cmpEmployee' = contramap (age.person) cmpAge
```

Composição de Comparadores

Escreva o Bifunctor em a , b para os seguintes tipos (p , q são Functors):

```
data K2 c a b = K2 c
```

```
data Fst a b = Fst a
```

```
data Snd a b = Snd b
```

```
data p q a b = Left (p a b) | Right (q a b)
```

```
data p q a b = (p a b , q a b)
```

Tipo Função

Tipo Função

Em Teoria das Categorias, definimos $C(a, b)$ como o conjunto de morfismos iniciando em a e terminando em b .

Na categoria dos tipos um morfismo é uma função que recebe um argumento do tipo a e retorna um tipo b .

Se $a \rightarrow b$ representa o conjunto de funções com essa assinatura, podemos definir um Tipo Função.

Tipo de Dados Algébrico Exponencial

A interpretação algébrica de um tipo função é a de um exponencial. Vamos verificar isso com funções de `Bool` para `a` e de `a` para `Bool`:

```
f :: Bool -> a
```

```
g :: a -> Bool
```

Tipo de Dados Algébrico Exponencial

De quantas maneiras podemos definir a função f ?

Tipo de Dados Algébrico Exponencial

A entrada é definida pelos dois possíveis valores True e False, portanto f é definida por um par de valores de a , ou (a, a) .

Com isso temos a^2 definições diferentes para essa função.

Tipo de Dados Algébrico Exponencial

A função g deve definir um valor `True` ou `False` para cada um dos valores de a , ou seja, é uma tupla $(\text{Bool}, \text{Bool}, \text{Bool}, \dots, \text{Bool})$ com a elementos.

Analogamente, isso é equivalente a 2^a possíveis definições.

Tipo de Dados Algébrico Exponencial

O tipo função $a \rightarrow b$ é representada por uma exponencial b^a , indicando as possíveis combinações de entrada e saída para essa assinatura de função.

Vamos verificar se esse tipo também obedece as propriedades algébricas de uma exponenciação.

Potência de Zero

Uma função $a^0 = 1$ tem a assinatura:

```
f :: Void -> a
```

que já vimos possuir apenas uma definição que é a função absurd.

Potência envolvendo Um

Analogamente, uma função $a^1 = a$ tem a assinatura:

`f :: () -> a`

que é a função `unit` que seleciona um valor de `a`, portanto possui a definições diferentes.

Potência envolvendo Um

Já a função $1^a = 1$ tem como assinatura e única definição:

```
const :: a -> ()
```

```
const x = ()
```

Somas de Exponenciais

Uma função a^{b+c} :

```
f :: Either b c -> a
```

```
f (Left x) = ...
```

```
f (Right y) = ...
```

deve ser definida para os casos $b \rightarrow a$ e $c \rightarrow a$.

Temos que definir um par de funções, o que é compatível com a propriedade da exponenciação $a^{b+c} = a^b \cdot a^c$.

Exponenciais de Exponenciais

Mostre que $(a^b)^c = a^{bc}$.

Exponenciais de Exponenciais

Uma função $(a^b)^c$ é interpretada como uma função que recebe um tipo c e retorna uma função de $b \rightarrow a$, ou seja, uma função de alta ordem:

$f :: c \rightarrow (b \rightarrow a)$

Exponenciais de Exponenciais

A função $f :: c \rightarrow (b \rightarrow a)$ é isomórfica a uma função que recebe um par (c, b) e retorna um a . Ou seja, $(a^b)^c = a^{(bc)}$

Currying

Em Haskell todas as funções de múltiplos argumentos são interpretadas como retornando uma função (denominada **curried**):

$$a \rightarrow (b \rightarrow c) = a \rightarrow b \rightarrow c$$

Currying

Isso fica claro na definição de uma função de duas variáveis em Haskell:

```
mult :: Int -> Int -> Int
```

```
mult x y = x*y
```

```
mult' :: Int -> (Int -> Int)
```

```
mult' x = \y -> x*y
```

Currying

Que se torna evidente quando fazemos uma aplicação parcial da primeira função:

```
dobra = mult 2
```

```
dobra :: Int -> Int
```

Currying

A biblioteca padrão do Haskell já tem a conversão das duas formas de definição de funções:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
```

```
curry f a b = f (a, b)
```

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

```
uncurry f (a, b) = f a b
```

Currying

Em C++ você pode fazer uma aplicação parcial de função utilizando o template `std::bind`:

```
int mult(int x, int y) {  
    return x*y;  
}  
  
using namespace std::placeholders;  
  
auto dobra = std::bind(mult, 2, _1);  
std::cout << dobra(4);
```

Currying

Em Python temos a função `partial`:

```
from functools import partial
```

```
def mult(x,y):  
    return x * y
```

```
dobra = partial(mult, 2)
```

Exponenciais de Exponenciais

Mostre que $(a \cdot b)^c = a^c \cdot b^c$.

