Teoria das Categorias para Programadores

Fabrício Olivetti de França

03 de Agosto de 2019

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

Topics

1. Teoria das Categorias
2. Categoria para Programadores
3. Monoids
4. Categoria Kleisli
5. Tipos de Dados Algébricos
6. Tipos Buracos

Teoria das Categorias

Teoria das Categorias

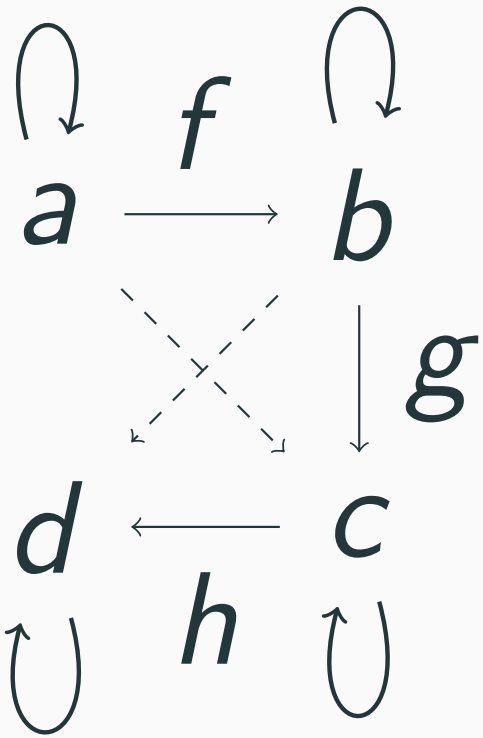
Teoria das Categorias é uma área da Matemática que formaliza, descreve e estuda estruturas abstratas com foco nas relações entre seus objetos.

Teoria das Categorias

Uma categoria C é definida como um conjunto de **objetos** e **morfismos** junto com um operador de composição ($\circ$) que garantem as seguintes propriedades:

- **Associatividade:** $h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$.
- **Identidade:** $f \circ id_A = id_B \circ f = f$.
- **Transitividade:** se $a, b, c \in C$ e $f : a \to b, g : b \to c$, então existe um $h = g \circ f : a \to c$.

Propriedades



Composição e Abstração

Tanto o conceito de **composição** como de **abstração** são conhecidos na Ciência da Computação e entre programadores.

Composição e Abstração

> Category theory is the study of compositionality: building big things out of little things, preserving guarantees.It would be utterly *astonishing* if this were not deeply useful for programming.We have barely scratched the surface of learning how to take advantage of this!
>
> — kenbot (@KenScambler) 15 de abril de 2019

Composição e Abstração

‘The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise’ - Edsger Dijkstra pic.twitter.com/S6UruJbBjF

— Computer Science (@CompSciFact) 4 de janeiro de 2018

Exemplo

```
1  char * inverte_str(char * orig) {
2      int len = 0;
3      char *ptr = orig, *dest, *pilha;
4      int i, topo = -1;
5
6      while (*ptr != '\0') ++ptr;
7      len = ptr - orig;
8
9      dest = malloc(sizeof(char)*(len+1));
10     pilha = malloc(sizeof(char)*len);
11
12     for (i=0; i<len; ++i) {
13         pilha[++topo] = orig[i];
14     }
15
16     i = 0;
17     while (topo != -1) dest[i++] = pilha[topo--];
18
19     dest[len] = '\0';
20     free(pilha);
21     return dest;
22 }
```

Exemplo com Modularização e Abstração

```
1  char * inverte_str(char * orig) {
2      int len = strlen(orig);
3      pilha * p = cria_pilha();
4      char * dest;
5
6      dest = cria_str(len);
7
8      while (*orig != '\0') {
9          empilha(p, *orig);
10         ++orig;
11     }
12
13     while (!vazia(p)) {
14         *dest = desempilha(p);
15         ++dest;
16     }
17
18     return dest;
19 }
```

Utilizando composição

```
1  char * inverte_str(char * orig) {
2      pilha * p = cria_pilha();
3      return desempilha_str(empilha_str(p, *orig));
4  }
```

Pergunta

Quais vantagens vocês percebem nessa última versão?

Resposta

- Construir o conceito de pilha uma única vez, utilizar em diversos contextos.
- Testar a corretude de cada módulo independentemente.
- Código declarativo
- Número menor de variáveis por módulo

Fim do curso?

O uso de abstração e composição é apenas o começo...

Fim do curso?

O estudo de Teoria das Categorias também compreende a identificação de padrões recorrentes.

Esses padrões são úteis para criações de estruturas genéricas que permitem lidar com diversos problemas recorrentes em programação.

Pergunta

Como você definiria a categoria das páginas Web? Quem são os objetos e morfismos? As propriedades são atendidas?

Pergunta

Como você definiria a categoria do Facebook? Quem são os objetos e morfismos? As propriedades são atendidas?

Categoria para Programadores

Categoria da Programação

Como podemos definir uma categoria apropriada para linguagens de programação? Quem são os objetos e morfismos?

Categoria dos Tipos

Categoria dos Tipos: os objetos são os tipos de uma linguagem de programação (primitivos ou compostos) e os morfismos são as funções que mapeiam um valor de um tipo para outro.

Tipo

Um **tipo** nada mais é que um conjunto de valores:

- `Int` compreende números inteiros representáveis com 32 bits
- `Char` caracteres da tabela ASCII ou Unicode
- `Primo` conjunto de números primos
- `Bool` contém apenas dois valores: `True, False`

Função

Representa um mapa do valor de um tipo para outro tipo (ou para o mesmo):

```
bool f (int);
```

Por ora vamos assumir apenas funções com apenas um único argumento como parte dos morfismos.

