

Teoria das Categorias para Programadores

Fabrício Olivetti de França

03 de Agosto de 2019



Topics

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

Sobre o Curso

Sobre o Curso

Baseado no curso de Bartosz Milewski disponível em <https://bartozmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/>

Sobre o Curso

O material do curso está disponível em:

<http://pesquisa.ufabc.edu.br/haskell/categorias.html>

e

<http://pesquisa.ufabc.edu.br/haskell/cursos/19.q2.categorias/>

Os códigos estão disponíveis no repositório

<https://github.com/folivetti/Category4Programmers>

Para obter o certificado de participação, o aluno deve fazer pelo menos 70% das atividades propostas submetidas no Github.

Teoria das Categorias

Teoria das Categorias

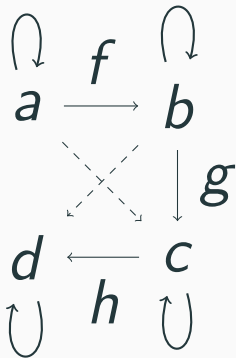
Teoria das Categorias é uma área da Matemática que formaliza, descreve e estuda estruturas abstratas com foco nas relações entre seus objetos.

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 $f : a \rightarrow b, g : b \rightarrow c$, então existe um $h = g \circ f : a \rightarrow c$.

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

Quais vantagens vocês percebem nessa última versão?

- 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.

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.

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.

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.

```
id :: a -> a
```

```
id x = x
```

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


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

```
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 .

O Haskell já possui um operador de composição:

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

$g . f = \backslash x \rightarrow g (f x)$

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

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 };
```

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

$$\begin{aligned} 1 & (f \cdot (g \cdot h)) x && = ((f \cdot g) \cdot h) x \\ 2 & (f \cdot (\lambda x \rightarrow g (h x))) x && = (\lambda x \rightarrow (f \cdot g)(h x)) x \\ 3 & (\lambda y \rightarrow f ((\lambda x \rightarrow g (h x)) y)) x && = (\lambda x \rightarrow (\lambda y \rightarrow f (g y))(h x)) x \\ 4 & f ((\lambda x \rightarrow g (h x)) x) && = (\lambda y \rightarrow f (g y))(h x) \\ 5 & f (g (h x)) && = f (g (h x)) \end{aligned}$$

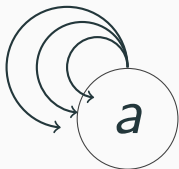
Transitividade

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 $\otimes : m \rightarrow m \rightarrow m$, com $m \in C$ e um valor identidade ϵ_{\otimes} correspondente a esse operador.

Propiedades

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

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

Monoids em Haskell

Em Haskell podemos criar uma classe de tipos:

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

Monoids em Haskell

Exemplo: multiplicação em números inteiros.

```
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 @empty.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:

```
1 type Qtd      = Int
2 type Preço   = Double
3 data Produto = P Qtd Preço
4
5 soma_produtos :: [Produto] -> Produto
6 soma_produtos ps = foldl somaProds prod0 ps
7   where
8     somaProds (P q1 p1) (P q2 p2) = P (q1+q2) (p1+p2)
9     prod0 = P 0 0.0
10
11 zera_estoque :: Produto -> Produto
12 zera_estoque _ = P 0 0.0
```

Sistema de Estoque

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

```
1 type Qtd      = Int
2 type Preço   = Double
3 type Imposto = Double
4 data Produto = P Qtd Preço Imposto
5
6 soma_produtos :: [Produto] -> Produto
7 soma_produtos ps = foldl somaProds prod0 ps
8   where
9     somaProds (P q1 p1 i1) (P q2 p2 i2) = P (q1+q2) (p1+p2)
10                                             (max i1 i2)
11     prod0 = P 0 0.0 0.0
12
13 zera_estoque :: Produto -> Produto
14 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

```
1 type Qtd      = Int
2 type Preço   = Double
3 data Produto = P Qtd Preço
4
5 instance Monoid Produto where
6     mempty = Produto 0 0.0
7     mappend (P q1 p1) (P q2 p2) = P (q1+q2) (p1+p2)
8
9 soma_produtos :: [Produto] -> Produto
10 soma_produtos ps = foldl mappend mempty ps
11
12 zera_estoque :: Produto -> Produto
13 zera_estoque _ = mempty
```

