Plano de Ensino

Inteligência na Web e Big Data

Fabricio Olivetti de França e Thiago Ferreira Covões
folivetti@ufabc.edu.br, thiago.covoes@ufabc.edu.br

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

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

Paradigma Funcional

Paradigmas

Em muitos cursos de Computação e Engenharia iniciam com paradigma imperativo.

Exemplo clássico da receita de bolo.

Paradigma Imperativo

```
1  int pares[10];
2  for (int i=0; i<10; i++) {
3      pares[i] = 2*i;
4  }
```

Paradigma Imperativo

- Descrevo passo a passo o que deve ser feito.
- Infame goto.
- Evoluiu para o procedural e estruturado com if, while, for.

Paradigma Imperativo

```
1  int pares[10];
2
3  for (int i=0; i<10; i++) {
4      pares[i] = dobro();
5  }
6
7  int dobro () {
8      static int i = 0;
9      ++i;
10     return i;
11 }
```

Problemas

Ao seguir o passo a passo, você chega no resultado…mas não tem ideia de qual será ele.

Qualquer um pode usar suas variáveis globais e suas funções, para o seu uso intencional ou não…

Orientação a Objetos

Outro paradigma muito estudado em cursos de Computação.

Encapsula dados com seus próprios métodos.

Evita que certas informações e procedimentos sejam utilizados fora de seu contexto.

```
1  class Pares {
2     private int i;   // não quero ninguém mudando meu
   ↪  estado
3     private ArrayList lista;
4     public Pares() {
5        i=0;
6        lista = new ArrayList();
7     }
8     public void addOne() {
9        ++i;
10       lista.add(2*i);
11    }
12    public ArrayList get(){
13       return lista;
14    }
15 }
```

```
1  class Main {
2     public static void main(String[] args) {
3          Pares l1 = new Pares();
4          l1.addOne();
5          l1.addOne();
6          l1.addOne();
7          System.out.println(l1.get());
8     }
9  }
```

esse código foi escrito propositalmente da pior forma

Problemas

Nos textos didáticos temos exemplos simples e criados para ilustrar quando faz sentido termos objetos.

Nem sempre isso representa a realidade.

Composição de funções através de herança = pode virar bagunça se alguns cuidados não forem tomados.

Problemas

Encapsula códigos imperativos para segurança e reuso, mas não evita os bugs na criação das classes.

Usa estado intensivamente, incentiva mutabilidade e não-determinismo.

Difícil de paralelizar.

Efeito colateral

A execução da instrução atual depende do estado atual do sistema.

Ex.: *dobro()* e *addOne()* vai depender do estado atual de i.

Efeito colateral

Eu posso executar a função *dobro()* para o primeiro e o segundo elemento em paralelo??

Paradigma Funcional

- Funções são atores principais.
- Computação = avaliação de composição de funções.
- Evita estados.
- Declarativo.

Paradigma Funcional

```haskell
take 10  [2*i | i <- zplus]

-- pegue 10 elementos da lista formada pelo
-- dobro dos números inteiros positivos.
```

Paradigma Funcional

Note que i não é uma variável, não muda de valor, é apenas um símbolo da definição.

Funções Puras

Linguagens funcionais incentivam (ou obrigam) a criação de funções puras.

Ao chamar a função com o mesmo argumento, sempre terá a mesma resposta.

Se não temos efeito colateral…

- …e o resultado de uma expressão pura não for utilizado, não precisa ser computado.
- …o programa como um todo pode ser reorganizado e otimizado.
- …é possível computar expressões em qualquer ordem (ou em paralelo).

Funções Puras

```c
double media (int * valores, int n) {
    double soma = 0;
    int i;
    for (i = 0; i < n; i++)
        soma_valor(&soma, valores[i]);
    return soma / n;
}

void soma_valor (double * soma, int valor) {
    soma += valor;
}
```

Programação sem bugs

A ausência de estados permite evitar muitos erros de implementação.

O lema da linguagem Haskell: ”se compilou, o código está correto!”