Haskell

Em Haskell a assinatura de uma função é similar a notação matemática:

```
f :: Bool -> Int
```

A função `f` mapeia um valor booleano para um inteiro.

É uma categoria?

Para os tipos e funções formarem uma categoria eles devem conter um morfismo identidade, um operador de composição e obedecer três propriedades: **identidade**, **associatividade** e **transitividade**.

Morfismo Identidade

A propriedade de identidade diz que existe um morfismo identidade (`id`) que pode receber qualquer tipo e retorna o mesmo valor de entrada para esse mesmo tipo.

Haskell

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

O tipo a é um tipo paramétrico, deve ser lido como *para qualquer a*.

C++

```
template <class A>
A id(A x) return x;
```

Python

```
def id(x):
    return x
```

Operador de Composição

O operador de composição $g \circ f$ pode ser lido como g *após* f e deve aplicar a função g na saída da função f.

Haskell

O Haskell já possui um operador de composição:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \x -> g (f x)
```

Dada uma função $g : b \to c$ e uma função $f : a \to b$, me entregue uma função $g \circ f : a \to c$.

C++

O C++11 introduziu o conceito de funções anônimas que nos permitem fazer:

```
1  auto const compose = [](auto g, auto f) {
2          return [f, g](auto x) {
3                  return g(f(x));
4          };
5  };
```

Python

Em Python fazemos:

```
def compose(g, f):
    return lambda x: g(f(x))
```

Identidade

```
1  (f . id) x          = f x
2  (\x -> f (id x)) x  = f x
3  (\x -> f x) x       = f x
4  f x                 = f x
```

Associatividade

```
1  (f . (g . h)) x          = ((f . g) . h) x
2  (f . (\x -> g (h x)) x = (\x -> (f . g)(h x)) x
3  (\y -> f ((\x -> g (h x) y)) x = (\x -> (\y -> f (g y))(h x)) x
4  f ((\x -> g (h x) x)  = (\y -> f (g y))(h x)
5  f (g (h x))              = f (g (h x))
```

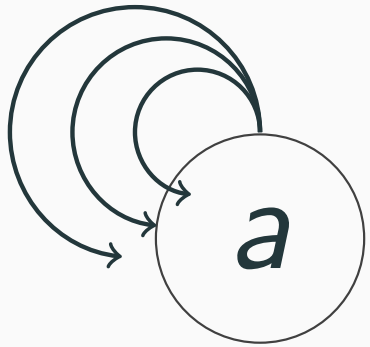
Transitividade

Consequência do nosso operador de composição.

Monoids

Monoids

O primeiro padrão que iremos aprender é uma categoria bastante simples, ela contém apenas um objeto!



Essa categoria é conhecida como **Monoid**.

Monoids

Em algebra um Monoid $M(C, \otimes, \epsilon_{\otimes})$ é composto por um objeto C, um operador binário $\otimes : m \to m \to m$, com $m \in C$ e um valor identidade $\epsilon_{\otimes}$ correspondente a esse operador.

Propriedades

$a, b, c \in C, (a \otimes b) \otimes c = a \otimes (b \otimes c) = a \otimes b \otimes c$

$a \otimes \epsilon = \epsilon \otimes a = a$

Monoids em Haskell

Em Haskell podemos criar uma classe de tipos:

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

Monoids em Haskell

Exemplo: multiplicação em números inteiros.

```haskell
instance Monoid Int where
    mempty  = 1
    mappend = (*)
```

Notação point-free

No Haskell as expressões `mappend x y = x*y` e `mappend = (*)` representam a mesma coisa!

Essa segunda expressão é chamada de *point-free* ou *igualdade extensional*.

Monoids em C++

Em C++20, podemos definir um Monoid como:

```
1  template<class T>
2    T mempty = delete;
3
4  template<class T>
5    T mappend(T, T) = delete;
6
7  template<class M>
8    concept bool Monoid = requires (M m) {
9      { mempty<M> } -> M;
10     { mappend(m, m); } -> M;
11   };
```

Monoids em C++

E o exemplo para inteiros com multiplicação (g++-8 com flags `-fconcepts -std=c++2a`):

```
1  template<>
2  int mempty<int> = {1};
3
4  int mappend(int x, int y) {
5      return x*y;
6  }
```

Monoids em Python

Em Python utilizamos `singledispatch`:

```
1  from functools import singledispatch
2
3  @singledispatch
4  def mempty(a):
5      raise Error("Not implemented for" + a)
6
7  @singledispatch
8  def mappend(a, b):
9      raise Error("Not implemented for" + a)
```

Monoids em Python

E o exemplo para inteiros com multiplicação:

```
1  @mempty.register(int)
2  def _(a):
3      return 1
4
5  @mappend.register(int)
6  def _(a,b):
7      return a * b
```

Sistema de Estoque

Vamos exemplificar com o seguinte trecho de código:

```haskell
type Qtd      = Int
type Preco    = Double
data Produto = P Qtd Preco

soma_produtos :: [Produto] -> Produto
soma_produtos ps = foldl somaProds prod0 ps
  where
    somaProds (P q1 p1) (P q2 p2) = P (q1+q2) (p1+p2)
    prod0 = P 0 0.0

zera_estoque :: Produto -> Produto
zera_estoque _ = P 0 0.0
```

Sistema de Estoque

Se eu quiser acrescentar um campo imposto? O que devo fazer?

