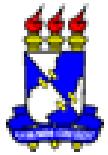
Abstração de Função

Prof. Alberto Costa Neto
alberto@ufs.br

Linguagens de Programação



Departamento de Computação
Universidade Federal de Sergipe

Conteúdo

- Abstração
- Conceitos importantes de LFs
  - Funções de Alta-Ordem
  - Composição de Funções
  - Currificação
  - Estratégias de Avaliação

Abstração

- Em computação, abstração alude a distinção entre
  - O quê?: O que faz uma peça de código
  - Como?: Como é implementada
- Abstração pode ser:
  - De processo
  - De dados
- Funções e Procedimentos são abstrações de processo
  - quem usa se preocupa com "o quê”
  - quem implementa, com o "como"

Parâmetros

- Uma abstração pode ser parametrizada
  - Evita-se a repetição do código para a mesma computação

- **Argumento:** é um valor que pode ser passado para uma abstração parametrizada
- **Parâmetro formal:** identificador usado na abstração para denotar um argumento
- **Parâmetro real:** uma expressão passada como argumento e é fornecida no momento da invocação da abstração (chamada da abstração)

Tipos de Abstração

- Uma **abstração** é uma entidade que incorpora alguma computação
  - **Abstração de função** é uma abstração sobre uma expressão
    - Uma chamada de função é uma expressão que retorna um valor
  - **Abstração de procedimento** é uma abstração sobre um comando
    - Uma chamada de procedimento é um comando que atualiza variáveis

Abstração de Função

- As duas visões de uma função:
  - para o usuário interessa o mapeamento
  - para o implementador, a avaliação do corpo (o algoritmo)

Haskell

```
power x n
  | n==1 = x
  | n/=1  = x * power x (n-1)
```

```
power x n
  | n==1 = x
  | n/=1  =
     if even n
     then power (sqr x)  (div n 2)
     else (power (sqr x)  (div n 2))*x
```

Abstração de Função

- Abstração de função em Pascal é mal feita
  - O corpo de uma função pode conter comandos que podem levar a efeitos colaterais
  - O identificador da função representa duas coisas diferentes dentro do mesmo escopo

Pascal

```
function potencia(x:real; n:Integer):Real;
begin
   if n = 1 then potencia := x
   else potencia := x * potencia(x, n-1)
end;
```

pseudo-variável

abstração de função propriamente dita que implica numa chamada recursiva

Abstração de Função

- Conclusão em Pascal:
  - Sintaticamente: corpo de função é um comando
  - Semanticamente: corpo de função é um tipo de expressão (já que retorna um valor)
- Mais natural e mais fácil de ser compreendido:
  - Corpo de função = simples expressão!
  - Ex:

ML

```
fun potencia(x:real; n:int) =
    if n = 1
    then x
    else x*potencia(x, n-1)
```

Abstração de Função

- Apesar de abstrações de funções serem construídas normalmente em definição de funções, Abstração e Vínculo podem ser conceitos distintos

```
fun quadrado (n: real) = n*n
≡
val    quadrado     =    fn (n:real) => n*n
```

ML

identificador

expressão

abstração de função é o **valor** vinculado

Abstração de Função

- Sendo assim....

ML

```
fun integral (a: real, b: real, f:  real->real) = ...

... integral (0.0, 1.0, quadrado) ...
... integral (0.0, 1.0, fn (x: real) => x*x) ...
```

Haskell

```
\x --> x*x
```

Composição de Funções

- Em Haskell, o operador "." compõe duas funções dadas, como definido abaixo:

```
(.) :: (t -> u) -> (s -> t) -> (s -> u)
f . g = \ x -> f(g(x))
```

f . g computa uma função h tal que h(x) = f(g(x))

- Dadas as funções da biblioteca:

```
not :: Bool -> Bool
odd :: Int -> Bool
```

- Podemos compô-las da seguinte forma;

```
even = not . odd   -- tipo é Int -> Bool
```