Monoid de Estoque

```
1 type Qtd      = Int
2 type Preço   = Double
3 type Imposto = Double
4 data Produto = P Qtd Preço Imposto
5
6 instance Monoid Produto where
7     mempty = Produto 0 0.0 0.0
8     mappend (P q1 p1 i1) (P q2 p2 i2) = P (q1+q2) (p1+p2)
9                                           (max i1 i2)
10
11 soma_produtos :: [Produto] -> Produto
12 soma_produtos ps = foldl mappend mempty ps
13
14 zera_estoque :: Produto -> Produto
15 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 criar 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 }
```

Essa função é pura ou impura?

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

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.
- Mascaram 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:

```
1 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: \gg

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

```
1 ( $\gg$ ) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
2 m1  $\gg$  m2 = \a ->
3     let (b, s1) = m1 a
4         (c, s2) = m2 b
5     in (c, s1 ++ s2)
```

Categoria Writer

```
1 notW :: Bool -> Writer Bool
2 notW b = (b, "not")
3
4 is_even :: Int -> Writer Bool
5 is_even x = (x `mod` 2 == 0, "even")
6
7 is_odd :: Int -> Writer Bool
8 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 $b = a$ e $s1 ++ s2 = s2$.

Vamos analisar se essa função atende nossa propriedade:

```
1 return :: a -> Writer a
2 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++

```
1 auto const compose = [] (auto m1, auto m2) {  
2     return [m1, m2] (auto a) {  
3         auto p1 = m1(a);  
4         auto p2 = m2(p1.first);  
5         return make_pair(p2.first, p1.second + p2.second);  
6     };  
7 };
```

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 $a \rightarrow m \ b$.

Exercício

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

```
1 type Writer a = (a, Int)
2
3 (>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
4 m1 >=> m2 = \x ->
5     let (y, s1) = m1 x
6         (z, s2) = m2 y
7     in (z, s1 + s2)
```

Exercício

```
1 fatorial :: Int -> Writer Int
2 fatorial 0 = (1,1)
3 fatorial 1 = (1,1)
4 fatorial n = (fatorial >=> (mul n)) (n-1)
5
6 mul :: Int -> Int -> Writer Int
7 mul n x = (n*x, 1)
8
9
10 main = do
11     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: raiz quadrada para números reais.

Categoria Kleisli para funções parciais

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

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

Categoria Kleisli para funções parciais

E optional no C++:

```
1 template<class A> class optional {  
2     bool _isValid;  
3     A    _value;  
4 public:  
5     optional()      : _isValid(false) {}  
6     optional(A v)  : _isValid(true), _value(v) {}  
7     bool isValid() const { return _isValid; }  
8     A value()     const { return _value; }  
9 };
```

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 Datos 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 anteriormente, 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:

```
1 data Void
```

Não existe equivalente em C++ ou Python.

Funções com Void

Que tipo de funções podemos criar com Void?

1 `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*.

E um tipo com apenas um elemento?

```
1 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?

```
1 unit :: a -> ()  
2 unit x = ()
```

Funções com Unit

E funções que recebem um unit?

```
1 x :: () -> Int
2 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 caracter.

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)**

Spoiler: o tipo produto é uma tupla.

Como definimos uma tupla utilizando Teoria das Categorias?

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

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.