```haskell
type Qtd      = Int
type Preco    = Double
type Imposto = Double
data Produto = P Qtd Preco Imposto

soma_produtos :: [Produto] -> Produto
soma_produtos ps = foldl somaProds prod0 ps
  where
    somaProds (P q1 p1 i1) (P q2 p2 i2) = P (q1+q2) (p1+p2)
                                            (max i1 i2)
    prod0 = P 0 0.0 0.0

zera_estoque :: Produto -> Produto
zera_estoque _ = P 0 0.0 0.0
```

Sistema de Estoque

Se eu tiver *n* funções que processam o tipo produto, tenho que fazer alteração em cada uma delas.

Devo tomar cuidado para que todas as alterações, repetitivas, sejam feitas corretamente!

Monoid de Estoque

```haskell
type Qtd     = Int
type Preco   = Double
data Produto = P Qtd Preco

instance Monoid Produto where
  mempty = Produto 0 0.0
  mappend (P q1 p1) (P q2 p2) = P (q1+q2) (p1+p2)

soma_produtos :: [Produto] -> Produto
soma_produtos ps = foldl mappend mempty ps

zera_estoque :: Produto -> Produto
zera_estoque _ = mempty
```

Monoid de Estoque

```haskell
type Qtd      = Int
type Preco    = Double
type Imposto = Double
data Produto = P Qtd Preco Imposto

instance Monoid Produto where
  mempty = Produto 0 0.0 0.0
  mappend (P q1 p1 i1) (P q2 p2 i2) = P (q1+q2) (p1+p2)
                                        (max i1 i2)

soma_produtos :: [Produto] -> Produto
soma_produtos ps = foldl mappend mempty ps

zera_estoque :: Produto -> Produto
zera_estoque _ = mempty
```

Monoid de Estoque

Agora eu só preciso alterar a instância de Monoid para o tipo Produto! Isso será feito uma única vez! Todas as outras funções continuam funcionando corretamente.

Categoria Kleisli

Traço de Execução

Imagine que temos diversas funções em C++, como por exemplo:

```
1  bool not(bool b) {
2      return !b;
3  }
4
5  bool is_even(int x) {
6      return x%2==0;
7  }
```

Traço de Execução

Precisamos criação um *log* do traço de execução de cada função no programa. Algo como:

```
1  int main () {
2     not(is_even(2));
3     not(is_even(3));
4  }
```

resultaria no *log* “even not even not”. Quais soluções podemos propor?

Traço de Execução

Alteramos todas as funções para retornarem `pair<T, string>` e concatenamos o log na função principal:

```
1  pair<bool, string> not(bool b) {
2      return make_pair(!b, "not ");
3  }
4
5  pair<bool, string> is_even(int x) {
6      return make_pair(x%2==0, "even ");
7  }
```

Traço de Execução

```
1  int main () {
2     pair<bool, string> p;
3     string log = "";
4
5     p = is_even(2);
6     log += p.second;
7     p = not(p.first);
8     log += p.second;
9
10    p = is_even(3);
11    log += p.second;
12    p = not(p.first);
13    log += p.second;
14 }
```

Traço de Execução

Código confuso, verboso, trabalhoso (aumentando a chance de *bugs*).

As funções não podem ser compostas como `not(is_even(3))`.

Traço de Execução

Para facilitar usamos uma variável global:

```
1  string log = "";
2
3  bool not(bool b) {
4      log += "not ";
5      return !b;
6  }
7
8  bool is_even(int x) {
9      log += "even ";
10     return x%2==0;
11 }
```

Traço de Execução

Nosso código precisa de poucas alterações e as funções continuam com a propriedade de composição:

```
1  int main () {
2      not(is_even(2));
3      not(is_even(3));
4  }
```

Traço de Execução

Efeito colateral

Cometemos um erro grave. Transformamos todas as nossas funções puras em funções com efeitos colaterais...

Funções Puras

Uma função f é pura se, para um mesmo valor de entrada, ela **sempre** retorna a mesma saída (inclusive suas saídas escondidas):

```
1 def f(x):
2   return 2*x
```

Função impura

Funções que retornam valores distintos ou causam um efeito em alguma outra forma de saída (arquivo, banco de dados, etc.):

```
1 def multRand(x):
2     return random()*x
```

Função impura

```
1  def escreve(s):
2      with open("arq.txt","a") as f:
3          f.write(s)
4      with open("arq.txt","r") as f:
5          lines = f.readlines()
6      return lines
```

Função Pura vs Impura

Para testar a pureza de uma função podemos tentar **memoizar**.
Se eu puder memoizar, então ela é pura, caso contrário não.

Pergunta

Essa função é pura ou impura?

```
1  int fatorial(int n) {
2    if (n <= 0) return 0;
3    if (n == 1) return 1;
4    return n * fatorial(n-1);
5  }
```

Pergunta

Essa função é pura ou impura?

```
1 std::getchar()
```

Pergunta

Essa função é pura ou impura?

```
1  bool f() {
2      std::cout << "Hello!" << std::endl;
3      return true;
4  }
```

Pergunta

Essa função é pura ou impura?

```
1  int f(int x) {
2      static int y = 0;
3      y += x;
4      return y;
5  }
```

Efeitos colaterais e bugs

Funções que causam efeitos colaterais devem ser evitadas ou muito bem documentadas e isoladas de todo o resto do programa pois

- Dependem de fontes externas que podem não ser confiáveis.
- Podem danificar essas fontes externas causando falhas em outras funções impuras.
- Mascarar alterações em uma estrutura de dados sendo utilizada (ex.: listas em Python)

Traço de Execução

Retomando nosso problema, não tem outro jeito exceto a primeira solução verbosa, feia, trabalhosa :(

```
1  pair<bool, string> is_odd(int x) {
2      pair<bool, string> p1, p2;
3      p1 = is_even(x);
4      p2 = not(p1.first);
5      return make_pair(p2.first, p1.second + p2.second);
6
7  }
```

Traço de Execução

Mas peraí, tem um padrão aí!