Exponenciais sobre produtos

Também podemos ter uma função $(a \cdot b)^c$ que é representada por:

$$f :: c \rightarrow (a, b)$$

Isso é equivalente a um par de funções $c \rightarrow a$ e $c \rightarrow b$ (nossas funções p , q do tipo produto) que nos dá

$$(a \cdot b)^c = a^c \cdot b^c$$

Isomorfismo de Curry Howard

Complementando nossa tabela do isomorfismo de Curry Howard temos:

Algebra	Tipos	Lógica
0	Void	Falso
1	()	Verdadeiro
$a + b$	Either a b	$a \vee b$
$a * b$	(a, b)	$a \wedge b$
b^a	$a \rightarrow b$	$a \implies b$

A definição de uma função é isomórfica a definição de uma implicação.

Isomorfismo de Curry Howard

Por exemplo, nossa função `eval` com assinatura:

```
eval :: ((a->b), a) -> b
```

Pode ser traduzida como: “Se `b` segue de `a` e `a` é verdadeiro, então `b` é verdadeiro.”.

Isomorfismo de Curry Howard

Provamos essa proposição mostrando que esse tipo é habitável por algum valor:

`eval :: ((a->b), a) -> b`

`eval (f, x) = f x`

Isomorfismo de Curry Howard

Vamos tentar provar a proposição $(a \vee b) \implies a$, ou seja, se a ou b forem verdadeiros, então a é verdadeiro:

```
f :: Either a b -> a
```

```
f (Left x) = x
```

```
f (Right y) = ???
```

Na segunda definição eu tenho que prover uma expressão que funcione para qualquer que seja o tipo a . Impossível!

Podemos então usar funções definidas em Haskell como provas de proposições lógicas.

Monoids Livres

Monoids Livres

Relembrando o conceito de Monoid, temos um único objeto M equipados com um operador de multiplicação \cdot e um elemento neutro ϵ tal que:

$$a, b \in M, a \cdot b \in M$$

$$a \cdot \epsilon = \epsilon \cdot a = a$$

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) = a \cdot b \cdot c$$

ou em Haskell:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

Os números inteiros podem representar um Monoid com o operador de multiplicação e o elemento neutro 1 ou com o operador de soma e o elemento neutro 0.

Uma lista também é um Monoid com o operador de concatenação e o elemento neutro de lista vazia.

```
instance Monoid [] where
  mempty  = []
  mappend = (++)
```

No caso dos números inteiros, além das propriedades dos Monoids, temos algumas propriedades extras, por exemplo $2 \cdot 3 = 6$, ou seja, o elemento $2 \cdot 3$ é equivalente ao elemento 6, ambos pertencentes a M .

Já para o caso de listas, temos que $[2] ++ [3] \neq [6]$, ou seja, uma lista contém as propriedades dos Monoids e nada mais.

Um Monoid livre é um Monoid sem nenhuma propriedade adicional e que consegue gerar outros Monoids, partindo de um gerador e adicionando novas operações e propriedades.

Em programação podemos utilizar uma lista como um Monoid livre.

Dado um tipo a podemos gerar o Monoid $[a]$ enumerando todas as combinações de elementos agrupados em uma lista.

Monoids Livres

Vamos transformar uma lista de `Bool` em um Monoid livre, começamos com o elemento neutro da lista e as listas contendo um dos valores possíveis:

```
data Bool = True | False
```

```
m = [ [], [True], [False] ]
```

Monoids Livres

Ao aplicar o operador de concatenação entre cada par de elementos dessa lista inicial temos:

```
m = [ [], [True], [False], [True, True],  
      [True, False], [False, True], [False, False]  
      ]
```

Continuando tal operação, temos uma lista infinita de todos os elementos do nosso Monoid livre.