Tipo Produto

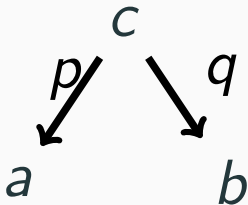
O tipo c é um tipo produto dos tipos a e b . O padrão que define c é a de duas funções de projeção:

1 $p :: c \rightarrow a$

2 $q :: c \rightarrow b$

Tipo Produto

Basicamente procuramos pelo seguinte padrão:



Tipo Produto

Para exemplificar, vamos imaginar que nosso objetivo é obter o produto entre um `Int` e um `Bool`.

Se definirmos `c :: Int` conseguimos implementar `p`, `q`?

Tipo Produto

Podemos definir `c :: Int` fazendo:

```
1 p :: Int -> Int
2 p x = x
3
4 q :: Int -> Bool
5 q _ = True
```

Tipo Produto

Se definirmos $c :: (\text{Int}, \text{Int}, \text{Bool})$ conseguimos implementar p, q ?

Tipo Produto

```
1 p :: (Int, Int, Bool) -> Int
2 p (x, _, _) = x
3
4 q :: (Int, Int, Bool) -> Bool
5 q (_, _, b) = b
```

Tipo Produto

Se definirmos $c :: (\text{Int}, \text{Bool})$ conseguimos implementar p, q ?

Tipo Produto

```
1 p :: (Int, Bool) -> Int
2 p (x, _) = x -- fst
3
4 q :: (Int, Bool) -> Bool
5 q (_, b) = b -- snd
```

Tipo Produto

Para ranquear 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 $m :: c' \rightarrow c$

2

3 $p' = p \cdot m$

4 $q' = q \cdot m$

Pensando na opção $c = (Int, Bool)$ e nas alternativas anteriores, vamos criar $m1$, $m2$ que satisfaça as propriedades.

Tipo Produto

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 $c :: (\text{Int}, \text{Bool})$ em $c' :: (\text{Int}, \text{Int}, \text{Bool})!$

1 $m_2'(x, b) = (x, 1, b)$

Mas também podemos fazer:

$$1 \quad m2' (x, b) = (x, 2, b)$$

Tipo Produto

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

Exercício

O tipo produto $((\text{Int}, \text{Bool}), \text{Int})$ é isomorfo com o tipo $(\text{Int}, (\text{Bool}, \text{Int}))$? Defina duas funções que converta uma em outra e sejam inversas.

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

Nem sempre conseguimos comparar dois objetos por igualdade.

Isomorfismo

Um objeto a é isomorfo ao objeto b se temos:

$$f \circ g = \text{id}$$

$$g \circ f = \text{id}$$

Exercício

Fazendo $a = \text{Int}$ como a quantidade de valores contidos no tipo `Int` e, analogamente, $b = \text{Bool}$ 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. Na categoria dos tipos, o `unit` representa esse valor.

Prove que $(a, ())$ é isomorfo a a .

Tipo Algébrico Produto

1 $f :: (a, ()) \rightarrow a$

2 $f (x, ()) = x$

3

4 $g :: a \rightarrow (a, ())$

5 $g x = (x, ())$

6

7 $f . g = id$

8 $g . f = id$

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

Uma palavra sobre o Haskell



Tipo Produto no Haskell

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

```
1 data Circunferencia      = Circ Double Double Double
2 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:

```
1 data Circunferencia = Circ { x_center :: Double
2                               , y_center :: Double
3                               , radius   :: Double }
4
5 area :: Circunferencia -> Double
6 area c = pi*r*r where r = radius c
```

Fim da propaganda!



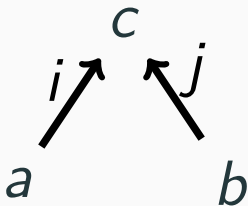
Tipo Coproduto

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

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

Nesse padrão estamos injetando valores do tipo a ou b em c. Temos uma **união** de conjuntos.

Tipo Coproduto



Tipo Coproduto

Basta invertermos as setas do tipo produto para obter a fatoração do melhor tipo coproduto:

1 $m :: c \rightarrow c'$

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:

```
1 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:

```
1 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:

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

que é isomorfo a:

```
1 data Bool = Either () ()
```

Provem!

Tipo Coproduto / Maybe

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

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

Álgebra dos Tipos

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

```
1  --  $a * 0 = 0$ 
2  data Absurdo a = Ab a Void = Void
3
4  --  $a * 1 = a$ 
5  data Unity a = U a () = a
6
7  --  $a + 0 = a$ 
8  data Soma0 a = A a | Void = a
```

Esses tipos também possuem propriedades distributivas entre eles?

```
1 type Alpha a b c = (a, (Either b c))
2
3 type Beta a b c = Either (a, b) (a, c)
```

Álgebra dos Tipos

```
1 f :: Alpha -> Beta
2 f (x, Left y) = Left (x, y)
3 f (x, Right y) = Right (x, y)
4
5 g :: Beta -> Alpha
6 g Left (x, y) = (x, Left y)
7 g Right (x, y) = (x, Right y)
```

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

```
1 instance Monoid Soma where
2     mempty  = Void
3     mappend = Either
4
5 instance Monoid Prod where
6     mempty  = ()
7     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:

```
1 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].

Quantos valores possíveis temos para um tipo `List a`?
Podemos resolver isso algebricamente definindo $x = \text{List } a$.

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

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

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

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

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

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

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 :

$$() \prime = \text{Void}$$

$$a \prime = ()$$

$$(a+b) \prime = () + \text{Void}$$

$$(a*b) \prime = 1*b = b$$

$$(a*a) \prime = 2a = a + a$$

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

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

O que é isso?

Buraco de uma Lista

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

Buraco de uma Lista

O tipo buraco da lista é conhecido como *Zipper*. A ideia é que, dada uma lista encadeada, essa estrutura permite otimizar o percurso na estrutura.

Buraco de uma Lista

A ideia é que a lista da direita armazene como primeiro elemento o *foco* atual e todos os elementos que aparecem após esse foco.

A lista da esquerda armazena os elementos anteriores, porém em ordem inversa, para otimizar o percurso.

Buraco de uma Lista

```
[] [1,2,3,4,5] -- direita  
[1] [2,3,4,5] -- direita  
[2,1] [3,4,5] -- direita  
[3,2,1] [4,5] -- esquerda  
[2,1] [3,4,5]
```

Buraco de uma Lista

A estrutura Zipper é definida como:

```
1 data Zipper a = Zip [a] [a]
2
3 criaZip :: [a] -> Zipper a
4 criaZip xs = Zip [] xs
```

Buraco de uma Lista

Para andar para esquerda, precisamos recuperar o elemento da primeira lista e inserir no começo da segunda lista:

```
1 esq :: Zipper a -> Zipper a
2 esq (Zip [] ds) = Zip [] ds
3 esq (Zip (e:es) ds) = Zip es (e:ds)
```

Buraco de uma Lista

Para andar para direita, precisamos remover o elemento-foco e inserir no começo da primeira lista:

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

Buraco de uma Lista

Com isso definimos uma lista duplamente ligada:

```
1  xs :: [Int]
2  xs = [1,2,3,4,5]
3
4  zs :: Zipper Int
5  zs = criaZip xs
6  -- zs = Zip [] [1,2,3,4,5]
7
8  zs' :: Zipper Int
9  zs' = (dir . dir) zs
10 -- zs' = Zip [2,1] [3,4,5]
11
12 zs'' :: Zipper Int
13 zs'' = esq zs'
14 -- zs'' = Zip [1] [2,3,4,5]
```

Buraco de uma árvore

Outro Zipper interessante pode ser criado a partir de uma árvore com elemento apenas nos nós internos:

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

Buraco de uma árvore

Algebricamente $\text{Tree } a = 1 + a * (\text{Tree } a) * (\text{Tree } a)$
e substituindo $\text{Tree } a$ por x , temos:

$$x = 1 + a * x^2$$

Buraco de uma árvore

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

Em Haskell podemos escrever essa estrutura como:

```
1 data Zipper a = Zip { left  :: Tree a
2                       , right :: Tree a
3                       , focus :: [Either (a, Tree a) (a, Tree a)]
4                       }
```

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

A estrutura inicial tem foco na raiz, acima da raiz não temos nada!

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

Buraco de uma árvore

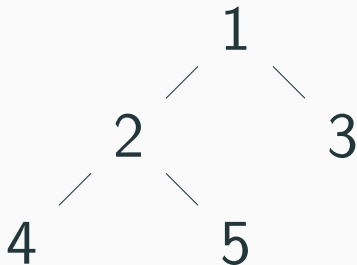
```
1  esq :: Zipper a -> Zipper a
2  esq tz =
3      case left tz of
4          Empty      -> tz  -- se não temos nós a esquerda
5          -- Caso contrario, o novo foco é o nó raiz da árvore esquerda
6          -- acima dele é a árvore direita
7          Node l x r -> Zip l r (Left (x, (right tz)) : focus tz)
8
9  dir :: Zipper a -> Zipper a
10 dir tz =
11     case right tz of
12         Empty      -> tz
13         Node l x r -> Zip l r (Right (x, (left tz)) : focus tz)
```

Buraco de uma árvore

```
1 upwards :: Zipper a -> Zipper a
2 upwards (Zip l r [])      = Zip l r []
3 upwards (Zip l r [x])    = Zip l r [x]
4 upwards (Zip l r (f:fs)) = t
5   where t = case f of
6             Left  (x, t') -> Zip (Node l x r) t' fs
7             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

```
1 t :: Tree Int
2 t = Node (Node (Node Empty 4 Empty) 2 (Node Empty 5 Empty))
3         1
4         (Node Empty 3 Empty)
5
6 tz :: Zipper Int
7 tz = criaZip t
8 -- Zip {left = Node (Node Empty 4 Empty) 2 (Node Empty 5 Empty)
9 --      , right = Node Empty 3 Empty
10 --     , focus = [Left (1,Empty)]}
11
12 esq tz
13 -- Zip {left = Node Empty 4 Empty
14 --      , right = Node Empty 5 Empty
15 --     , focus = [Left (2,Node Empty 3 Empty),
16 --                Left (1,Empty)]
17 }
```

Buraco de uma árvore

```
1 (esq . esq) tz
2 -- Zip {left = Empty
3 --      , right = Empty
4 --      , focus = [Left (4,Node Empty 5 Empty),
5 --                  Left (2,Node Empty 3 Empty),
6 --                  Left (1,Empty)]
7      }
8
9 (dir . esq . esq) tz
10 -- Zip {left = Empty
11 --      , right = Empty
12 --      , focus = [Left (4,Node Empty 5 Empty),
13 --                  Left (2,Node Empty 3 Empty),
14 --                  Left (1,Empty)]
15      }
16
17
18 (upwards . esq) tz = tz
```

Atividades para Casa

1. Que outros tipos formam um Monoid? Escreva uma ou mais instâncias em sua linguagem favorita.
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 implementando o tipo `Maybe`.
4. Mostre que os tipos $a + a = 2 * a$ são isomorfos.

Atividades para Casa

5. Dado o tipo soma definido em Haskell:

```
1 data Shape = Circle Float
2             | Rect Float Float
```

podemos definir uma função genérica area:

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

Implemente a estrutura Shape como um interface na sua linguagem OOP favorita! Compare a quantidade de linhas de código.

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

```
1 circ :: Shape -> Float
2 circ (Circle r) = 2.0 * pi * r
3 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.

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?

8. [opcional] Implemente os Zippers de lista e árvore binária em sua linguagem favorita (exceto Haskell).