

Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

Recursão mútua com pares

- Agora podemos definir o par de funções mutuamente recursivas:

```
letrec pi = cons(fun (x)
                  if x < 1 then true
                  else snd(pi)(x-1) end
                end,
                fun(x)
                  if x < 1 then false
                  else fst(pi)(x-1) end
                end)
in let par = fst(pi), impar = snd(pi) in (par)(4) end end
```

- Toda essa volta pode parecer um exercício tolo quando já tínhamos funções recursivas no top-level, mas isso é uma prova de que o top-level não é parte essencial da linguagem, e poderia ser compilado para lets e recs!
- De fato, o cálculo lambda só tem três termos: variáveis, funções e aplicações

Tipos algébricos

- Podemos representar estruturas de dados mais complicadas usando funções
- A ideia é um elemento do tipo ser uma função que recebe uma função para cada construtor

- Por exemplo, para listas:

*def tamanho (l: List) =
l match {
case Vazia() =>
0
case Cons(h, t) =>
1 + tamanho(t)
}*

```
fun Cons(h, t)  
  fun (v, c)  
    (c)(h, t)  
  end  
end
```

```
fun Vazia()  
  fun (v, c)  
    (v)()  
  end  
end
```

```
fun tamanho(l)  
  (l)(fun () 0 end,  
      fun (h, t) 1 + tamanho(t) end)  
end
```

*Cons(1,
Cons(2,
Cons(3,
Vazia())
))*

rec na própria linguagem

- Se o cálculo lambda tem apenas variáveis e funções, como conseguimos fazer funções recursivas?
- Existe uma maneira de definir *rec* como uma função!
- Vamos voltar ao exemplo do fatorial:

```
let fat = rec fat = fun (x)
  if x < 2 then 1
  else x * (fat)(x-1) end
end
in (fat)(5) end
```

rec x = T(x)

Duplicação

- Primeiro vamos extrair o núcleo de *rec*, o termo T(x):

```
fun (fat)
  fun (x)
    if x < 2 then 1
    else x * (fat)(x-1) end
  end
end
```

- Mas isso ainda não é a função fatorial! Podemos chegar na fatorial usando um truque:

```
let fat = (let F2 = fun (F1)
  fun (x)
    if x < 2 then 1
    else x * ((F1)(F1))(x-1) end
  end
end
in (F2)(F2) end) ~ (fat)(5) end
```

Função fatorial

- Por que (F2)(F2) é a função fatorial? Vamos expandir o let:

```
(fun (F1)
  fun (x)
    if x < 2 then 1
    else x * ((F1)(F1))(x-1) end
  end
end) (fun (F1)
  fun (x)
    if x < 2 then 1
    else x * ((F1)(F1))(x-1) end
  end
end)
```

Função fatorial

(fun(x) (x) (x) end)

(fun(x) (x) (x) end)

- Agora fazemos a aplicação:

```
fun (x)
  if x < 2 then 1
  else x * ((fun (F1)
              fun (x)
                if x < 2 then 1
                else x * ((F1)(F1))(x-1) end
              end)
            (fun (F1)
              fun (x)
                if x < 2 then 1
                else x * ((F1)(F1))(x-1) end
              end)
            )(x-1) end
end
```

"recursão"

- Agora está se parecendo mais com uma função fatorial!

Função fatorial

- Dentro do corpo da função fatorial temos uma cópia de (F2)(F2), ou seja, fatorial:

```
fun (x)
  if x < 2 then 1
  else x * ((fun (F1)
              fun (x)
                if x < 2 then 1
                else x * ((F1)(F1))(x-1) end
              end)
            (fun (F1)
              fun (x)
                if x < 2 then 1
                else x * ((F1)(F1))(x-1) end
              end)
            )(x-1) end
end
```

Extraindo *fix*

- Podemos extrair a transformação acima para uma função:

```
fun rec fix(f)
  let F2 = fun (F1)
    =      (f)((F1)(F1))
          end
  in (F2)(F2) end
end
```

- E a função fatorial vira (note o uso de um parâmetro CBN!):

```
let fat = fix(fun (_fat)
  fun (x)
    if x < 2 then 1
    else x * (_fat)(x-1) end
  end
end) in (fat)(5) end
```

fix em ação

- Para entender como *fix* funciona, primeiro expandimos o let dentro dela:

```
fun fix(f)
  (fun (F1)
    (f)((F1)(F1))
  end)(fun (F1)
    (f)((F1)(F1))
  end)
end
```

Combinador Y

- Agora podemos aplicar *fix* à função do slide anterior

Fatorial com *fix*

- Aplicando *fix* temos:

```
let fat = (fun (F1)
  (fun (_fat)
    fun (x)
      if x < 2 then 1
      else x * (_fat)(x-1) end
    end
  end)((F1)(F1))
end)(fun (F1)
  (fun (_fat)
    fun (x)
      if x < 2 then 1
      else x * (_fat)(x-1) end
    end
  end)((F1)(F1))
end)
in (fat)(5)
```

