Linguagens de Programação

Fabio Mascarenhas - 2013.1

http://www.dcc.ufrj.br/~fabiom/lp

Casamento de padrões

- Nós vimos que podemos criar listas em Scala usando `List` e `::` e podemos *desmontar* listas usando `isEmpty`, `head` e `tail`
- Várias funções em listas que escrevemos começam com um padrão simples:
  - `if (l.isEmpty) ... else ... l.head ... l.tail`
- Mas usar as funções de desmontar a lista para isso não é *idiomático*
- Scala (e outras linguagens funcionais) prefere que usemos a sua sintaxe de *casamento de padrões*

Match

- O casamento de padrões é uma espécie de *switch*, mas em cima de expressões envolvendo os construtores de alguma estrutura de dados

- Usa a palavra chave match seguindo a expressão que queremos casar, seguida de um *bloco de casos*

- Cada caso associa uma *padrão* com uma expressão

```
def tamanho[T](l: List[T]): Int = l match {
  case Nil => 0
  case h :: t => 1 + tamanho(t)
}
```

Padrões

- Um padrão usa:
  - Construtores, como `List`, `Nil` e `::`
  - Variáveis, como `h`, `t`, `foo`, etc.
  - O *coringa* _
  - Constantes, como `1`, “`foo`”, `true`
- Uma variável só pode aparecer uma vez em um padrão, já que um padrão *define* variáveis

Casando um padrão

- Um padrão como List(p_1, ..., p_n), casa uma lista que pode ser construída com o construtor `List` e argumentos que casam com os padrões p_1, ..., p_n
- Nil casa com a lista vazia
- p_1 :: p_2 casa com uma lista não vazia se p_1 casar com a cabeça da lista e p_2 com a cauda
- Uma variável casa com qualquer valor, e é associada a esse valor dentro da expressão associada ao padrão
- Uma constante casa com um valor igual a ela (ou seja, ela mesma)
- _ também casa com qualquer valor, mas pode ser usado várias vezes

Exemplos

- List(x,2,y) casa com uma lista de três elementos se o segundo elemento for igual a 2, e associa x ao primeiro elemento e y ao terceiro
- x :: 2 :: y :: Nil é equivalente ao padrão acima
- h :: t casa com uma lista não vazia e associa h à cabeça e t à cauda
- _ :: x :: _ casa com uma lista com pelo menos dois elementos, e associa x ao segundo

Avaliando *match*

- Uma expressão `e match { case` p_1 `=>` e_1 `... case` p_n `=>` e_n `}` primeiro avalia e até obter um valor
- Depois tenta casar esse valor com cada padrão p_1, ..., p_n em sequência
- Se casar com o padrão p_i então avalia-se a expressão e_i depois de substituir as ocorrências das variáveis que foram associadas pelo padrão
- Se nenhum padrão casa o resultado é um erro

Padrões com *val* (*destructuring bind*)

- Podemos usar um padrão como lado esquerdo de um *val*
  - `val h :: t = List(1,2,3)`
  - `h` é associado a 1, `t` a List(2,3)
- Tenta casar o valor obtido com o lado direito com o padrão, se não conseguir dá erro

Tuplas

- Listas são sequências com um número arbitrário de elementos do mesmo tipo
- Uma *tupla* é uma sequência com um número *fixo* de elementos de diferentes tipos
  - Generalização de par ordenado
- Um tipo tupla é (T_1, ..., T_n), onde T_1, ..., T_n são tipos quaisquer
- A mesma sintaxe, usada com expressões, (e_1, ..., e_n) é o *construtor* de tuplas
- Para acessar os elementos de uma tupla usamos casamento de padrões com seu construtor

Zipper

- Andar para a “frente” em uma lista é fácil, mas como andamos de volta para “trás”?

Zipper

- Andar para a “frente” em uma lista é fácil, mas como andamos de volta para “trás”?
- Um *zipper* de uma lista é uma estrutura de dados para isso
- A ideia central é manter uma lista com os elementos que foram visitados
- Um zipper para uma List[T] é uma tripla (List[T],T,List[T]) onde o elemento do meio é o *foco* (o elemento na posição atual), e as listas à esquerda à direita são os elementos à esquerda e direita do foco, *na ordem na qual eles aparecem*

Zipper

```
def zipper[T](l: List[T]): (List[T], T, List[T]) = match l {
  case Nil => error(“lista vazia”)
  case h :: t => (Nil, h, t)
}

def paraFrente(z: (List[T], T, List[T])):
   (List[T], T, List[T]) = match z {
  case (e, f, Nil) => error(“final do zipper”)
  case (e, f, h :: t) => (f :: e, h, t)
}

def paraTras(z: (List[T], T, List[T])):
   (List[T], T, List[T]) = match z {
  case (Nil, f, d) => error(“início do zipper”)
  case (h :: t, f, d) => (t, h, f :: d)
}
```