

Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

Memória

- Como o ambiente, a memória é um mapeamento, mas ao invés de mapear variáveis para valores a memória mapeia *endereços* ou *locais* para valores
- Vamos usar números inteiros para nossos endereços:

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

- A função *eval* agora vai receber, além das funções e do ambiente, uma memória, e vai retornar um valor e uma nova memória contendo os *efeitos colaterais*

Eval com memória – literais

- Temos quatro termos que representam *literals*: 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) => (NumV(v), mem)
case True() => (Bool(true), mem)
case False() => (Bool(false), mem)
case Fun(params, corpo) => (FunV(env, 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) = e1.eval(funs)(env)(mem)  
  e2.eval(funs)(env)(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 (v1, mem1) = e1.eval(funs)(env)(mem)  
  val (v2, mem2) = e2.eval(funs)(env)(mem1)  
  (v1, v2) match {  
    case (NumV(n1), NumV(n2)) => (NumV(n1 + n2), mem2)  
    case _ => sys.error("soma precisa de dois números")  
  }  
}
```

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

Alocando novas caixas

- A operação Ref cria uma nova caixa, mas o que é um valor caixa agora? É um endereço!

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

- Mas em qual endereço vai uma nova caixa? Podemos usar uma caixa para isso, em um endereço especial (vamos usar 0):

```
def aloca(v: Valor)(mem: Mem): (Valor, Mem) = {  
  val NumV(l) = mem(0)  
  val n1 = l.toInt + 1  
  (Caixa(n1), mem + (0 -> NumV(n1), 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 = deref(c) + 1; c) = deref(c) + 1
end
```

Aplicações de função, CBN, ordem

- O mesmo cuidado que tivemos com o sequenciamento das subexpressões dos outros termos deve ser tomado com o sequenciamento dos argumentos de uma aplicação
- A mesma coisa vale quando se avalia a expressão de variáveis CBN
- 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
  deref(c = 1; c)
end
```

```
let c = ref(0) in
  deref(c = 1; c) + deref(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 é como algo que recebe uma lista de funções e um ambiente e retorna uma *ação*
- Ações produzem um valor e mais *efeitos colaterais*, recebendo uma memória e retornando um valor e outra memória
- Podemos representar ações genéricas com um tipo `Acao[T]`, que nada mais é que uma função `Mem => (T, Mem)`
- Como os conjuntos da segunda lista de exercícios

Ações primitivas

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

```
def id[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.get(l) match {  
  case Some(v) => (v, m)  
  case None => sys.error("endereço inválido: " + l)  
}
```

```
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 NumV(1) = lv  
      val n1 = 1.toInt + 1  
      bind(escreve(0, NumV(n1)),  
        (_: 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 {  
  NumV(1) <- le(0)  
  n1 <- id(1.toInt+1)  
  _ <- escreve(0, NumV(n1))  
  _ <- 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

Ações com *for* - aritmética

- Vamos usar a definição de ações com *for* em nosso interpretador, como na implementação do caso Soma de *eval* abaixo:

```
for {
  v1 <- e1.eval(funs)(env)
  v2 <- e2.eval(funs)(env)
} yield (v1, v2) match {
  case (NumV(n1), NumV(n2)) => NumV(n1 + n2)
  case _ => sys.error("soma precisa de dois números")
}
```

- Podemos usar *for* e recursão pra ações mais complexas, como a que avalia os argumentos para uma função
- Também podemos mudar a definição de ação sem precisar reescrever todos os casos do interpretador