Fatorial com *fix*

- Fazendo a aplicação do lado direito do *let*:

```
let fat = fun (x)
```

```
  if x < 2 then 1
```

```
  else x * ((fun (F1)
```

```
    (fun (_fat)
```

```
      fun (x)
```

```
        if x < 2 then 1
```

```
        else x * (_fat)(x-1) end
```

```
      end
```

```
    end)((F1)(F1))
```

```
  end)(fun (F1)
```

```
    (fun (_fat)
```

```
      fun (x)
```

```
        if x < 2 then 1
```

```
        else x * (_fat)(x-1) end
```

```
      end
```

```
    end)((F1)(F1))
```

```
  end))(x-1) end
```

```
end
```

```
in (fat)(5)
```

Por que um parâmetro CBN na função pra *fix*

- A função que passamos para *fix* precisa de um parâmetro CBN, ou *fix* entra em loop infinito!
- Mesmo se a linguagem não tem parâmetros call-by-name podemos evitar o loop, a custo de uma maior carga sintática

```
fun fix(f)
  let F2 = fun (F1)
            (f)(fun () (F1)(F1) end)
          end
  in (F2)(F2)
end
let fat = fix(fun (fat)
              fun (x)
                if x < 2 then 1
                else x * ((fat)())(x-1) end
              end)
in (fat)(5) end
```

$(\lambda x. ((\lambda y. (\lambda z.)))(x)) \rightarrow \text{Combinador Z}$

Efeitos colaterais: referências e atribuição

- A partir de agora vamos começar a sair do mundo funcional e explorar outros paradigmas de programação
- Vamos começar revendo o paradigma *imperativo*, onde o programa não é apenas uma expressão algébrica pura, mas executa *ações* que influenciam um *estado* externo ao programa
- Outro nome para programação imperativa é a programação com *efeitos colaterais*
- Primeiro vamos adicionar uma forma bem simples de efeito colateral a fun, referências e atribuição, e ver como isso muda radicalmente nosso interpretador

Referências de primeira classe

- Vamos adotar o modelo de *referências* de Standard ML (SML)
 - [ML](#) é a avó das linguagens de programação funcionais modernas
 - É um modelo simples mas flexível, diferente das variáveis imperativas
- Uma referência é valor que representa uma caixa para guardar algum outro valor (até mesmo outra referência), e o conteúdo da caixa pode ser lido ou mudado
- Usando referências podemos modelar tanto atribuição simples quanto estruturas de dados imperativas complexas

Operações em referências

- Referências têm três operações primitivas

- Criar uma referência: ref <exp>

valor inicial

- Ler uma referência: !<exp>

valor que contém // uma referência

- Escrever uma referência: <exp> := <exp>

- Também introduzimos a noção de *sequência*, para poder fazer várias operações que modificam referências: <exp>₁ ; <exp>₂

(fun (f) exp2 val) (exp1)

Exemplo: refs em SML

- Um programa simples com referências:

```
let val p = (ref 0, ref 1) in
  (#1 p) := 1;
  (#2 p) := 2;
  p
end
```

- A variável p é um par imutável para duas referências contendo números, o corpo do let escreve novos valores nas duas referências e depois avalia para o valor do par

Refs e funções anônimas em SML

- Com uma referência e uma função anônima podemos criar um contador:

```
let val cont =  
  let val n = ref 0 in  
    fn () => (n := !n + 1; !n)  
  end  
in  
  cont();  
  cont();  
  cont()  
end
```

- A função anônima está modificando a caixa criada fora dela, por isso o valor “persiste” entre as chamadas a ela
- O que acontece se jogarmos a criação da caixa para dentro da função anônima?

~ ~~no~~ cont() sempre retorna 1!

Referências em fun

- Referências são valores de primeira classe, então precisamos de mais um caso no tipo algébrico Valor

```
case class Caixa(v: Valor) extends Valor Exp
```

Caixa *Exp* *Valor*

- Também precisamos de novos casos para Exp:

```
! case class Seq(e1: Exp, e2: Exp) extends Exp  
! case class Atrib(lval: Exp, rval: Exp) extends Exp  
! case class Ref(e: Exp) extends Exp  
! case class Deref(l: Exp) extends Exp
```

- Agora podemos cuidar das definições de eval e step para as novas expressões

Eval – Ref e Deref

- Avaliar os casos Ref e Deref parece ser bem simples
 - Uma Ref avalia a expressão e cria uma nova caixa com aquele valor
 - Uma Deref avalia a expressão, que deve ser uma caixa, e extrai o valor dela
- Só que o que torna referências “especiais” não são essas duas operações, que não são imperativas por si só, mas sim a operação Atrib
- Mas para entender o funcionamento de Atrib, vamos primeiro examinar Seq

Eval - Seq

- Vamos fazer um esboço do que seria uma implementação natural da *eval* para Seq:

```
case Seq(e1, e2) => {  
  eval(e1)  
  eval(e2)  
}
```

- A primeira coisa que notamos é que o valor de *e1* é descartado, mas até aí tudo bem, a primeira expressão da sequência vale apenas pelos seus efeitos colaterais
- Mas para onde estão indo esses efeitos?