Programação Funcional

Paradigmas de Programação – BCC/UFRPE
Lucas Albertins – lucas.albertins@deinfo.ufrpe.br

Agenda

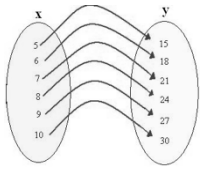
- Introdução
- Formas Funcionais
- Fundamentos de Linguagens Funcionais
- Avaliação
- Exemplo: Haskell
- Exercícios

Introdução

- Linguagens imperativas seguem arquitetura de von Neumann
  - Eficiência ao invés de adequabilidade ao desenvolvimento de software
- Linguagens funcionais são baseadas em funções matemáticas
  - Base teórica sólida mais próxima do usuário, mas sem tantas preocupações com arquitetura de máquinas nas quais os programas são executados

Funções Matemáticas

- Mapeamento de membros de um conjunto chamado domínio para outro conjunto chamado imagem



- cube(x) = x * x * x
- cube (2.0) = 2.0 * 2.0 * 2.0 = 8

Expressões Lambda

- Especifica os parâmetros e o mapeamento de uma função sem nome através da seguinte forma:

λ`(x) x * x * x`

para a função `cube (x) = x * x * x`

- São aplicadas aos parâmetros através da alocação dos parâmetros após a expressão

`(λ(x) x * x * x)(2)`

A qual retorna `8`

Formas Funcionais

- Uma função de alta ordem, ou forma funcional, é uma função que recebe funções como parâmetros ou retorna uma função como resultado, ou ambos
- Composição de Função: recebe duas funções como parâmetro e retorna uma função cujo valor é a primeira função aplicada a aplicação da segunda
  - Forma: `h = f ° g`, significa `h (x) = f(g(x))`

  Ex: `f(x) = x + 2` e `g(x)= 3 * x`,
  `h = f ° g` retorna `(3 * x)+ 2`

Formas Funcionais

- Aplique a tudo (*Apply-to-all*) é uma forma funcional que recebe uma única função como parâmetro e retorna uma lista de valores obtidos através da aplicação da função dada para cada elemento da lista de parâmetros
- Forma: α
- Ex: `h(x) = x * x`
- α`(h, (2, 3, 4))` retorna `(4, 9, 16)`

Fundamentos de Linguagens Funcionais

- Objetivo: imitar funções matemáticas da melhor forma possível
- Resolução de problemas é diferente do paradigma imperativo:
  - Não usa variáveis e comandos de atribuição
  - Não há comandos iterativos (iteração só através de recursão)
- Variáveis não são necessárias no paradigma funcional
  - Não há possibilidade de efeitos colaterais

Transparência Referencial

- No paradigma funcional, as expressões:
  - são a representação concreta da informação;
  - podem ser associadas a nomes (definições)
  - denotam valores que são determinados pelo interpretador da linguagem
- No âmbito de um dado contexto, todos os nomes que ocorrem numa expressão têm um valor único e imutável.
- Uma função sempre produz o mesmo resultado quando são dados os mesmos parâmetros
  - Semântica simplificada, como também testes mais simples

Fundamentos de Linguagens Funcionais

- Ao invés de comados, computação é feita através de avaliações de expressões
- Unidade Básica: Função
  - Abstrações sobre expressões
  - Valores de primeira classe
  - Função de alta ordem
- Polimorfismo paramétrico
  - Funções operam sobre famílias de valores
    - second :: (s, t) -> t
    - length :: [t] -> Int

Fundamentos de Linguagens Funcionais

- Abstração de Dados
  - Separação de Interesses (*Separation of Concerns*)
  - Operações de tipos abstratos são constantes e funções
- Avaliação
  - Estrita (*eager/strict*)
  - Normal/Não estrita
    - Preguiçosa (*Lazy Evaluation*)

