

Linguagens de Programação

Fabio Mascarenhas - 2017.2

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

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>`

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
```

Imutável

- 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

Ações

- Uma maneira de enxergar o interpretador big-step é como algo que recebe uma lista de funções e uma expressão e retorna uma *ação*
- Ações produzem um valor e mais *efeitos colaterais*
- Podemos representar ações genéricas com um tipo `Acao[+T]`
- Para fun com referências, uma `Acao[+T]` é uma função $\text{Mem} \Rightarrow (T, \text{Mem})$
- Vamos usar um tipo algébrico *implícito* associado, como em Parser, para ter `flatMap/map/filter/withFilter`

Ações primitivas

- A ação mais simples é a que produz um valor sem precisar nem modificar a memória:

```
def empty[T](v: T): Acao[T] = m => (v, m)
```

- Também precisamos de ações que leem e escrevem valores na memória:

```
def le(l: Int): Acao[Valor] = m => (m(l), m)
```

```
def escreve(l: Int, v: Valor): Acao[Valor] = m => (v, m + (l -> v))
```

Ações primitivas - *bind*

- A quarta primitiva que precisamos é uma maneira de encadear ações, passando o resultado de uma para a outra
- Mas uma ação consome apenas uma memória; o jeito de uma ação consumir um valor é usar uma função que produz uma ação dado esse valor
- Isso nos dá a nossa primitiva de sequência, *bind*:

```
def bind[T, U](a: Acao[T], f: T => Acao[U]): Acao[U] = m => {  
  val (v, nm) = a(m)  
  f(v)(nm)  
}
```

Aloca usando as primitivas

- Podemos agora definir a ação aloca como uma combinação dessas primitivas:

```
def aloca(v: Valor): Acao[Valor] =  
  bind(le(0),  
    (lv: Valor) => {  
      val Num(1) = lv  
      val n1 = 1.toInt + 1  
      bind(escreve(0, Num(n1.toString())),  
        (_: Valor) => bind(escreve(n1, v),  
                          (_: Valor) => id(Caixa(n1))))  
    })
```

- A memória agora é costurada implicitamente entre as diferentes ações, então não é possível introduzir bugs acessando memórias “usadas”
- Mas a carga sintática de encadear várias ações com *bind* é grande

bind e flatMap

- Vamos examinar a assinatura de bind:

```
def bind[T, U](a: Acao[T], f: T => Acao[U]): Acao[U]
```



- E comparar com uma velha conhecida, *flatMap*:

```
def flatMap[T, U](l: List[T], f: T => List[U]): List[U]
```

- Só muda o tipo sobre o qual estamos trabalhando, de listas para ações
- Podemos criar definições análogas para ações de *map* e *filter*, também, e usar a sintaxe do *for* para criar nossas ações compostas

Ações com for - *aloca*

- A definição de *aloca* usando for fica muito mais limpa, e com a mesma resistência a bugs no acesso a memória:

```
def aloca(v: Valor): Acao[Valor] = for {  
  Num(1) <- le(0)  
  n1 <- id(1.toInt+1)  
  _ <- escreve(0, Num(n1.toString()))  
  _ <- escreve(n1, v)  
} yield Caixa(n1)
```

- Experimente aplicar as regras de desugaring que vimos para o for, e vamos ter um resultado bem parecido com a definição de *aloca* do slide 13, a menos de se usar *flatMap* e *map* ao invés de *bind*, e de se usar a sintaxe OO de Scala