Linguagens de Programação Funcional

George Darmiton da Cunha Cavalcanti
(gdcc@cin.ufpe.br)

Universidade Federal
de Pernambuco

cin.ufpe.br

Introdução

- O projeto de linguagem imperativas é baseado na arquitetura de *von Neumann*
  - Eficiência é o objetivo principal, mesmo que isso seja visto, por alguns, como uma restrição desnecessária ao projeto de desenvolvimento de software

- O projeto de linguagens funcionais é baseado em funções matemáticas
  - Uma sólida base teórica

Programação Funcional

- Programação com alto nível de abstração
- Soluções elegantes, concisas e poderosas
- Funções
  - computam um resultado que depende apenas dos valores das entradas
- Forte fundamentação teórica, o que permite mais facilmente provas de propriedades sobre os programas

Programação Funcional: Objetivos

- Programação com um alto nível de abstração, possibilitando:
  - alta produtividade
  - programas mais concisos
  - programas mais fáceis de entender
  - menos erros
  - provas de propriedades sobre programas

Funções matemáticas

- Uma função matemática faz um mapeamento entre membros de um conjunto, chamado domínio, para um outro conjunto, chamado imagem

- Uma expressão *lambda* especifica os parâmetros e o mapeamento de uma função da seguinte forma

  `λ(x) x * x * x`

  para a função `cube (x) = x * x * x`

Expressões Lambda

- Expressões *lambda* descrevem funções sem nome

- Expressões *lambda* são aplicadas aos parâmetros colocando os parâmetros após a expressão

```
(λ(x) x * x * x)(2)
```

que é avaliado como sendo `8`

Formas Funcionais

- Uma função de ordem superior, ou forma funcional, é aquela que toma funções como parâmetros, que produz uma função como resultado, ou ambos

Composição de Funções

- Uma forma funcional que tem dois parâmetros funcionais e produz uma função cujo valor é a primeira função paramétrica real aplicada ao resultado da segunda

Forma: `h ≡ f ° g`

que significa `h (x) ≡ f ( g ( x))`

Para `f (x) ≡ x + 2` e `g (x) ≡ 3 * x`,

`h ≡ f ° g` gerando `(3 * x)+ 2`

Apply-to-all

- É uma forma funcional que toma uma única função como parâmetro e se aplica a uma lista de argumentos, gerando uma lista como saída

  Forma: α

  Para `h (x) ≡ x * x`

  α`( h, (2, 3, 4))` produz `(4, 9, 16)`

Fundamentos de Linguagens de Programação Funcionais

- O objetivo do projeto LPF é imitar funções matemáticas no maior grau possível
- O processo básico de computação é fundamentalmente diferente entre LPF e linguagens imperativas
  - Em uma linguagem imperativa, operações são realizadas e os resultados são armazenados em variáveis para posterior uso
  - Gerenciamento de variáveis é uma constante preocupação e fonte de complexidade para a programação imperativa
- Em uma LPF, variáveis não são necessárias, como é o caso na matemática

Transparência Referencial

- Em uma LPF, a avaliação de uma função sempre produz o mesmo resultado dado os mesmos parâmetros

Notação: programação baseada em definições


```
answer :: Int
answer = 42

greater :: Bool
greater = (answer > 71)

yes :: Bool
yes = True
```

Definição de funções


```
square :: Int -> Int
square x = x * x

allEqual :: Int -> Int -> Int -> Bool
allEqual n m p = (n == m) && (m == p)

maxi :: Int -> Int -> Int
maxi n m | n >= m    = n
         | otherwise = m
```

Avaliando Expressões

- Encontrar o valor de uma expressão
- Usar definições das funções

```
addD :: Int -> Int -> Int
addD a b = 2 * (a+b)

addD 2 (addD 3 4)
= 2 * (2 + (addD 3 4))
= 2 * (2 + 2 * (3 + 4))
= 32
```

Prova de propriedades

- Exemplo:

```
addD a b        = 2 * (a+b)
                = 2 * (b+a) = addD b a
```

- Válida para quaisquer argumentos a e b
- Não seria válida em linguagens imperativas, com variáveis globais.

