

Linguagens de Programação

Fabio Mascarenhas - 2013.1

<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, 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

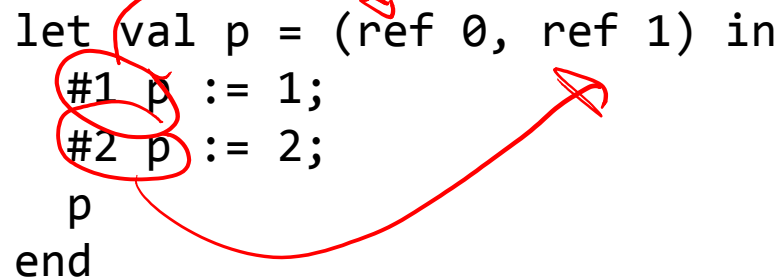
Operações em referências

- 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>`
- Em *fun*, vamos usar *deref* ao invés de `!` para não ter conflito com o `!<exp>`, e exigir parênteses em volta da expressão de *ref* e *deref*

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 closures

- 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* 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) => {  
  e1.eval(funs)(env)  
  e2.eval(funs)(env)  
}
```

- 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 (note os parênteses):

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

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

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

- As duas expressões no parênteses são iguais, mas claramente algo muda

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 = deref(c) + 1; deref(c)) +
  (c = deref(c) + 1; deref(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

E se o ambiente mudar?

- 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 *o que mudou*
- Podemos tentar fazer *o que mudou* ser o ambiente:

```
def eval(funs: Env[Fun1])(env: Env[Valor]): (Valor, Env[Valor]) = ...
```

- Mas isso tem implicações para o escopo léxico, considere o exemplo abaixo:

```
(let c = ref(0) in 1 end) + deref(c)
```

- A variável *c* não está em escopo no lado direito da soma, mas usar o ambiente para registrar mudanças traria *c* para esse escopo

Ambientes não são a solução

- A interação de referências, escopo e funções de primeira classe dá resultados bem consistentes, mas incompatíveis com a ideia de *eval*/mudar o ambiente:

```
let val c = ref 0 in
  let val f = fn (x) => x + !c in
    c := 2;
    f(10)
  end
end
```

- O resultado tem que ser 12
- A ideia é separar a tarefa de manter o escopo léxico feita pelo ambiente, da tarefa de registrar efeitos colaterais, que vai ser feita por mais uma estrutura: a *memória*

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))  
}
```