```
1  pair<bool, string> is_not_odd(int x) {
2      pair<bool, string> p1, p2;
3      p1 = is_odd(x);
4      p2 = not(p1.first);
5      return make_pair(p2.first, p1.second + p2.second);
6
7  }
```

Traço de Execução

Nosso objetivo é fazer com que as funções se conversem de forma natural. Vamos tentar criar uma categoria para esse tipo de função!

Traço de Execução

Vamos voltar a usar o Haskell por conta de sua notação mais próxima da matemática. Temos funções:

```
1 is_even :: Int  -> (Bool, String)
2 not     :: Bool -> (Bool, String)
```

Traço de Execução

E queremos criar um operador de composição para elas:

```
1  compose :: (a -> (b, String))
2          -> (b -> (c, String))
3          -> (a -> (c, String))
```

Traço de Execução

Para facilitar a notação, vamos criar o tipo `Writer` definido por:

```haskell
type Writer a = (a, String)
```

Categoria Writer

Para que o tipo `Writer` forme uma categoria, precisamos de uma função composição e uma identidade.

A função identidade deve ter como propriedade
$f \circ id = id \circ f = f$.

Composição: >=>

A composição é feita pelo operador conhecido como *peixe*:

```haskell
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \a ->
    let (b, s1) = m1 a
        (c, s2) = m2 b
    in (c, s1 ++ s2)
```

Categoria Writer

```haskell
notW :: Bool -> Writer Bool
notW b = (b, "not")

is_even :: Int -> Writer Bool
is_even x = (x `mod` 2 == 0, "even")

is_odd :: Int -> Writer Bool
is_odd = is_even >=> notW
```

Identidade Writer

Pensando na composição, como deve ser nossa função identidade?

```
1  (id >=> f) = \a ->
2    let (b, s1) = id a
3        (c, s2) = f b
4    in (c, s1 ++ s2)
```

Para que a função seja identidade temos que `b = x` e `s1 ++ s2 = s2`.

Identidade Writer

Vamos analisar se essa função atende nossa propriedade:

```haskell
return :: a -> Writer a
return x = (x, "")
```

(o nome `return` será explicado no final do curso)

Pergunta

Utilizamos a string vazia na função `return` por ela ser um elemento neutro da concatenação. Tenho um elemento neutro, um operador binário (concatenação) e o tipo `String`. O que isso forma?

Categoria Writer C++

```
1  template<class A>
2  Writer<A> identity(A x) {
3      return make_pair(x, "");
4  }
```

Categoria Writer C++

```
auto const compose = [](auto m1, auto m2) {
    return [m1, m2](auto a) {
        auto p1 = m1(a);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
};
```

Categoria Writer C++

```
1  Writer<bool> notW(bool b) {
2      return make_pair(!b, "not ");
3  }
4
5  Writer<bool> is_even(int x) {
6      return make_pair(x%2==0, "even ");
7  }
8
9  Writer<bool> is_odd(int x) {
10     return compose(is_even, notW)(x);
11 }
```

Categoria Kleisli

Essa categoria que acabamos de criar é generalizada pela categoria **Kleisli** que é um dos componentes da definição de **Monads** (mas ainda é cedo pra dizer que sabem o que é um Monad).

Categoria Kleisli

Generalizando, a categoria Kleisli possui os mesmos objetos da categoria dos tipos...

...porém os morfismos são da forma `a -> m b`.

Exercício

O que acontece se alterarmos a definição de `Writer` para carregar um inteiro no segundo componente?

```haskell
type Writer a = (a, Int)

(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 + s2)
```

Exercício

```haskell
fatorial :: Int -> Writer Int
fatorial 0 = (1,1)
fatorial 1 = (1,1)
fatorial n = (fatorial >=> (mul n)) (n-1)

mul :: Int -> Int -> Writer Int
mul n x = (n*x, 1)


main = do
    print $ fatorial 5
```

Writer para funções impuras

A nossa definição do tipo `Writer` nos permitiu transformar uma solução que gera efeitos colaterais em uma função pura!

Categoria Kleisli para funções parciais

Uma função parcial é aquela que não tem um valor definido para todos os possíveis argumentos. Exemplo: raíz quadrada para números reais.

Categoria Kleisli para funções parciais

Uma forma de tratar esse tipo de função é utilizando o tipo `Maybe` no Haskell:

```haskell
data Maybe a = Nothing | Just a
```

Categoria Kleisli para funções parciais

E optional no C++:

```
template<class A> class optional {
    bool _isValid;
    A    _value;
public:
    optional()    : _isValid(false) {}
    optional(A v) : _isValid(true), _value(v) {}
    bool isValid() const { return _isValid; }
    A value() const { return _value; }
};
```

Categoria Kleisli para funções parciais

Vamos supor a existência de duas funções `safe_root` e `safe_reciprocal`:

```
1  optional<double> safe_root(double x) {
2      if (x >= 0) return optional<double>{sqrt(x)};
3      else return optional<double>{};
4  }
5
6  optional<double> safe_reciprocal(double x) {
7      if (x != 0) return optional<double>{1.0/x};
8      else return optional<double>{};
9  }
```

Categoria Kleisli para funções parciais

Nosso operador *peixe* para compor duas funções que retornam `optional` devem seguir a lógica:

- Se o resultado da primeira função for **nada**, retorna nada. Senão, passa o resultado como argumento para a segunda função.
- Se o resultado da segunda função for **nada**, retorna nada. Senão retorna o resultado.

Categoria Kleisli para funções parciais

