Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

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 `Mem => (T, Mem)`
- Vamos usar um tipo algébrico associado, como fizemos com `Parser`

Ações primitivas

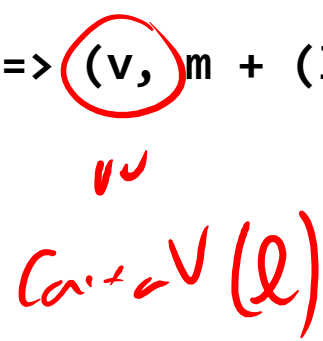
- A ação mais simples é a que produz um valor sem precisar nem modificar a memória:

```
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))
```



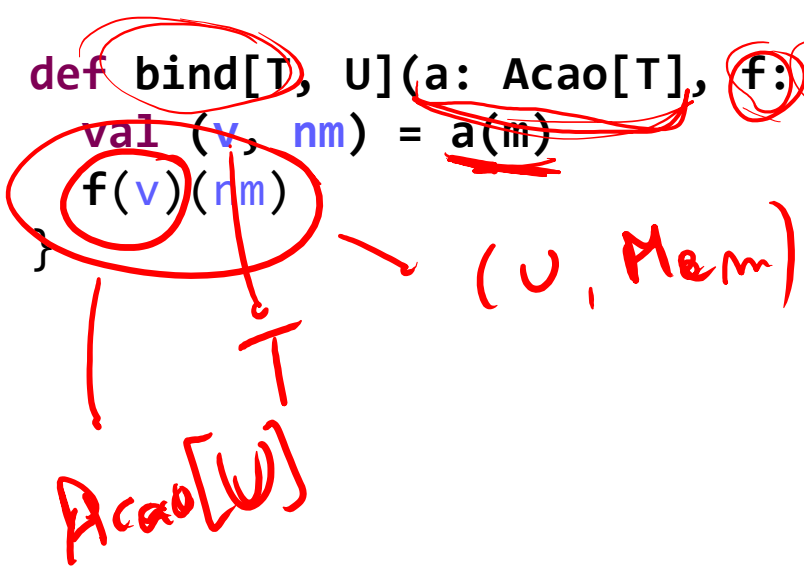
Ações primitivas - *bind*

- 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),
       { case NumV(l) =>
            val nl = l.toInt
            bind(escreve(0, NumV(nl+1)),
                 (_: Valor) => bind(escreve(nl, v),
                                    (_: Valor) => empty(CaixaV(nl))))
       })
```

- 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(l) <- le(0)
  nl <- empty(l.toInt)
  _ <- escreve(0, NumV(nl+1))
  _ <- escreve(nl, v)
} yield CaixaV(nl)
```

- Experimente aplicar as regras de desugaring que vimos para o for, e vamos ter um resultado bem parecido com a definição de *aloca* de dois slides atrás, 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 {
  NumV(n1) <- eval(e1)
  NumV(n2) <- eval(e2)
} yield NumV(n1 + n2)
```

- 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

Chamadas de função

- Para avaliar uma chamada de função, precisamos avaliar a expressão que dá a função, além de todos os argumentos
- Só que cada uma dessas expressões pode ter efeitos colaterais
- A avaliação dos argumentos é uma ação que produz uma *lista* de valores
- Uma ação está limitada a produzir só valores da nossa linguagem, pode produzir qualquer coisa; só nossa função `eval` que está restrita a produzir uma `Acao[Valor]`

Exceções

- Vários erros podem acontecer em nossos programas: fazer aritmética com valores que não são números, chamar coisas que não são funções, ou com o número de parâmetros errados, tentar atribuir ou dereferenciar valores que não são referências...

- Em uma semântica checada, todos esses erros abortariam a execução, retornando um valor de erro

- Mas e se quisermos poder detectar e recuperar esses erros na própria linguagem?

Erros

- Uma `Acao[T]` não vai produzir mais `T`, mas um valor `Talvez[T]`, que é como `Option[T]` com um valor associado ao caso `None`:

```
trait Talvez[T]
case class Ok[T](v: T) extends Talvez[T]
case class Erro[T](msg: String) extends Talvez[T]
```

- Um valor `Erro[T]` faz *bind* entrar em curto circuito, e não continuar com a sua outra ação

- As primitivas *id* e *le* produzem valores `Ok`, e uma nova primitiva `erro` produz um valor `Erro` com alguma mensagem de erro

- O interpretador ainda precisa ser reescrito para checar todas as possíveis condições de erro e chamar `erro` nos locais certos