

Tipos, *Pattern Matching*, Recursão de Cauda

Linguagens de Programação

12 de março de 2019

- 1 Introdução
- 2 Construindo novos tipos
 - Tipos “Each-of”
 - Tipos “One-of”
 - Tipos polimórficos
- 3 Mais Pattern-Matching
- 4 Tail Recursion e Acumuladores

Como aprender qualquer Linguagem de Programação

Componentes de uma LP?

- Sintaxe
- Semântica
- *Idioms*
- Bibliotecas
- Ferramentas

São menos relevantes para o curso:

- Bibliotecas
- Ferramentas

Construindo Novos Tipos

CONSTRUINDO NOVOS TIPOS

Construindo novos tipos

Tipos de tipos

- Tipos básicos. Ex: int, bool, unit
- Tipos compostos. Ex: tuplas, listas, Options

Construtores de tipos

- “Each-of” (product types)
- “One-of” (sum types)
- “Self-reference” (recursive types)

Tipos "Each-of"

Product types em ML

- Tuplas (*by position*)
- Registros (*by name*)

Registros

- componentes: campos com nome
- Exemplo de tipo: `{foo : int, bar : int*bool, baz: bool*int}`
- Exemplo de expressão: `{bar = (1+2,true), foo = 3+4, baz = (false,9)}`
- Obtendo valor de um campo: `#foo` e

Exemplo de declaração

```
datatype mytype = Twolnts of int * int  
                | Str of string  
                | Pizza
```

Qual o valor de:

- Twolnts?
- Str?
- Pizza?

Exemplos de uso


```
val pt = Twolnts(19,10)  
val str1 = Str "oi"  
val acabouempizza = Pizza
```

Case Expression

```
fun f x =  
  case x of  
    Pizza => 3  
  | Twolnts(i1,i2) => i1 + i2  
  | Str s => String.size s
```


Exemplo de Oneof

```
datatype naipes = Copas | Paus | Ouro | Espada;  
datatype face = A | Rei | Rainha | Valete | Num of int;  
type carta = face * naipes;  
  
val zap = (Num(4), Paus);  
val escopeta = (Num(7), Copas);  
val espadilha = (A, Espada);  
val pica_fumo = (Num(7), Ouro);  
val manilhas = [zap, escopeta, espadilha, pica_fumo];
```

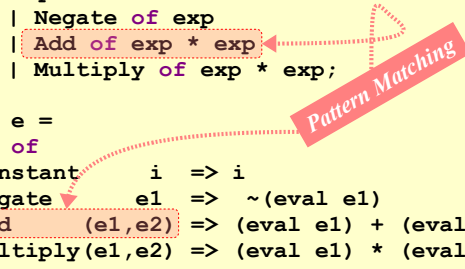


Criando e Avaliando Expressões

```
datatype exp = Constant of int
             | Negate of exp
             | Add of exp * exp
             | Multiply of exp * exp;

fun eval e =
  case e of
    Constant i      => i
  | Negate e1       => ~(eval e1)
  | Add (e1,e2)     => (eval e1) + (eval e2)
  | Multiply(e1,e2) => (eval e1) * (eval e2);

val x = eval (Add (Constant 19,
                  Negate(Constant 4)))
      ;
```



```

datatype my_int_list =
  Empty
  | Cons of int * my_int_list;

fun append_mylist (xs,ys) =
  case xs of
    Empty => ys
  | Cons(x,xs') => Cons(x,append_mylist(xs',ys));
val d1 = Cons(26,Cons(10,Cons(2015,Empty)));

```

```

fun append (xs,ys) =
  case xs of
    [ ] => ys
  | x::xs' => x::append(xs',ys);

```

[] e :: são construtores

```

val d2 = 26::10::2015::[]

```

Não :: (x,xs) MAS x::xs

Tipos polimórficos

Se t for um tipo, então $t \text{ option}$ é um tipo

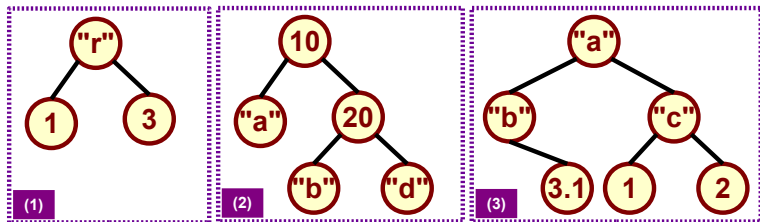
`datatype 't option = NONE | SOME of 't`

Uma árvore binária

`datatype ('a,'b) tree =`

`Node of 'a * ('a,'b) tree * ('a,'b) tree`

`| Leaf of 'b;`



Tipos Polimórficos

Exercício

Escrever funções para percorrer a árvore em pré-ordem, pós-ordem e ordem simétrica.

Tipos Polimórficos

Exercícios (Qual o tipo da função?)

- Função para somar os valores de todos os nós da árvore.
- Função para somar os valores de todas as folhas da árvore.
- Função para contar o número de folhas da árvore.