```
1  auto const fish = [](auto f, auto g) {
2      return [f, g](double x) {
3          auto z = f(x);
4          if (z) return g(z.value());
5          else return z;
6      };
7  };
```

Categoria Kleisli para funções parciais

Com isso podemos fazer a sequência:

```
auto sequencia = fish(safe_reciprocal, safe_root);
```

Tipos de Dados Algébricos

Tipos Fundamentais

Podemos analisar os tipos em uma linguagem de programação pela quantidade de elementos que eles representam e as funções que eles podem fazer parte como entrada ou saída.

Tipos Fundamentais

Dentre os tipos listados anteriomente, o menor deles foi o `Bool`, contendo apenas dois elementos.

Absurdo!

Podemos pensar em um tipo com 0 elementos?

Void

No Haskell temos o tipo `Void` definido por:

```haskell
data Void
```

Não existe equivalente em C++ ou Python.

Funções com Void

Que tipo de funções podemos criar com `Void`?

```haskell
absurd :: Void -> a
```

Essa função é equivalente a lei da lógica clássica *ex falso quodlibet* que diz que *qualquer coisa pode seguir de uma contradição*.

Unit

E um tipo com apenas um elemento?

```haskell
data () = ()
```

ele é chamado de **unit** e é implementado pelo tipo `void` no C/C++/Java e `None` no Python.

Funções com Unit

Que funções podemos construir que retorna um unit?

```haskell
unit :: a -> ()
unit x = ()
```

Funções com Unit

E funções que recebem um unit?

```haskell
x :: () -> Int
x () = 10
```

que, nesse exemplo, escolhe um valor do tipo `Int`.

Exercício: Funções com Bool

Que tipos de funções construímos com o tipo `Bool`?

Exercício: Funções com Bool

Que tipos de funções construímos com o tipo `Bool`?

```
1 isAlpha :: Char -> Bool
2 isGreaterThanFive :: Int -> Bool
```

Predicados!

Exercício: Funções com Bool

Que tipos de funções construímos com o tipo `Bool`?

```
1 ifthenelse :: Bool -> Int
2 ifthenelse True  = 10
3 ifthenelse False = 20
```

Tipos Compostos

Digamos que eu tenha uma função que recebe um `Bool` e, dependendo do valor, deve retornar ou um inteiro, ou um caractere.

Também pense no tipo que representa produtos em nosso estoque, eles são compostos por `Int` e `Double`, como represento tais tipos?

Vamos definir os tipos **produtos** e **soma** (**coprodutos**)

Tipo Produto

Spoiler: o tipo produto é uma tupla.

Tipo Produto

Como definimos uma tupla utilizando Teoria das Categorias?

Construção Universal

Na Teoria das Categorias temos uma visão *de fora* dos objetos, ou seja, não temos acesso ao seu conteúdo. Isso permite criar estruturas abstratas.

Construção Universal

A Construção Universal é feito em dois passos:

- Criamos um padrão que compreende o que nos interessa (e outros que não).
- Criamos um rank para que o que nos interessa fique em primeiro lugar.

Isomorfismo

Isomorfismo: relação que indica que dois objetos são estruturalmente iguais.

Nem sempre conseguimos comparar dois objetos por igualdade.

Isomorfismo

Um objeto `a` é isomórfico ao objeto `b` se temos:

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

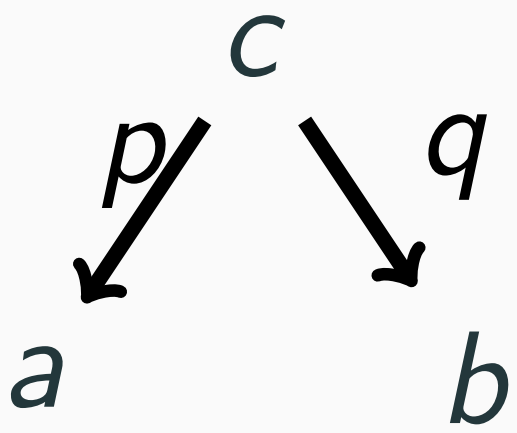
Tipo Produto

O tipo *c* é um tipo produto dos tipos *a* e *b*. O padrão que define *c* é a de duas funções de projeção:

```
1 p :: c -> a
2 q :: c -> b
```

Tipo Produto

Basicamente procuramos pelo seguinte padrão:



Tipo Produto

Pensando no par `Int` e `Bool`, podemos definir `c :: Int` fazendo:

```haskell
p :: Int -> Int
p x = x

q :: Int -> Bool
q _ = True
```

Tipo Produto

Outro candidato é fazer com que `c :: (Int, Int, Bool)`:

```haskell
p :: (Int, Int, Bool) -> Int
p (x, _, _) = x

q :: (Int, Int, Bool) -> Bool
q (_, _, b) = b
```

Tipo Produto

Para rankear nossos candidatos e encontrar o **melhor** tipo produto, dizemos que, dado c, c' e os morfismos p, q, p', q', c é melhor que c' se existe apenas um único m tal que:

```
1  m :: c' -> c
2
3  p' = p . m
4  q' = q . m
```

Tipo Produto

Pensando na opção $c = (Int, Bool)$ e nas alternativas anteriores, podemos fazer:

```
1  m1 x = (x, True)
2
3  m2 (x, _, b) = (x, b)
```

Tipo Produto

Vamos tentar fazer o oposto agora, encontrar um `m2'` que converta `c` na segunda opção de `c'`:

```
1  m2' (x, b) = (x, 1, b)
```

mas também podemos fazer:

```
1  m2' (x, b) = (x, 2, b)
```

Tipo Produto

Conclusão: o melhor tipo para representar o nosso tipo produto é uma tupla!

Exercício

O tipo produto `((Int, Bool), Int)` é isomórfico com o tipo `(Int, (Bool, Int))`? Defina duas funções que converta uma em outra e sejam inversas.

Exercício

Fazendo `a = Int` como a quantidade de valores contidos no tipo `Int` e, analogamente, `b = Bool` como a quantidade de valores no tipo `Bool`. Quantos valores possui o tipo `(Int, Bool)`?

Tipo Algébrico Produto

Como temos um tipo chamado *produto*, será que ele possui as mesmas propriedades algébricas de um produto? Já vimos que ele é associativo...

Tipo Algébrico Produto

Na álgebra o produto possui um elemento neutro, que tem valor 1. Em tipos temos o unit:

```haskell
f :: (a, ()) -> a
f (x, ()) = x

g :: a -> (a, ())
g x = (x, ())

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

Ou seja, o tipo `(a, ())` (e analogamente `((), a)`) são isomórficos a `a`, pois carregam a mesma informação.

Tipo Produto no Haskell

No Haskell, podemos representar pares utilizando nomes específicos para diferenciar um par de outro:

```haskell
data Circunferencia      = Circ Double Double Double
data TrianguloRetangulo = Tri Double Double Double
```

Tipo Produto no Haskell

Nesse código `Circunferencia` é o nome do tipo criado e `Circ` é o nome do construtor desse tipo:

```
1 Circ :: Double -> Double -> Double -> Circunferencia
```

Tipo Produto no Haskell

Embora os tipos `Circunferencia` e `TrianguloRetangulo` sejam isomórficos, no Haskell um não pode ser utilizado no lugar do outro (como poderiam em Python e C++):

```
1 area :: Circunferencia -> Double
2 area (Circ xc yc r) = pi*r*r
3
4 tri :: TrianguloRetangulo
5 tri = Tri 10.0 10.0 5.0
6
7 z = area tri
```

Type-driven Development

Esse código apresentará um erro de compilação, pois o Haskell exige que os tipos da assinatura de função e seus argumentos sejam os mesmos.

Type-driven Development

Lema do Haskell: *"if a Haskell program compiles, it probably works"* pois o sistema de tipos da linguagem impede que você passe argumentos para funções que sejam isomórficos porém representam diferentes contextos.

Type-driven Development

Essa ideia é atualmente estudada com o nome de **Type-driven Development**, em que o objetivo é dificultar a compilação de um programa de tal forma que, quando ele compila corretamente, minimiza as chances de *bugs*.

Tipo Produto no Haskell

Também podemos representar um produto como um registro:

```haskell
data Circunferencia = Circ { x_center :: Double
                           , y_center :: Double
                           , radius   :: Double }

area :: Circunferencia -> Double
area c = pi*r*r where r = radius c
```

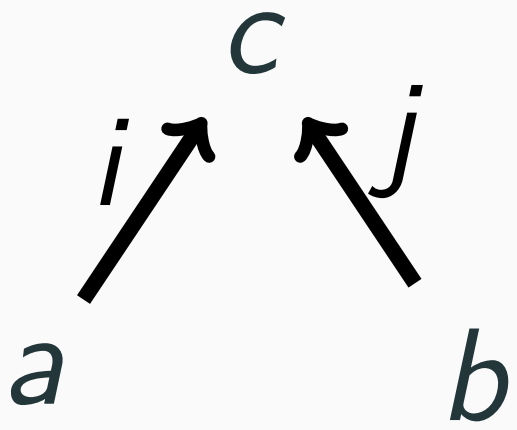
Tipo Coproduto

Vamos pensar agora no padrão dual ao produto, chamado **coproduto**:

```
1 i :: a -> c
2 j :: b -> c
```

Tipo Coproduto

Agora queremos o padrão dual ao produto!



Tipo Coproduto

A fatoração para rankeamento do melhor tipo para definir o coproduto é:

```
1  m :: c -> c'
2
3  i' = m . i
4  j' = m . j
```

Tipo Coproduto

O **tipo coproduto** ou **tipo soma** de dois objetos a e b é o objeto c equipado com duas funções injetivas para a e b.

Tipo Coproduto

É uma união disjunta de dois conjuntos. Em Haskell podemos definir um tipo soma como:

```haskell
data C = A Int | B Bool
```

Lemos essa definição como "O tipo `C` é composto de **ou** um componente `A` do tipo `Int` **ou** um componente `B` do tipo `Bool`.