Para definir um novo Monoid a partir do Monoid livre, basta definirmos um morfismo $h :: [a] \rightarrow a$ de tal forma que:

$$h (\text{mappend } x \ y) = \text{mappend } (h \ x) \ (h \ y)$$

Monoids Livres

No nosso caso podemos definir `h = and` para a instância de Monoid de `Bool` com `&&`:

```
and :: [Bool] -> Bool
```

```
and [] = True
```

```
and (b:bs) = b && (and bs)
```

```
instance Monoid Bool where
```

```
  mempty = True
```

```
  mappend = (&&)
```

Monoids Livres

Podemos verificar que essa é uma função que segue a propriedade:

```
h ([True] ++ [False]) = (h [True]) && (h [False])
```

```
h [True, False] = True && False
```

```
False = False
```

Defina `h` para o Monoid do tipo `Bool` e operador `||`.

Transformação Natural

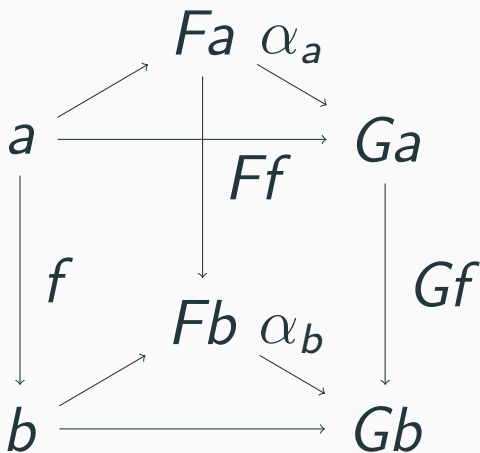
Transformação Natural

Dados dois Functors $F, G : C \rightarrow D$ da categoria C para a categoria D , chamamos o morfismo $\alpha_a :: Fa \rightarrow Ga$ uma **Transformação Natural** entre F e G :

$\alpha :: F \ a \ \rightarrow \ G \ a$

Transformação Natural

Tal transformação deve permitir o quadrado comutativo:



Transformação Natural

Isso permite criarmos a função $g :: F\ a \rightarrow G\ b$ de duas maneiras:

```
--  $G\ f . \alpha = \alpha . F\ f$   
g = fmap f . alpha = alpha . fmap f
```

Transformação Natural

A comutatividade implica que ao aplicar a primeira ou a segunda definição de g para um valor de a , o resultado deve ser exatamente o mesmo valor de b .

Transformação Natural

Considere a seguinte transformação natural de lista para o tipo Maybe:

```
safeHead :: [a] -> Maybe a  
safeHead []      = Nothing  
safeHead (x:xs) = Just x
```

Transformação Natural

Para ser uma transformação natural devemos garantir que `fmap`
`f . safeHead = safeHead . fmap f`:

```
fmap f . safeHead [] = safeHead . fmap f []
```

```
fmap f Nothing = safeHead []
```

```
Nothing = Nothing
```

```
fmap f . safeHead (x:xs) = safeHead . fmap f (x:xs)
```

```
fmap f . Just x = safeHead . f x
```

```
Just (f x) = Just (f x)
```

Functors Representáveis

Functors Representáveis

Relembrando a definição do Functor Reader:

```
type Reader a b = a -> b
```

```
instance Functor (Reader a) where  
  fmap f g = f . g
```

Functors Representáveis

Functor Representável é um Functor que mapeia uma categoria C na categoria dos conjuntos, denominada **Set**.

Um exemplo desse tipo de Functor é o **Reader** a (chamado de *hom-set*).

Functors Representáveis

Como consequência, todo Functor isomorfo a `Reader a` também é um Functor Representável:

```
alpha :: Reader a x -> F x
```

```
beta  :: F x -> Reader a x
```

```
alpha . beta = id :: F
```

```
beta . alpha = id :: Reader a
```

Functors Representáveis

Um contra-exemplo de um Functor representável é uma lista!
Vamos tentar montar os morfismos `alpha`, `beta` para esse
Functor partindo de `Reader Int`.

Functors Representáveis

Temos diversas escolhas para `alpha`, podemos aplicar a função em uma lista arbitrária de `Int`:

```
alpha :: Reader Int x -> [x]
alpha h = fmap h [12]
```


Functors Representáveis

A função inversa poderia ser implementada como:

```
beta :: [x] -> Reader Int x
```

```
beta xs = \y -> head xs
```

Functors Representáveis

Porém, para o caso de lista vazia essa função não funciona.
Não temos outra operação possível que funcione para um tipo x arbitrário.

