

Linguagens de Programação

Fabio Mascarenhas - 2017.2

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

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, e 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>`



- Ler uma referência: `!<exp>`



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



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

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
- Vamos usar um REPL online de SML:
https://www.tutorialspoint.com/execute_smlnj_online.php

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?

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
```

- 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?

Voltando a SML

- A expressão abaixo em SML avalia para 2:

```
let val c = ref 0 in
  c := !c + 1;
  c := !c + 1;
  !c
end
```

- Queremos que seu equivalente *fun* também avalie para Num(2):


```
let c = ref 0 in
  c := !c + 1;
  c := !c + 1;
  !c
end
```

- As duas primeiras expressões na sequência são iguais, mas seu resultado não

Outro exemplo

- A expressão SML abaixo avalia para 3:

```
let val c = ref 0 in
  (c := !c + 1; !c) + (c := !c + 1; !c)
end
```



- Seu equivalente *fun* também deveria avaliar para 3:

```
let c = ref 0 in
  (c := !c + 1; !c) + (c := !c + 1; !c)
end
```

- As subexpressões da soma geram chamadas idênticas para *eval*, mas seu valor deve ser diferente!
- Isso é impossível com *eval* como está, pois viola um preceito básico da programação funcional: mesma entrada => mesma saída

Memória

- Se *eval* tem que mudar alguma coisa isso tem que ficar refletido no seu tipo de saída, que não pode ser mais apenas um valor, mas também os *efeitos colaterais*
- No caso de referências, os efeitos colaterais são as referências que mudaram
- Para rastrear o que foi mudado vamos adicionar um nível de indireção e fazer a caixa de uma referência conter um *endereço* em uma *memória*

```
type End = Int
```

```
type Mem = Map[End, Valor]
```

```
case class Caixa(l: End) extends Valor
```

- A função *eval* agora recebe uma memória, e retorna um valor e uma nova memória

Eval com memória – literais

- Temos quatro termos que já representam valores: Num, True, False e Fun
- Esses são os mais fáceis, já que um literal não pode modificar a memória, então apenas retornamos o valor junto com a mesma memória que foi passada a *eval*

```
case Num(v) => (Num(v), mem)
case True() => (Bool(true), mem)
case False() => (Bool(false), mem)
case Fun(params, corpo) => (Fun(params, corpo), mem)
```

Sequência

- Os efeitos colaterais da primeira expressão de uma sequência claramente afetam a segunda expressão
- Isso quer dizer que a memória resultante de avaliar a primeira expressão deve ser usada como entrada para a segunda:

```
case Seq(e1, e2) => {  
  val (_, nmem) = eval(e1, mem)  
  eval(e2, nmem)  
}
```

- O *valor* resultante da primeira expressão é descartado
- Note a segunda linha: esse padrão vai se repetir bastante daqui pra frente

Aritmética

- Como vimos no exemplo do slide 11, os efeitos colaterais do lado esquerdo da soma afetam o lado direito
- Isso quer dizer que as operações aritméticas (e relacionais) também introduzem uma *sequência* de avaliação, e por isso são avaliadas de um modo parecido com Seq:

```
case Soma(e1, e2) => {  
  val (Num(n1), mem1) = eval(e1, mem)  
  val (Num(n2), mem2) = eval(e2, mem1)  
  (Num((BigDecimal(n1) + BigDecimal(n2)).toString),  
   mem2)  
}
```

- A memória é *costurada* nas subexpressões, é o fio que as conecta

Alocando novas caixas

- A operação Ref cria uma nova caixa, mas qual o endereço dessa nova caixa? Como os endereços são números, podemos guardar em um endereço especial (como 0) o próximo endereço livre:

```
def aloca(v: Valor, mem: Mem): (Valor, Mem) = {  
    val Num(1) = mem(0)  
    val n1 = 1.toInt + 1  
    (Caixa(n1), mem + (0 -> Num(n1.toString),  
                      n1 -> v))  
}
```


Ref, Deref e Atrib

- As operações Ref e Deref agora são fáceis de implementar, contanto que tenhamos cuidado com a sequência
- Lembre que as expressões passadas a Ref e Deref também podem ter efeitos colaterais que devem ser levados em conta!
- As mesmas considerações valem para Atrib; o programa abaixo avalia para 2:

```
let c = ref 0 in  
  (c := !c + 1; c) := !c + 1  
end
```

Aplicações de função e ordem de avaliação

- O mesmo cuidado que tivemos com o sequenciamento das subexpressões dos outros termos deve ser tomado com o sequenciamento dos argumentos de uma aplicação
- Qualquer erro na passagem da memória de uma expressão para a outra introduz furos na linguagem: partes da linguagem onde efeitos colaterais “somem”
- Note que a introdução de efeitos colaterais exige que o interpretador fixe a *ordem de avaliação*, pois ela dá a ordem em que os efeitos acontecem

Bugs

- Um único termo, como Deref, tem várias possibilidades de introduzir *bugs*:

```
let c = ref 0 in  
  !(c := 1; c)  
end
```

```
let c = ref 0 in  
  !(c := 1; c) + !c  
end
```

- Os bugs são introduzidos quando quebramos a *linearidade* da memória; felizmente, podemos abstrair a costura da memória de modo a garantir a linearidade