Tipo Coproduto

Uma forma mais genérica é obtida pelo tipo paramétrico `Either`:

```haskell
data Either a b = Left a | Right b
```

Tipo Coproduto

Em C++ implementamos o tipo soma como um *tagged union*:

```
template<class A, class B>
struct Either {
    enum { isLeft, isRight } tag;
    union { A left; B right; };
};
```

Tipo Coproduto

Um outro exemplo de tipo soma no Haskell é o tipo `Bool` definido como:

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

que é isomórfico a:

```haskell
data Bool = Either () ()
```

Tipo Coproduto / Maybe

Um exemplo de tipo soma que vimos recentemente é o `Maybe`:

```haskell
data Maybe a = Nothing | Just a -- = Either () a
```

Álgebra dos Tipos

Será que podemos representar outras regras de soma e produto com tipos?

```haskell
-- a * 0 = 0
data Absurdo a = Ab a Void = Void

-- a * 1 = a
data Unity a = U a () = a

-- a + 0 = a
data Soma0 a = a | Void = a
```

Álgebra dos Tipos

Esses tipos também possuem propriedades distributivas entre eles?

```haskell
type Alpha a b c = (a, (Either b c))

type Beta a b c = Either (a, b) (a, c)

f :: Alpha -> Beta
f (x, Left y)  = Left (x, y)
f (x, Right y) = Right (x, y)

g :: Beta -> Alpha
g Left (x, y)  = (x, Left y)
g Right (x, y) = (x, Right y)
```

Semi-Anel

Tanto o tipo Soma como o tipo Produto formam um Monoid e essa combinação forma um **semi-anel**:

```
instance Monoid Soma where
    mempty  = Void
    mappend = Either

instance Monoid Prod where
    mempty  = ()
    mappend = (,)
```

Equivalência Álgebra-Tipos

Temos a seguinte tabela de equivalência entre a algebra e os tipos soma e produto:

Algebra	Tipos
0	`Void`
1	`()`
$a + b$	Either a b
$a * b$	(a, b)
$2 = 1 + 1$	`Bool = True | False`
$1 + a$	`Maybe a = Nothing | Just a`

Equivalência Lógica-Tipos

Da mesma forma que os tipos em Haskell são isomórficos ao semi-anel da álgebra, eles também são isomórficos com a lógica clássica:

Lógica	Tipos
Falso	`Void`
Verdadeiro	`()`
$a \vee b$	Either a b
$a \wedge b$	(a, b)

Esse isomorfismo é conhecido como *Isomorfismo de Curry-Howard* e pode ser estendido para categorias.

Tipos Recursivos

Uma outra forma interessante de construção de tipos é através da recursão. Por exemplo, considere o tipo que representa uma lista em Haskell:

```haskell
data List a = Vazio | (:) a (List a)
```

Tipos Recursivos

Essencialmente isso nos permite definir uma lista como:

```
1 xs = List Int
2 xs = 1 : 2 : 3 : Vazio
```

que representa a lista `[1,2,3]`.

Lista

Quantos valores possíveis temos para um tipo `List a`?

Podemos resolver isso algebricamente também definindo `x = List a`:

$$x = 1 + a \cdot x$$

$$x = 1 + a \cdot (1 + a \cdot x)$$

$$x = 1 + a + a^2 \cdot x$$

$$x = 1 + a + a^2 + a^3 \cdot x$$

$$x = 1 + a + a^2 + a^3 + a^4 + \ldots$$

Lista

Esse resultado pode ser interpretado como:

Uma lista do tipo a pode conter um único valor (lista vazia) **ou** um valor do tipo a **ou** dois valores do tipo a, etc.

Tipos Buracos

Mais e mais álgebra

O que mais podemos fazer com os tipos algébricos? Uma outra operação possível é o cálculo da derivada de nossos tipos!

Derivada de Tipos

Vamos estabelecer algumas derivadas básicas em função de um tipo a:

```
()'     = Void
a'      = ()
(a+b)' = () + Void
(a*b)' = 1*b = b
(a*a)' = 2a = a + a
```

Tipos Buracos

Os tipos resultantes da derivada são chamados de **buracos** pois elas criam um ponto de foco na nossa estrutura.

Buraco de uma Lista

A representação algébrica da lista é:

$x = 1 + a \cdot x$

Isolando a variável x temos:

$x = 1/(1 - a)$

Derivando esse tipo, temos:

$x' = 1/(1 - a)^2$

Buraco de uma Lista

O buraco de uma lista é o produto de duas listas!

```haskell
data Zipper a = Zip [a] [a]

criaZip :: [a] -> Zipper a
criaZip xs = Zip [] xs

esq :: Zipper a -> Zipper a
esq (Zip []     ds) = Zip [] ds
esq (Zip (e:es) ds) = Zip es (e:ds)

dir :: Zipper a -> Zipper a
dir (Zip es [])     = Zip es []
dir (Zip es (d:ds)) = Zip (d:es) ds
```

Buraco de uma Lista

Com isso definimos uma lista duplamente ligada:

```haskell
xs :: [Int]
xs = [1,2,3,4,5]

zs :: Zipper Int
zs = criaZip xs
-- zs = Zip [] [1,2,3,4,5]

zs' :: Zipper Int
zs' = (dir . dir) zs
-- zs' = Zip [2,1] [3,4,5]

zs'' :: Zipper Int
zs'' = esq zs'
-- zs'' = Zip [1] [2,3,4,5]
```

Buraco de uma árvore

Outro `Zipper` interessante é o de uma árvore com elemento apenas nos nós internos:

```haskell
data Tree a = Empty | Node (Tree a) a (Tree a)
```

Buraco de uma árvore

Fazendo `Tree a = 1 + a * (Tree a) * (Tree a)` e substituindo `Tree a` por x, temos:

$x = 1 + a * x^2$

Derivando em função de a temos:

$x' = x^2 + 2 * a * x * x'$

Que pode ser resolvido como:

$x' = (x^2)/(1 - 2ax) = x^2 \cdot 1/(1 - (ax + ax))$

Buraco de uma árvore

Transformando em um tipo algébrico, isso representa o produto entre uma tupla de árvores e uma lista com o foco atual, a árvore acima e o caminho utilizado, ou

```
(Tree a, Tree a, [Either (a, Tree a)])
```

Buraco de uma árvore

```haskell
data Zipper a = Zip { left  :: Tree a
                    , right :: Tree a
                    , focus :: [Either (a, Tree a) (a, Tree a)]
                    }
```

Buraco de uma árvore

Então o foco em um elemento x nessa árvore binária é composta por:

- ramo da esquerda
- ramo da direita
- lista de subárvores acima, sendo o primeiro elemento o nó-foco

Cada subárvore é sinalizada com `Left` ou `Right` indicando o caminho tomado para chegar até ele.