Functors Representáveis

A definição de um Functor representável em Haskell é dada por:

```
class Representable f where
  -- a
  type Rep f :: *

  -- alpha :: Reader a x -> F x
  tabulate :: (Rep f -> x) -> f x

  -- beta  :: F x -> Reader a x
  index   :: f x -> Rep f -> x
```

Functors Representáveis

Nessa definição `Rep f` é o nosso tipo `a` em que nosso Functor é representável, o `*` significa que ele é um tipo não-paramétrico.

Substituindo `Rep f` nas funções, percebemos que `tabulate` representa nosso `alpha` e `index` nosso `beta`.

Functors Representáveis

Vejamos um exemplo de Functor representável com uma lista infinita não-vazia (fluxo contínuo de dados):

```
data Stream x = Cons x (Stream x)
```

```
instance Functor Stream where
```

```
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Functors Representáveis

A instância `Representable` fica:

```
instance Representable Stream where
  type Rep Stream = Integer
  tabulate f = Cons (f 0) (tabulate (f . (+1)))
  index (Cons b bs) n | n == 0      = b
                      | otherwise = index bs (n-1)
```

Functors Representáveis

A função `tabulate` cria uma lista infinita do tipo `Stream` iniciando com `f 0` e fazendo, na sequência, `f 1`, `f 2`, `f 3`, `...`

Functors Representáveis

A função `index` simplesmente recupera o n -ésimo elemento dessa lista. Esse Functor é uma generalização da memoização de funções cujo argumento é um inteiro positivo!

Functors Representáveis

Vamos verificar a capacidade de memoização da função Fibonacci:

```
1 f :: Integer -> Integer
2 f 0 = 0
3 f 1 = 1
4 f n = f (n-1) + f (n-2)
5
6 t :: Stream Integer
7 t = tabulate f
```

Functors Representáveis

```
1 main :: IO ()
2 main = do
3   args <- getArgs
4   case args of
5     [k] -> do
6         let i = read k
7             print $ f i
8             print $ f i
9     [k, "--rep"] -> do
10        let i = read k
11            print $ index t i
12            print $ index t i
13   _ -> error "Argumentos inválidos"
```

Functors Representáveis

Mensurando o tempo de execução desse programa utilizando ou não o Representable Functor, temos:

```
time ./fib 36  
real    0m4.176s
```

```
time ./fib 36 --rep  
real    0m2.098s
```

Lema de Yoneda

Lema de Yoneda

O Lema de Yoneda diz que:

“Uma transformação natural entre um hom-functor (Reader) e qualquer outro Functor F é completamente determinado ao especificar o valor do componente inicial do hom-functor.”

Lema de Yoneda

Para entender esse lema, vejamos duas transformações naturais:

$\text{alphaX} :: \text{Reader } a \ x \rightarrow F \ x$

$\text{alphaY} :: \text{Reader } a \ y \rightarrow F \ y$

Lema de Yoneda

Com tais transformações podemos desenhar o seguinte diagrama, que deve ser comutativo:

$$\begin{array}{ccc} \text{Reader } a \ x & \xrightarrow{\text{Reader } a \ f} & \text{Reader } a \ y \\ \downarrow \alpha_x & & \downarrow \alpha_y \\ F \ x & \xrightarrow{F \ f} & F \ y \end{array}$$

Lema de Yoneda

Lembrando que esse quadrado é comutativo, temos que

$$\text{alphaY} \cdot \text{Reader } a \text{ f} = (\text{F } f) \cdot \text{alphaX}$$

Lema de Yoneda

Como um Functor em uma função é o `fmap`, temos que:

$$\text{alphaY (fmap f) = fmap f . alphaX}$$

que vai ser aplicado a uma função `h`, ou seja:

$$\text{alphaY (fmap f h) = fmap f . alphaX h}$$

Lema de Yoneda

Como o `fmap` no Functor `Reader` a é apenas uma composição de funções, temos

$$\text{alphaY } (f \ . \ h) = \text{fmap } f \ . \ \text{alphaX } h$$

Lema de Yoneda

Sabendo que $h :: a \rightarrow x$ e fazendo $x = a$ temos que α_X é uma transformação natural entre um **Reader** $a \rightarrow a$ e um **F** a , podemos chamar de α_A .

Lema de Yoneda

Nesse caso a única opção para a função é $h = \text{id}$, temos então

$$\text{alpha}_Y f = (\mathbf{F} f) (\text{alpha}_A \text{id})$$

Lema de Yoneda

Com isso temos que $\text{alphaY } f = \text{fmap } f (\mathbf{F} \ a)$, a definição para nosso alpha fica:

```
alpha :: (a -> x) -> F x
```

```
alpha f = fmap f fa
```

Lema de Yoneda

Para um F a qualquer. Substituindo $f = \text{id}$, temos:

$\text{fa} :: F\ a$

$\text{fa} = \text{alpha id} = \text{fmap id fa}$

Lema de Yoneda

Ou seja, a quantidade de definições de α é a mesma que a quantidade de $F a$, sendo então isomorfas.

Lema de Yoneda

Podemos prontamente transformar um $(a \rightarrow x) \rightarrow F\ x$ em $F\ a$ fazendo `alpha id`.

Também podemos transformar um $F\ a$ em $(a \rightarrow x) \rightarrow F\ x$ fazendo `\h -> fmap h fa`.

Lema de Yoneda

Especializando no Functor identidade, ou seja, fazendo

$F = \text{Identity}$ temos:

$$(a \rightarrow x) \rightarrow x = a$$

Lema de Yoneda

Que pode ser lida como, dada uma função de alta ordem que recebe uma função $a \rightarrow x$ e retorna um valor de x , ela é isomorfa ao tipo a .

Lema de Yoneda

Também temos a definição de Co-Yoneda, o complemento do Yoneda, que diz:

$$(x \multimap a) \multimap F x = F a$$

Fusão de fmap

Fmap Fusion

A composição de `fmaps` no Haskell nem sempre é otimizada pelo compilador. Por exemplo, se tivermos o seguinte programa (código completo aqui:

```
1 data Tree a = Bin a (Tree a) (Tree a) | Nil deriving (Eq, Show)
2
3 t :: Tree Integer
4 t = go 1
5     where go r = Bin r (go (2*r)) (go (2*r + 1))
6
7 transform :: (Functor f, Num a) => f a -> f a
8 transform = fmap (^2) . fmap (+1) . fmap (*2)
9
10 printTree k = print . sumTree . takeDepth k
```

Fmap Fusion

Ao executar `printTree k (transform t)`, primeiro toda a árvore será percorrida para aplicar o mapa ($*2$), em seguida toda a árvore é percorrida novamente para aplicar ($+1$) e mais uma vez para aplicar (2).

Fmap Fusion

Sabemos que, pelas leis do Functor temos que

`fmap f . fmap g . fmap h = fmap (f.g.h)`. Podemos automatizar esse processo utilizando o Yoneda Embedding.

Fmap Fusion

Vamos criar o tipo Co-Yoneda `CY` que representa o lado esquerdo do lema de Yoneda (em sua versão complementar):

```
data CY f a = forall b . CY (b -> a) (f b)
```


Fmap Fusion

Precisamos definir uma instância de Functor para esse tipo:

```
instance Functor (CY f) where
  fmap f (CY b2a fb) = CY (f . b2a) fb
```

Fmap Fusion

E, utilizando o que aprendemos sobre Yoneda, podemos definir as funções `toCY` e `fromCY`:

```
toCY :: f a -> CY f a
```

```
toCY fa = CY id fa
```

```
fromCY :: Functor f => CY f a -> f a
```

```
fromCY (CY f fb) = fmap f fb
```

Fmap Fusion

Agora, se fizermos `(fromCY . transform . toCY) t`,
teremos:

```
(fromCY . transform . toCY) t
= (fromCY . transform) (CY id t)
= fromCY $ (fmap (^2) . fmap (+1) . fmap (*2))
                                     (CY id t)
= fromCY $ (fmap (^2) . fmap (+1)) (CY ((*2) . id) t)
= fromCY $ (fmap (^2)) (CY ((+1) . (*2) . id) t)
= fromCY (CY ((^2) . (+1) . (*2) . id) t)
= fmap ((^2) . (+1) . (*2) . id) t
```

Fmap Fusion

Isso é a base da biblioteca `Data.List.Stream` do Haskell que permite otimizar o uso de funções `fmap`, `filter`, `fold`, dentre outros.

Atividades para Casa

Atividades para Casa

1. Defina as instâncias de Functor vistas em aula em outra linguagem de programação.
2. É possível definir uma lista infinita em C++ ou Python? Crie tal definição.
3. Implemente o exemplo Functor Contravariante em outra linguagem de programação. Teve alguma vantagem em fazer dessa forma?
4. [opcional] Implemente um Functor Representável para uma árvore de jogos (siga o passo a passo da atividade no Github)