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. Functors Adjuntos

Functors Adjuntos

Isomorfismos de Categorias

Como podemos definir o isomorfismo entre duas categorias C, D?

Isomorfismos de Categorias

Os morfismos entre duas categorias são Functors, duas categorias são isomorfas se temos dois Functors, $R : C \to D$ e $L : D \to C$, tal que a composição deles forma o Functor identidade.

Isomorfismos de Categorias

$$R \circ L = I_D$$

e

$$L \circ R = I_C$$

Isomorfismos de Categorias

Lembrando que o Functor identidade é dado por:

```haskell
data Identity a = Identity a

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

Isomorfismos de Categorias

Para provar que duas categorias são isomórficas, precisamos definir transformações naturais entre a composição dos Functors e o Functor Identidade:

```
eta  :: Id    -> (R.L)
eta' :: (R.L) -> Id

eps  :: (L.R) -> Ic
eps' :: Ic    -> (L.R)
```

Functors Adjuntos

Dizemos que dois Functors `R, L` são adjuntos se eles possuem as transformações naturais `eta, eps`, mas sem a necessidade de existir `eta', eps'`.

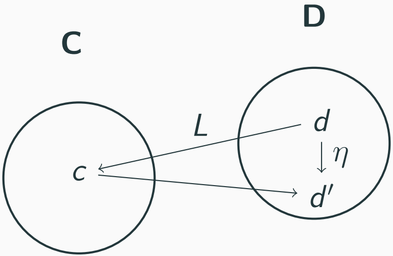
Functors Adjuntos

- O Functor `L` é denominado adjunto esquerdo (*left adjunct*)
- O Functor `R` é o adjunto direito (*right adjunct*)
- `eta` é chamado de `unit`
- `eps` de `counit`.

Denotamos $L \dashv R$ como L é o adjunto esquerdo de R.

Functors Adjuntos

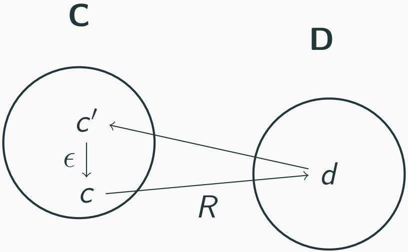
A função `eta` pega um objeto de `D` e faz um *passeio* entre as categorias `C, D` utilizando os Functors `R . L`, retornando um outro objeto de `D` encapsulado no functor `R . L`.




```
eta  :: Identity d -> (R.L) d
```

Functors Adjuntos

A função eps indica como extrair um `c` do Functor `L . R` passando pela categoria `D`.




```
eps  :: (L.R) c -> Identity c
```

Functors Adjuntos

Em outras palavras, o `unit` (também chamado de `return` e `pure` em outros contextos) permite introduzir um container ou Functor `R.L` em todo tipo `d`.

```
-- F = R . L
unit :: d -> F d
```

Functors Adjuntos

Por outro lado, o `counit` (em algumas linguagens conhecido como `extract`) permite retirar um objeto de um container ou Functor.

```
-- G = L . R
counit :: G c -> c
```

Functors Adjuntos

A classe de Functors adjuntos é definido como:

```haskell
class (Functor f, Representable y) =>
  Adjunction f u | f -> u, u -> f where
    unit   :: x -> u (f x)
    counit :: f (u x) -> x
```

Functors Adjuntos

A condição `f -> u, u -> f` é uma **dependência funcional** e implica que só pode existir uma única instância para `f` na esquerda e para `u` a direita.

Functors Adjuntos

Ou seja, se eu defino `Adjunction [] (Reader a)`, não posso definir `Adjunction Maybe (Reader a)` nem `Adjunction [] Maybe`.

Functors Adjuntos

Junto dessas duas funções também podemos definir essa classe através de `leftAdjunction` e `rightAdjunction`:

```
class (Functor f, Representable y) =>
  Adjunction f u | f -> u, u -> f where
    leftAdjunct  :: (f x -> y) -> x -> u y
    rightAdjunct :: (x -> u y) -> f x -> y
```

Functors Adjuntos

E elas se relacionam da seguinte forma:

```
unit          = leftAdjunct id
counit        = rightAdjunct id
leftAdjunct   = fmap g . unit
rightAdjunct = counit . fmap g
```

Functors Adjuntos

Existem poucos Functors pertencentes a **Hask** que são adjuntos, porém a combinação $L.R$ e $R.L$ formam os conceitos de Monads e Comonads, conforme veremos mais adiante.

Curry e Uncurry

Um exemplo de Functors adjuntos no Haskell são `(,a)` e `Reader a = (a ->)`. Podemos definir a instância como:

```haskell
instance Adjunction (,a) (Reader a) where
  -- unit :: c -> Reader a (c,a)
  unit c = \a -> (c, a)

  -- counit :: (Reader a c, a) -> c
  counit (f, c) = f c
```

Curry e Uncurry

Podemos definir `leftAdjunct` como:

```haskell
-- f = (,a); u = (a ->)
-- leftAdjunct :: ((x,a) -> y) -> x -> (a -> y)
leftAdjunct g x = (fmap g . unit) x
= (fmap g) (unit x)
= (fmap g) (\a -> (x, a))
= g . (\a -> (x, a))
= \a -> g (x,a)
```

Curry e Uncurry

ou...

```
-- f = (,a); u = (a ->)
-- leftAdjunct :: ((x,a) -> y) -> (x -> a -> y)
leftAdjunct g = \x -> \a -> g (x,a)
```

Curry e Uncurry

E `rightAdjunct` como:

```
-- f = (,a); u = (a ->)
-- rightAdjunct :: (x -> (a -> y)) -> (x,a) -> y
rightAdjunct g (x,a) = counit . (fmap g) (x,a)
= counit (g x, a)
= g x a
```

Curry e Uncurry

Podemos perceber que `leftAdjunct = curry` e `rightAdjunct = uncurry`.

Curry e Uncurry

Lembrando que `(,a) = Writer a` e, partindo da definição de Functors Adjuntos, podemos criar dois novos Functors com a composição de `Reader` com `Writer`:

Curry e Uncurry

```haskell
-- State a b = Reader a (Writer a b)
type State a b = a -> (b,a)

-- Store a b = Writer (Reader a b) a
type Store a b = (a -> b, a)
```

Esses Functors permitem a criação de estados e armazenamento em uma linguagem puramente funcional.