Funções e Funções

A construção de um programa nesse paradigma é através da composição de programas menores. O código anterior se traduz em:

```
1 (take 10 . map (2*)) zplus
```

Iterações vs Recursões

Em linguagens funcionais os laços iterativos são implementados via recursão, geralmente levando a um código enxuto e declarativo.

Iterações vs Recursões

```
1  int gcd (int m, int n) {
2    int r = m % n;
3    while(r != 0) {
4        m = n; n = r; r = m%n;
5    }
6    return m;
7  }
```

Iterações vs Recursões

```haskell
mdc 0 b = b
mdc a 0 = a
mdc a b = mdc b (a `rem` b)
```

Funções como atores principais

Composição e Abstração

Dois conceitos importantes do paradigma funcional são a composição e a abstração.

Composição e Abstração

<blockquote class=”twitter-tweet”data-lang=”pt»<p lang=”en”dir=”ltr»'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 <a href=”https://t.co/S6UruJbBjF»pic.twitter.com/S6UruJbBjF</p>— Computer Science (@CompSciFact) <a href=”https://twitter.com/CompSciFact/status/948965367107465218?ref_src=twsrc%5Etfw»4 de janeiro de 2018</blockquote> <script async src=”https://platform.twitter.com/widgets.js”charset=”utf-8»</script>

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
```

Exemplo

```
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;
```

Exemplo

```
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?

Vantagens

- 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

Composição de funções

A composição de funções permite minimizar o uso de variáveis auxiliares, tornar o código declarativo e pensar melhor no fluxo de processamento.

Composição de funções

Estamos acostumados a pensar nesse fluxo de processamento no bash do Linux:

```
cat texto | grep "Big Data" | more
```

Construindo Pipelines

A construção do fluxo de processamento em programação é feita seguindo os tipos de entrada e saída das funções construídas.

Construindo Pipelines

Matematicamente podemos pensar em funções do tipo $f : X \to Y$ que recebem um único argumento do tipo X e retornam um valor do tipo Y.

Construindo Pipelines

Exemplos:

```
1  int dobra( int );
2  string toStr(int);
3  bool isEven(int);
4  bool isOnlyAlpha(string);
```

Extraindo informação da assinatura

A assinatura de uma função permite tirar algumas informações dela. Por exemplo, toda função que retorna um *booleano* representa um predicado, uma consulta.

Construindo Pipelines

Para que duas funções sejam compostas o tipo de saída de uma função deve ser o mesmo da entrada da outra.

Pergunta

Seguindo as funções anteriores, quais composições você consegue montar?

```
1  int dobra( int );
2  string toStr(int);
3  bool isEven(int);
4  bool isOnlyAlpha(string);
```

Funções de múltiplos argumentos

Para todos os efeitos, funções de múltiplos argumentos são equivalentes a funções de um único argumento cujo tipo é uma tupla de valores:

```
1  int soma(int, int) = int soma( pair<int, int> );
```

Listas e Functores

Listas

Um tipo importante em programação funcional é a lista.

A lista é um container de elementos de certo tipo que podem ser transformados, condensados e filtrados.

Listas

A operação mais importante em uma lista é a transformação, geralmente feita através de uma função chamada ‘map‘:

```
1  # map : ((A -> B), List[A]) -> List[B]
2  def map(f, xs):
3    return (f(x) for x in xs)
```

Listas

Reparem que essa função recebe uma função do tipo A para o tipo B e uma lista de A como parâmetro retornando uma lista do tipo B.

Isso é chamado de função de alta ordem.

Listas

Uma outra interpretação que pode ser dada para a função `map` é de que ela recebe uma função de A para B e retorna outra função de $L[A]$ para $L[B]$. Isso é denominado de Functor:

```
1  def fmap(f):
2    def g(xs):
3      return (f(x) for x in xs)
4    return g
5
6  fmap(dobra)(xs) == map(dobra, xs)
```

Functor

Um Functor generaliza a função map para qualquer tipo de container! Exemplo:

```
1  def fmap(f):
2    def g(dic):
3      return {k:f(v) for k, v in dic.iter()}
4    return g
```

Leis do Functor

1. Dada uma função identidade `id`: `fmap(id) == id`
2. Dadas duas funções `f, g`: `fmap comp(f,g) == comp(fmap(f), fmap(g))`

sendo `comp` a composição de funções.