Avaliação

- Estrita (*Eager*)
  - Argumentos são completamente avaliados durante a chamada independentemente de serem ou não utilizados na computação final.Argumentos só são avaliados quando utilizados e a cada vez que forem utilizados.
- Normal
  - O parâmetro real é avaliado apenas quando necessário
  - Substituímos cada ocorrência do parâmetro formal pelo real
- `sqr n = n * n` → `sqr(m+1)`
  - Estrita: m+1 é avaliado e vinculado ao parâmetro formal n, mesmo que ele não seja usado
  - Normal: o parâmetro formal `n` é associado a expressão m+1 e toda vez que ele é utilizado a expressão é avaliada
    - `Sqr n = (m+1) * (m+1)`

Avaliação

- Considere

```
cand b1 b2 = if b1 then b2 else False
```

- Com a chamada
  - `cand (n>0) (t/n>50)` com n=2 e t=80
    - Avaliações (estrita e normal) com os mesmos valores
    - Para n=0 e t=80
      - Estrita: falha (função depende da avaliação dos argumentos)
      - Normal: false (função executada avaliando apenas parte dos argumentos)

Avaliação

- Preguiçosa (*Lazy Evaluation*)
  - O argumento é avaliado apenas no seu primeiro uso e o resultado da avaliação é armazenado para posteriores usos

Comparação Imperativo vs Funcional

Imperativo	Funcional
Variáveis para representar memória	Expressões
Comandos	Funções
Mais eficiente	Menos eficiente (interpretadas)
Em geral, sintaxe e semântica mais complexa	Em geral, sintaxe e semântica mais simples

Comparação Imperativo vs Funcional

Imperativo (C)

```
int sum_cubes(int n){
    int sum = 0;
    for(int index = 1; index <= n; index++)
        sum += index * index * index;
return sum;
}
```

Funcional (Haskell)

```
sumCubes n = sum (map (^3) [1..n])
```

Paradigma Funcional

- Adequado para
  - Inteligência artificial
  - Sistemas para a área da matemática
  - Aplicações lógicas
- Principais linguagens
  - LISP: uma das primeiras linguagens funcionais
  - ML: declarações de tipo explícitas
  - Haskell: puramente funcional (sem variáveis, sem atribuição e sem efeitos colaterais)
  - Scala: Híbrida (Java + Funcional)
  - F#: Híbrida (C# + Funcional)

Haskell

- Valores e tipos
  - Tipos primitivos: Bool, Char, Tipos Numéricos
  - Tipos compostos:
    - Tuplas – (3,"maria",9.1)
    - tipos algébricos (uniões disjuntas) – data shape = Point | Circular r
    - Listas
    - funções

Haskell - Listas

- Notação: elementos entre colchetes
  - Ex: direcoes = [“norte”, “sul”, “leste”, “oeste”]
- Funções para listas:
  - Cabeça da lista - `head direcoes` -> `“norte”`
  - Cauda da lista - `tail direcoes` -> `[“sul”, “leste”, “oeste”]`
  - Tamanho da lista – `length direcoes -> 4`
  - `++` (concatenação) – `direcoes ++ [“nordeste”] -> [“norte”, “sul”, “leste”, “oeste”, “nordeste”]`
  - Serie aritmética .. - `[2, 4..10]` -> `[2, 4, 6, 8, 10]`
  - Construção - `1:[3, 5, 7]` -> `[1, 3, 5, 7]`

Exemplo de função com lista

```
soma:: [Int] -> Int

soma [] = 0

soma(n:ns) = n + soma ns
```

- Chamada a função possui a forma “E_1 E_2”. Se `f` é uma função que resulta em nova função, então `f x` resulta em uma função

Exemplos Lista

```
produto :: [Int] -> Int
produto [] = 1
produto (n : ns) = n * produto ns

through :: Int -> Int -> [Int]
m 'through' n =
      if m > n then []
      else m : (m+1 'through' n)

factorial n = produto (1 ‘through’ n)
```