Em uma linguagem imperativa


```
int b;
...
int f (int x) {
    b = x;
    return (5)
}

addD (f 3) b == addD b (f 3)?
```

Tipos Básicos

- Inteiros
  - `1,2,3,...:: Int`
  - `+, *, -, div, mod :: Int -> Int -> Int`
  - `>, >=, ==, /=, <=, < :: Int -> Int -> Bool`

- Booleanos
  - `True, False :: Bool`
  - `&&, || :: Bool -> Bool -> Bool`
  - `not :: Bool -> Bool`

Exemplo :: sales (vendas)

- suponha vendas semanais dadas pela função
  `sales :: Int -> Int`

- total de vendas da semana 0 à semana n?
  `totalSales :: Int -> Int`

- **`sales 0 + sales 1 + ... + sales (n-1) + sales n`**

- ```
  totalSales n
    | n == 0     = sales 0
    | otherwise = totalSales (n-1) + sales n
  ```
  ```

Recursão

- Definir caso base, i.e. valor para fun 0

- Definir o valor para fun n
  usando o valor de fun (n-1)
  Este é o caso recursivo.

```
maxSales :: Int -> Int
maxSales n
   | n == 0     = sales 0
   | otherwise = maxi (maxSales (n-1))
                      (sales n)
```

Casamento de Padrões


```
myNot :: Bool -> Bool
myNot True  = False
myNot False = True

myOr :: Bool -> Bool -> Bool
myOr True  x = True
myOr False x = x

myAnd :: Bool -> Bool -> Bool
myAnd False x = False
myAnd True  x = x
```

Caracteres e Strings


```
'a', 'b',...:: Char

'\t', '\n', '\\' :: Char

"abc", "andre" :: String

++ :: String -> String -> String
   "abc" ++ "def"
```

' ', "" e " " são diferentes!

Ponto Flutuante

- `Float`
- `Double`
- `22.3435 :: Float`
- `+,-,*,/ :: Float -> Float -> Float`
- `pi :: Float`
- `ceiling, floor, round :: Float -> Int`

Estruturas de dados :: Tuplas

- `intP :: (Int, Int)`
- `intP = (33,43)`

- `(True, 'x') :: (Bool, Char)`

- `(34, 22,'b') :: (Int, Int, Char)`

- `addPair :: (Int,Int) -> Int`
- `addPair (x,y) = x+y`

- `shift :: ((Int,Int),Int) -> (Int,(Int,Int))`
- `shift ((x,y),z) = (x,(y,z))`

Sinônimos de Tipos

- `type Name = String`
- `type Age = Int`
- `type Phone = Int`
- `type Person = (Name, Age, Phone)`

- `name :: Person -> Name`
- `name (n,a,p) = n`

Definições Locais

- Estilo bottom-up ou top-down

```
sumSquares :: Int -> Int -> Int

sumSquares x y = sqX + sqY
    where sqX = x * x
          sqY = y * y

sumSquares x y = sq x + sq y
    where sq z = z * z

sumSquares x y = let sqX = x * x
                     sqY = y * y
                 in sqX + sqY
```

Definições Locais


```
maxThreeOccurs :: Int -> Int -> Int -> (Int,Int)
maxThreeOccurs m n p = (mx, eqCount)
     where mx = maxiThree m n p
           eqCount = equalCount mx m n p
     ...
```

- Duas formas:
  - **`let`** *definições* **`in`** *expressão*
  - *definições* **`where`** *definições*

Notação

- Maiúsculas: tipos e construtores
- Minúsculas: funções e argumentos
- Case sensitive
- Comentários:

```
-- isto e' um comentario de uma linha
{- isto e´ um comentario de varias linhas... -}
```

Referências

- A Gentle Introduction to Haskell 98.
- Haskell – The Craft of Functional Programming. Simon Thompson.
- www.haskell.org

Leituras Recomendadas

- Why Functional Programming Matters. John Hughes.

- A Gentle Introduction to Haskell 98. Tutorial sobre Haskell.

Créditos

- Alguns destes slides foram baseados num conjunto de slides originalmente criados por André Santos (alms@cin.ufpe.br) e por Hermano Perrelli (hermano@cin.ufpe.br)

Aplicações de Linguagens Funcionais

- LISP é usada em inteligência artificial
  - Representação do conhecimento
  - Aprendizagem de Máquina
  - Processamento de Linguagem Natural
  - Modelagem da fala e da visão
- *Scheme* é usada para ensinar programação em um número considerável de universidades
- Programas com milhares de linhas de código:
  - compiladores, provadores de teoremas, entre outros
- Ericsson utiliza *Erlang*, uma linguagem funcional concorrente, para programação de *switches* de redes/telecomunicações, com excelentes resultados

Linguagens Funcionais *versus* Linguagens Imperativas

- Linguagem Imperativas
  - Execução eficiente
  - Semântica Complexa
  - Sintaxe complexa
  - Concorrência é formulada pelo programador
- Linguagens Funcionais
  - Semântica simples
  - Sintaxe simples
  - Execução ineficiente
  - Programas podem ser feitos concorrentes de maneira automática