Funções de Alta-Ordem

- Exemplo

Haskell

```
quadrado :: Int -> Int
quadrado n = n*n

duasVezes :: (Int -> Int) -> (Int -> Int)
duasVezes f = f . f

-- função quartaPot?
quartaPot = duasVezes(quadrado)
```

quartaPot(2) *retorna 16*

Currificação (*Currying*)

- Também chamada de Aplicação Parcial
  - Técnica que permite aplicar um subconjunto dos parâmetros de uma função, gerando uma nova função
  - Esta função gerada como resultado requer o restante dos parâmetros quando chamada

Haskell

```
power :: Int -> Float -> Float
power n b =
   if n = 0 then 1.0
   else b * power (n-1) b

sqr = power 2      -- curried
cube = power 3  -- curried
```

Currificação (*Currying*)

- Exemplo:

Haskell

```
subtrair :: Int → Int → Int
subtrair x y = x – y

-- definir a função negativo a partir de subtrair?
negativo :: Int → Int
negativo = subtrair 0
```

Haskell

```
potencia n x = x^n

-- definir a função quadrado a partir de potencia?
quadrado = potencia 2
```

Estratégias de Avaliação

- Quando exatamente cada parâmetro real de uma função é avaliado quando uma abstração é chamada?
- Duas possibilidades básicas:
  - Avaliar o parâmetro real no local da chamada (**eager evaluation**)
  - Adiar a sua avaliação até que o argumento seja realmente usado (**normal-order / lazy**)

Estratégias de Avaliação

- *Eager (Innermost Reduction / Anciosa / Ávida)*
  - O parâmetro real é avaliado **uma única vez** e o valor resultante **substituído para cada ocorrência do parâmetro formal** na abstração
  - Java, ML, Pascal
- *Lazy (Outermost Reduction / Preguiçosa)*
  - O parâmetro real é avaliado **somente na primeira vez que ele é necessário** (o valor computado é guardado)
  - Haskell
- *Normal-order*
  - O parâmetro real é avaliado **sempre que for necessário** (mesmo que já tenha sido computado)

Estratégias de Avaliação

- Considere a função

```
function quadrado(n: int) = n*n
```

quadrado (3 + 4) => quadrado (7) (+)

eager => n → 7 (vínculo)

=> n*n

=> 49

quadrado (3 + 4) => n → expressão (3+4) (vínculo)

normal => (3+4)*(3+4) (*)

=> 7*(3+4)

=> 7*7

=> 49

Funções estritas a argumentos

- Função é dita estrita a um argumento se ela sempre utiliza aquele argumento
  - Função quadrado é estrita ao seu único argumento

    quadrado n **=** n*n
  - Função teste é estrita ao seu primeiro argumento

    cand b1 b2 **= if** b1 **then** b2 **else** False
    - Chamando cand (x>0, t/x > 50)
      - p/ x = 2 e t = 80, Eager: False e Normal-order: False
      - p/ x = 0 e t = 80, Eager: Falha e Normal-order: False

Propriedade Church-Rosser

"Se existe a possibilidade de uma expressão ser avaliada, ela poderá ser avaliada consistentemente utilizando normal-order. Se uma expressão puder ser avaliada em várias ordens diferentes (misturando normal e eager) então todas as ordens deverão levar ao mesmo resultado"

- Linguagens que permitem efeitos colaterais não respeitam esta propriedade

Estratégias de Avaliação

- Produzem sempre o mesmo resultado?

```
function readInt(f: File) = ....
```

efeito colateral !!!

exemplo de aplicação

```
...quadrado(readInt(arq))...
```

eager: quadrado(10) => 10*10 => 100

normal-order: quadrado(readInt(arq))

=> readInt(arq)*readInt(arq)

=> 10*5

=> 50

arq → a.txt

```
10
5
3
6
...
```

Transparência Referencial

- A execução de uma função sempre produz o mesmo resultado quando dados os mesmos argumentos

- Característica de **linguagens puramente funcionais**

Sugestões de Leitura

- Concepts of Programing Languages (Robert Sebesta)
  - Seções 15.1 até 15.3
- Programming Language Concepts and Paradigms (David Watt)
  - Seções 5.1 e 5.3
  - Seção 14.1