Expressões LET

- Usadas para calcular valores intermediários

```
roots a b c =
        let
          det = sqrt (b*b - 4*a*c)
          twice_a = 2*a
        in
         ((-b + det) / twice_a, (-b - det) /
twice_a)
```

Compreensão de Listas

- Expressão iterativa sobre listas
  - `[ n + 1 | n <- ns]`
  - Forma geral: [E | Q_1, ..., Q_n], onde Qi é um gerador ou um filtro
- Matemática: $A = \{ x^2 \mid X \in N \}$
- Haskell
  - `listaQuad = [ x^2 | x <- [1..30] ]`
  - `listaQuadInf = [ x^2 | x <- [1..] ]`
    - `elem 4 listaQuadInf`

Compreensão de Listas

- Exemplo

```
dobraPos::[Int]->[Int]
dobraPos xs = [2*x | x<-xs, x>0]
```

Como seria a versão sem compreensão?

- Exemplo (Quicksort)

```
sort :: [Int] -> [Int]
sort [] = []
sort (x:xs) =
   sort [y | y <- xs, y<x]
   ++ [x]
   ++ sort [z | z<-xs, z >= x]
```

Outras funções sobre listas

Função	Descrição	Exemplo
last	Retorna o ´ ultimo elemento da lista	> last [4,3,2] 2
elem	Verifica se um elemento pertence a lista	> elem 5 [1,5,10] True
(!!)	Operador de índice da lista, retorna o elemento mantido numa posição	> [1,3,5,7,9] !!0 1 > (!!) ['b','g','r','w'] 3 'w'
reverse	Inverte os elementos de uma lista	> reverse [4,5,2,2] [2,2,5,4]
splitAt	Divide uma lista num par de sub-listas fazendo a divisão numa determinada posição	> splitAt 2 [3,4,2,1,5] ([3,4],[2,1,5])
maximum	Retorna o maior elemento de uma lista	> maximum [4,5,1,2] 5
minimum	Retorna o menor elemento de uma lista	> minimum [5.2,0.3,7.2] 0.3

Casamento de Padrão

- Padrão lembra uma expressão em que pode haver identificadores

```
data Shape = Pointy | Circular Float |
Rectangular(Float,Float)

area :: Shape -> Float

area Pointy = 0.0

area (Circular r) = pi * r * r

area (Rectangular(h,w)) = h * w
```

Currificação

```
power :: (Int, Float) -> Float

power(n, b) = if n == 0 then 1.0 else b * power(n-1, b)


powerc :: Int -> Float -> Float

powerc n b = if n == 0 then 1.0 else b * powerc (n-1) b


sqr = powerc 2

sube = powerc 3
```

Funções genéricas de alta ordem

- Funções genéricas sobre listas aplicam alguma regra geral sobre os elementos de uma lista
- Três tipos:
  - Mapeamento (*mapping*)
  - Filtragem (*filter*)
  - Redução (*folding*)

Mapeamento

- Uma função é aplicada a cada elemento de uma lista, de modo que uma nova lista modificada é retornada.
- Recebe:
  - uma função de transformação
  - uma lista de elementos a serem transformados
- Definição

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

Mapeamento

- Exemplos

```
Main> map (+7) [1,2,3]
[8,9,10]
Main> map (even) [1,2,3,4]
[False,True,False,True]
Main> map ("Sr. " ++) ["Joao","Pedro","Luiz"]
["Sr. Joao","Sr. Pedro","Sr. Luiz"]
Main> map (True &&) [True,False]
[True,False]
Main> map (False ||) [False, True]
[False,True]
```

Mapeamento

- Outro Exemplo

```
convertChar xs = [ ord x | x <- xs]
Main> convertChar "adriana"
[97,100,114,105,97,110,97]
Main> convertChar ['a','b','c']
[97,98,99]
```

- De forma equivalente, podemos usar o mapeamento através da função map:

```
Main> map ord "adriana"
[97,100,114,105,97,110,97]
Main> map ord ['a','b','c']
[97,98,99]
```