Leis do Functor

Note que a segunda lei nos permite otimizar operações feitas em nossos dados:

```
map(dobra, map(somaUm, xs)) == map(comp(somaUm, dobra), xs)
```

Na segunda forma percorremos a lista uma única vez.

Exercício

Dado o tipo descrito abaixo, como você escreveria a função `fmap`?

```
class Identity:
  def __init__(self, x):
    self.val = x

def fmap(f):
  def g(i):
    return ???
return g
```

Exercício

Dado o tipo descrito abaixo, como você escreveria a função `fmap`?

```
1  class Tree:
2    def __init__(self, x):
3      self.root = x
4      self.left = None
5      self.right = None
6
7    def insert(self, x):
8      if x < self.root:
9        self.left = Tree(x) if self.left is None
10                           else self.left.insert(x)
11     elif x > self.root:
12       self.right = Tree(x) if self.right is None
13                            else self.right.insert(x)
```

Avaliação Preguiçosa

Avaliação Preguiçosa

Algumas linguagens funcionais implementam o conceito de avaliação preguiçosa.

Quando uma expressão é gerada, ela gera uma promessa de execução.

Se e quando necessário, ela é avaliada.

Avaliação Preguiçosa

```
1  int main () {
2      int x = 2;
3      f(x*x, 4*x + 3);
4      return 0;
5  }
6
7  int f(int x, int y) {
8      return 2*x;
9  }
```

Avaliação Preguiçosa

```
1  int main () {
2      int x = 2;
3      f(2*2, 4*2 + 3);
4      return 0;
5  }
6
7  int f(int x, int y) {
8      return 2*x;
9  }
```

Avaliação Preguiçosa

```
1  int main () {
2      int x = 2;
3      f(4, 4*x + 3);
4      return 0;
5  }
6
7  int f(int x, int y) {
8      return 2*x;
9  }
```

Avaliação Preguiçosa

```
1  int main () {
2      int x = 2;
3      f(4, 11);
4      return 0;
5  }
6
7  int f(int x, int y) {
8      return 2*x;
9  }
```

Avaliação Preguiçosa

```
1  int main () {
2      int x = 2;
3      8;
4      return 0;
5  }
6
7  int f(int x, int y) {
8      return 2*x;
9  }
```

Avaliação Preguiçosa

```haskell
f x y = 2*x

main = do
  let z = 2
  print (f (z*z) (4*z + 3))
```

Avaliação Preguiçosa

```haskell
f x y = 2*x

main = do
  let z = 2
  print (2 * (z*z))
```

Avaliação Preguiçosa

```
1  f x y = 2*x
2
3  main = do
4    let z = 2
5    print (8)
```

A expressão $4 * z + 3$ nunca foi avaliada!

Avaliação Preguiçosa

Isso permite a criação de listas infinitas:

```
1 [2*i | i <-[1..]]
```

Avaliação Preguiçosa

No Python podemos fazer algo similar utilizando `yield`:

```
1 def gerador(x):
2   while x != 0:
3     yield x%2
4     x = x/2
```

A sequência dos valores sucessivos de `x` serão gerados sob demanda.

Comentários Finais

Funcional e Big Data

Os conceitos do paradigma funcional são importantes quando trabalhamos com *Big Data*:

- A pureza das funções permite concorrência de operações
- A imutabilidade dos dados evita a necessidade de lidar com alterações custosas
- Da mesma forma, garante consistência nos resultados
- A ideia de *Functors* garante paralelismo e a capacidade de otimização do *pipeline* de processamento
- A avaliação preguiçosa permite evitar processamentos desnecessários