Teoria das Categorias para Programadores

Fabrício Olivetti de França

10 de Agosto de 2019

Topics

Functors

Functors

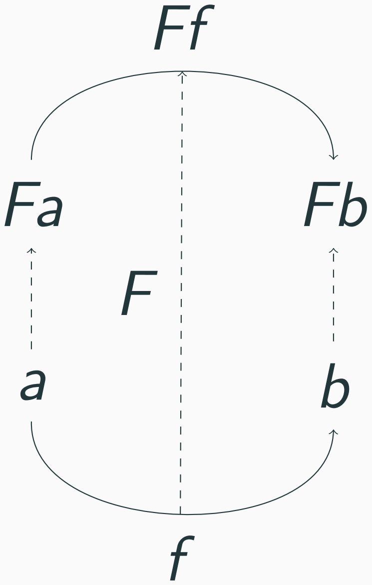
Um outro conceito de Teoria das Categorias que pode ser diretamente relacionado com programação é o **Functor**.

Functors

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

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

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:

$$Fid_a = id_{Fa}$$

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, ou seja.

Functors em Linguagem de Programação

Podemos pensar em um Functor F como um tipo paramétrico, ele é capaz de pegar qualquer tipo a de nossa categoria e criar 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:

```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 $f : a \to b$, devo ter um $Ff : Fa \to Fb$.

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:

```haskell
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:

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

Functor Lista

Aplicando no seguinte exemplo temos:

```haskell
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**:

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

Como você implementaria `fmap` para esse Functor?

Const Functor

```haskell
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

```haskell
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

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

Como você definiria a função `fmap`?

Construindo novos Functors

```haskell
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

```haskell
f :: Maybe a -> NadaOuUm a
f Nothing  = Const ()
f (Just x) = Identity x

g :: NadaOuUm a -> Maybe a
g (Left (Const ()))     = Nothing
g (Right (Identity x)) = Just x

f . g = id
g . f = id
```

Functor Maybe

```haskell
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:

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

É realmente um Functor?

```haskell
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):

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

Como reescrever utilizando `Const` e `Identity`?

Functor Writer

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

Como escrevemos a definição de `fmap` para esse tipo?

Functor Writer

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

Functor Reader

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

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

Functor Reader

Dado um `Reader r a` e uma função `a -> b` o `fmap` deve criar um `Reader r b`.

Functor Reader

Dado um `r -> a` e uma função `a -> b` o `fmap` deve criar um `r -> b`.

Functor Reader

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

Functor Reader

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

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

square :: Integer -> Integer
square x = x*x

xs :: [Integer]
xs = [1 .. 10]

fmap (fmap square) (maybeTail xs)
=
(fmap . fmap) square (maybeTail xs)
```

Functors em outras linguagens

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

```cpp
template<class A, class B>
std::optional<B> fmap(std::function<B(A)> f, std::optional<A> op
{
    if (!opt.has_value())
        return std::optional<B>{};
    else
        return std::optional<B>{ f(*opt) };
}
```

Functors em outras linguagens

E para o tipo vector:

```
template<class A, class B>
std::vector<B> fmap(std::function<B(A)> f, std::vector<A> v)
{
   std::vector<B> w;
   std::transform(std::begin(v), std::end(v), std::back_inserter
   return w;
}
```

Functors em outras linguagens

```
int dobra(int x) {
  return 2*x;
}

int main()
{
  std::optional<int> o1, o2;
  std::function<int(int)> f = dobra;

  std::vector<int>  v{ 1, 2, 3, 4 };

  o1 = {3};
  o2 = {};

  std::cout << fmap(f, o1).value_or(-1) << std::endl;
  std::cout << fmap(f, o2).value_or(-1) << std::endl;
  for (auto const& c : fmap(f, v))
    std::cout << c << ' ';
  std::cout << std::endl;

  return 0;
}
```

Functors em outras linguagens

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

```python
class Maybe:
    def __init__(self, x = None):
        self.val = x

@singledispatch
def fmap(a, f):
    print("Not implemented for" + str(type(a)))
```

Functors em outras linguagens

```python
@fmap.register(list)
def _(l, f):
    return list(map(f, l))

@fmap.register(Maybe)
def _(m, f):
    if m.val is None:
        m.val = None
    else:
        m.val = f(m.val)
    return m
```

Functors em outras linguagens

```python
f = lambda x: x*2

l = [1,2,3]
m1 = Maybe(2)
m2 = Maybe()

print(fmap(l, f))
print(fmap(m1, f).val)
print(fmap(m2, f).val)
```

Profunctors e 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

```haskell
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:

```
instance Bifunctor Either where
    bimap f _ (Left x)  = Left (f x)
    bimap _ g (Right y) = Right (g y)

instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

Tipo Função

Transformação Natural

Limites e Colimites

Monoids Livres