Filtragem

- As vezes é interessante produzir sub-listas através da seleção de elementos que compartilham uma determinada propriedade.
- Uma função filtro recebe a função que define a propriedade e uma lista de entrada, e retorna uma sub-lista contendo os elementos que satisfazem a propriedade.
- Definição

```
p :: a -> Bool
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
      | p x = x : filter p xs
      | otherwise = filter p xs
```

Filtragem

- Exemplo

```
pegaDigitos :: String -> String

pegaDigitos xs = filter isDigit xs


pegaLetras :: String -> String

pegaLetras xs = filter isAlpha xs
```

Redução (*Folding*)

- As vezes é interessante transformar todos os elementos de uma lista num único valor, dada uma propriedade de transformação.
- sum [1,2,3] = 1 + 2 + 3 = 6.
- Uma Redução implementa a operação de aplicar um operador ou função à uma lista de valores e combiná-los.
- Haskell tem quatro tipos de funções para redução:
  - esquerda-direita: foldl e foldl1
  - direita-esquerda: foldr e foldr1

Exemplos

```
Main> foldr1 (+) [1,2,3]
6
Main> foldr (+) 1 [1,2,3]
7

Main> foldr (++) "ana" ["ab","bc"]
"abbcana"
Main> foldl (++) "ana" ["ab","bc"]
"anaabbc"
```

Programação Funcional em linguagens imperativas

- Várias linguagens imperativas suportam conceitos de programação funcional
- Funções Anônimas (expressões lambda)
  - JavaScript: deixa o nome da função fora da sua definição
  - C#: `i => (i % 2) == 0` (retorna true ou false dependendo se o parâmetro é par ou ímpar
  - Python: `lambda a, b : 2 * a  - b`

Programação Funcional em linguagens imperativas

- Python suporta as funções de alta ordem map e filter (geralmente usadas junto com expressões lambda como parâmetros)

```
map(lambda x : x ** 3, [2, 4, 6, 8])
```

Returns `[8, 64, 216, 512]`

- Python suporta currificação

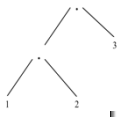
```
from operator import add
add5 = partial (add, 5)
```

Exercícios em sala de aula

- Defina a função sumsq n, que calcula a soma dos números quadrados de 1 a n.
- Defina uma função que dado um inteiro positivo informe se ele é um número primo ou não

Exercício extra-classe

- A seguinte estrutura de dados representa uma árvore binária contendo valores apenas nas folhas:
  - `data Tree a = Node (Tree a) (Tree a) | Leaf a`
- Considere a árvore t de inteiros apresentada a lado
- A representação de t como um objeto do tipo Tree Int tem Haskell seria:
  - `Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)`



Exercício extra-classe

- Implemente as seguintes funções em Haskell:

a) A função foldTree do tipo (a -> a -> a) -> (b -> a) -> Tree b -> a funciona da seguinte maneira:

- `foldTree n l t` substitui todas as ocorrências do construtor `Node` na árvore `t` por `n` e substitui todas as ocorrências do construtor `Leaf` em `t` por `l`. Assim, para a árvore `t` mostrada no slide anterior, `foldTree (+) id t` deveria gerar: `(+)((+)(id 1)(id 2))(id 3)`, que, finalmente, resulta em 6. Neste exemplo, `Node` foi substituído por `(+)` e `Leaf` foi substituído por `id`.

Exercício extra-classe

b) Use a função foldTree da letra (a) para implementer a função maxTree, que retorna o maior elemento da árvore.

Leitura Adicional

- Capítulo 15 – Linguagens de Programação Funcional. SEBESTA, R. W. Conceitos de Linguagens de Programação. 9ª ED. BOOKMAN, 2011.
- Próxima Aula:
  - Capítulo 16 – Linguagens de Programação Lógica. SEBESTA, R. W. Conceitos de Linguagens de Programação. 9ª ED. BOOKMAN, 2011.

Prolog