Buraco de uma árvore

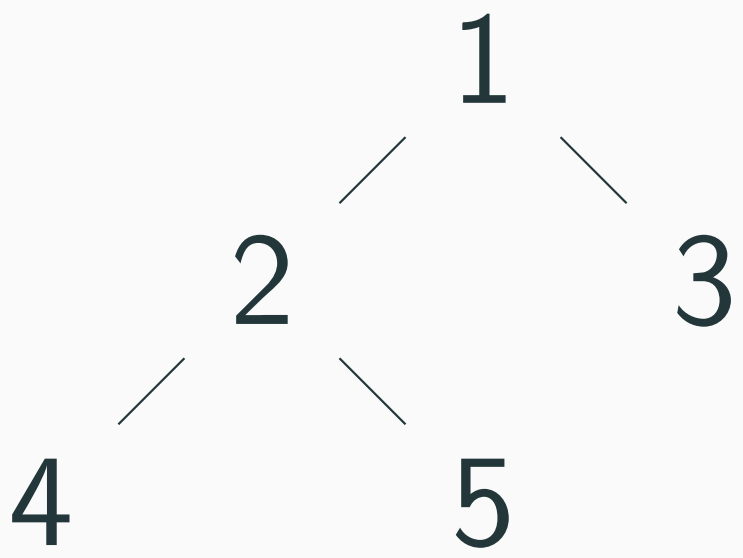
```haskell
criaZip :: Tree a -> Zipper a
criaZip Empty = Zip Empty Empty []
-- Foco inicial não tem ninguém acima dele (Empty)
criaZip (Node l x r) = Zip l r [Left (x, Empty)]

esq :: Zipper a -> Zipper a
esq tz =
  case left tz of
    Empty      -> tz  -- se não temos nós a esquerda
    -- O novo foco é o nó raiz da árvore esquerda
    -- acima dele é a árvore direita
    Node l x r -> Zip l r (Left (x, (right tz)) : focus tz)

dir :: Zipper a -> Zipper a
dir tz =
  case right tz of
    Empty      -> tz
    Node l x r -> Zip l r (Right (x, (left tz)) : focus tz)
```

Buraco de uma árvore

```haskell
upwards :: Zipper a -> Zipper a
upwards (Zip l r [])     = Zip l r []
upwards (Zip l r [x])    = Zip l r [x]
upwards (Zip l r (f:fs)) = t
  where t = case f of
              Left  (x, t') -> Zip (Node l x r) t' fs
              Right (x, t') -> Zip t' (Node l x r) fs
```

Buraco de uma árvore

Com esse tipo de árvore podemos fazer um algoritmo de *backtracking* de forma eficiente:



Buraco de uma árvore

```haskell
t :: Tree Int
t = Node (Node (Node Empty 4 Empty) 2 (Node Empty 5 Empty))
         1
         (Node Empty 3 Empty)

tz :: Zipper Int
tz = criaZip t
-- Zip {left = Node (Node Empty 4 Empty) 2 (Node Empty 5 Empty)
--     , right = Node Empty 3 Empty
--     , focus = [Left (1,Empty)]}

esq tz
-- Zip {left = Node Empty 4 Empty
--     , right = Node Empty 5 Empty
--     , focus = [Left (2,Node Empty 3 Empty),
                  Left (1,Empty)]
       }
```

Buraco de uma árvore

```haskell
(esq . esq) tz
-- Zip {left = Empty
--     , right = Empty
--     , focus = [Left (4,Node Empty 5 Empty),
                  Left (2,Node Empty 3 Empty),
                  Left (1,Empty)]
       }

(dir . esq . esq) tz
-- Zip {left = Empty
--     , right = Empty
--     , focus = [Left (4,Node Empty 5 Empty),
                  Left (2,Node Empty 3 Empty),
                  Left (1,Empty)]
       }


(upwards . esq) tz = tz
```

Atividades para Casa

1. Escreva a definição de Monoid em sua linguagem favorita. Crie um outro exemplo de aplicação.
2. Escreva o operador de composição da categoria Writer na sua linguagem favorita.
3. Implemente o tipo `Either` na sua segunda linguagem favorita (a primeira sendo o Haskell). Teste com aplicações como Pedra-Papel-Tesoura, o tipo Maybe, etc.
4. Mostre que os tipos $a + a = 2 * a$ são isomórficos.

Atividades para Casa

5. Dado o tipo soma definido em Haskell:

```haskell
data Shape = Circle Float
           | Rect Float Float
```

podemos definir uma função genérica `area`:

```haskell
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rect d h) = d * h
```

Implemente a estrutura Shape como um `interface` na sua linguagem OOP favorita!

Atividades para Casa

6. Podemos acrescentar uma função para calcular circunferência das formas no Haskell:

```haskell
circ :: Shape -> Float
circ (Circle r) = 2.0 * pi * r
circ (Rect d h) = 2.0 * (d + h)
```

Acrescente essa função na `interface` criada no exercício anterior. Marque as linhas de código que você teve que alterar.

Atividades para Casa

7. Adicione a forma `Square` tanto no tipo `Shape` do Haskell como na `interface` implementada por você. O que teve que ser feito em Haskell e na sua linguagem favorita?