Mais Pattern Matching

MAIS PATTERN MATCHING

Pattern-Matching para tipos Each-of

Registros e Tuplas

- Registro: $\{f1=v1, \dots, fn=vn\}$; Padrão: $\{f1=x1, \dots, fn=xn\}$; $vi \Leftrightarrow xi$
- Tupla: $(v1, \dots, vn)$; Padrão: $(x1, \dots, xn)$; $vi \Leftrightarrow xi$

Pattern-Matching para tipos Each-of: Usando Case

Soma elementos de uma tripla

```
fun sum_triple (triple : int * int * int) =  
  case triple of  
    (x,y,z) => z + y + z
```

Concatena partes do nome

```
fun full_name (r: {first: string, mid: string, last: string}) =  
  case r of  
    {first=x,mid=y,last=z} => x ^ " " ^ y ^ " " ^ z
```

case com apenas 1 caso NÃO é bom estilo

Pattern-Matching para tipos Each-of: Usando val

Soma elementos de uma tripla

```
fun sum_triple (triple : int * int * int) =  
  let val (x,y,z) = triple  
  in  
    z + y + x  
  end
```

Concatena partes do nome

```
fun full_name (r: first: string, mid: string, last: string) =  
  let val {first=x,mid=y,last=z} = r  
  in  
    x ^ " " ^ y ^ " " ^ z  
  end
```

Pattern-Matching para tipos Each-of: Melhor estilo

Soma elementos de uma tripla

```
fun sum_triple (x,y,z) =  
  z + y + z
```

Concatena partes do nome

```
fun full_name {first=x,mid=y,last=z} =  
  x ^ " " ^ y ^ " " ^ z
```

Toda função tem APENAS 1 argumento!!!

- Uma função com “muitos argumentos” é, na realidade uma função que aceita uma tupla como único argumento.
- Não existem funções sem argumentos

Padrões Aninhados

Exemplos

- $a :: (b :: (c :: d))$: lista com no mínimo 3 elementos
- $a :: (b :: (c :: []))$: lista com exatamente 3 elementos
- $(a, b, c) :: d$: lista não vazia de triplas

Outro Exemplo

```
fun len xs =
  case xs of
    [ ] => 0
  | x::xs' => 1 + len xs'      (* _::xs' => 1 + len xs' *)
```

Pattern-Matching (function bindings)

Exemplo

```
fun append ([], ys) = ys  
  | append (x::xs',ys) = x :: append(xs',ys)
```

Outro Exemplo

```
datatype exp = Constant of int | Negate of exp | Add of exp * exp  
fun eval (Const i) = i  
  | eval (Negate e2) = (eval e2)  
  | eval (Add (e1,e2)) = (eval e1) + (eval e2)
```

Tail recursion e Accumulators

TAIL RECURSION E ACUMULADORS

Recursão de Cauda

```
fun f1 xs = case xs of
  [ ] => 0
  i::xs' => i + sum xs'
```

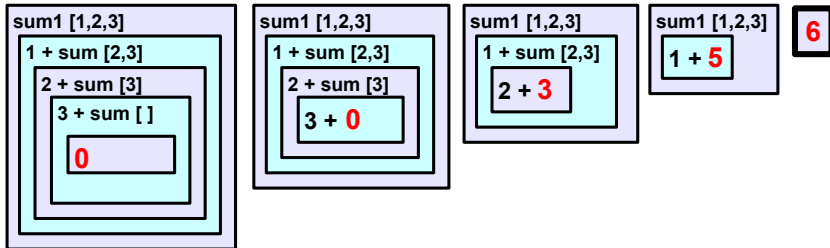
```
fun f2 xs =
  let fun f (xs,acc) = case xs of
    [ ] => acc
    i::xs' => f(xs', i + acc)
  in f(xs,0) end
```

```
val L = [1,2,3]
val r1 = f1 L
val r2 = f2 L
```

*Qual o valor de r1 e r2?
O que f1 calcula? O que f2 calcula?
Qual a diferença?*

Recursão de Cauda

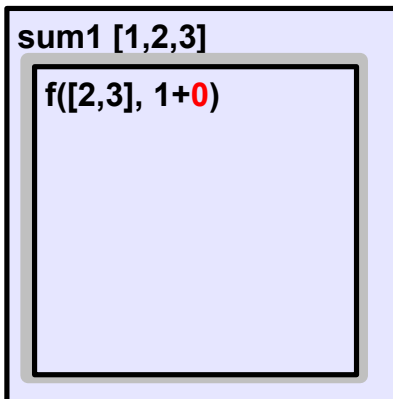
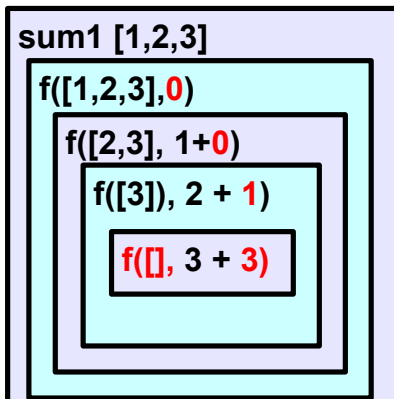
```
fun sum1 xs = case xs of
  [ ] => 0
  i::xs' => i + sum xs'
val s = sum1 [1,2,3]
```



*Muitos registros de ativação "esperando"
o mais interno prover o resultado*

Recursão de Cauda

```
fun sum2 xs =
  let fun f (xs,acc) = case xs of
    [ ] => acc
    i::xs' => f(xs', i + acc)
  in f(xs,0) end
```



Tail Position

Definindo *Tail Position*

- Em `fun f(x) = e`, `e` está em *tail position*
- Se uma expressão não está em *tail position*, nenhuma de suas subexpressões estará
- Se `if e1 then e2 else e3` estiver em *tail position*, então `e2` e `e3` estarão, mas não `e1`
- Se `let b1 ...bn in e end` está em *tail position*, então `e` estará em *tail position*, mas nenhum `bi` estará.
- Argumentos de chamada de funções nunca estarão em *tail position*
- ...