Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

Passagem por referência

- A passagem por referência é um caso restrito da passagem por nome:

```
fun troca(&a, &b)  -- a e b são por ref
  let tmp = a in
    a = b;
    b = tmp
  end
end
let x = 1, y = 2 in
  -- troca os valores x e y!
  troca(x, y);
  x - y
end
```

- O corpo de *troca* é avaliado como se estivéssemos substituindo *a* e *b* por *x* e *y*, onde *x* e *y* mantém as associações do escpo onde foram definidos

Passagem por referência

- Com a substituição do corpo de *troca*, avaliamos a seguinte expressão:

```
let x = 1, y = 2 in
  let tmp = x in
    x = y;
    y = tmp
  end;
```

- É fácil ver que isso realmente troca os valores das variáveis *x* e *y*!

- A passagem por referência adiciona uma restrição onde os argumentos precisam ser *lvalues*: apenas expressões que podem aparecer do lado esquerdo de uma atribuição

- Os parâmetros por referência são ponteiros dereferenciados implicitamente em cada uso

Mais controle

- Com `Acao` abstraímos que efeitos colaterais a linguagem faz, e garantimos que quando sequenciamos ações os efeitos colaterais são corretamente acumulados

- Mas continuamos tendo pouco controle sobre o *sequenciamento* das ações; no máximo podemos deixar de executar a próxima ação, como quando simulamos exceções

- Vamos ver como aumentar o nosso controle sobre esse sequenciamento, e usar isso para ter várias linhas de execução no programa

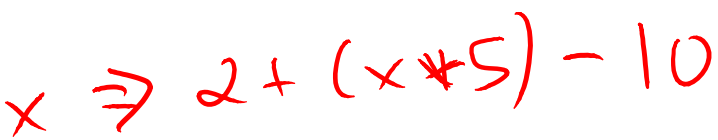
Continuações

- A *continuação* de um ponto do programa é tudo o que tem que ser executado a partir daquele ponto

```
2 + (3 * 5) - 10
```

- No programa acima, a continuação de 3 é “multiplicar por 5, depois somar com 2, e subtrair 10”

- Em geral a continuação é bem comportada, e pode ser dada estaticamente pelo texto do programa



Continuações dinâmicas

- Mas algumas construções mudam dinamicamente a continuação:

```
try
  1 / x
catch div0
  0
end + 2
```

- A continuação de x vai depender se x é 0 ou não: se não for 0, a continuação é "dividir 1 por x, depois somar 2", se for 0 a continuação é "somar 0 com 2"

- A divisão por 0 abandona a parte da continuação da divisão que vem da expressão do `try`, e a substitui pela expressão do `catch`

Abstraindo a continuação

- Podemos representar uma continuação usando uma função (deixando de lado exceções e o tipo `Talvez`):

```
Valor => Comp
```

- Onde `Comp` (de *computação*) é o que o tipo `Acao` do nosso interpretador era. O tipo `Acao` passa a ser:

```
(Valor => Comp) => Comp
```

- Ou seja, uma ação agora recebe uma continuação, e nos dá uma computação que (espera-se) leva essa continuação em conta

Primitivas

- A definição de id para as novas ações é simples:

```
def id(v: Valor): Acao = k => k(v)
```

- Para as primitivas le, escreve, SP, setSP e free precisamos acessar o stack pointer e a memória, e passar quaisquer modificações para a continuação:

```
def le(l: Int): Acao = k => (sp, mem) => mem.get(l) match {
  case Some(v) => k(v)(sp, mem)
  case None => sys.error("endereço inexistente")
}
def escreve(l: Int, v: Int): Acao = k => (sp, mem) => k(v)(sp, mem + (l -> v))
val SP: Acao = k => (sp, mem) => k => k(sp)(sp, mem)
def setSP(l: End): Acao = k => (sp, mem) => k(sp)(l, mem)
def free(l: End): Acao = k => (sp, mem) => k(sp)(sp, mem - l)
```

Bind

- Na definição de `bind` vemos como estamos passando o controle do sequenciamento para “dentro” da ação:

```
def bind(a1: Acao, f: Valor => Acao): Acao = k => a1(v => f(v)(k))
```

- `bind` passa para a primeira ação uma continuação em que obtém a nova ação a partir do valor e de `f` e a chama com a continuação do `bind`

- Para entender por que essa definição, vamos ver o que acontece quando fazemos:

```
bind(escreve(0, 2), _ => le(0))
```

Desenrolando *bind*

$k_0 = v \Rightarrow (sp, m) \Rightarrow (v, sp, m)$

$k_0(z) = (sp, m) \Rightarrow (z, sp, m)$

```
bind(escreve(0, 2), _ => le(0))
```

$k \Rightarrow escreve(0, 2)(v \Rightarrow (\_ \Rightarrow le(0))(v)(k_1))$

$k_1 \Rightarrow (k_2 \Rightarrow (sp, m) \Rightarrow k_2(2)(sp, m + (0 \to 2)))(\_ \Rightarrow le(0))(v)(k_1)$

$k_1 \Rightarrow (sp, m) \Rightarrow ((\_ \Rightarrow le(0))(v)(k_1))(sp, m + (0 \to 2))$

$k_1 \Rightarrow (sp, m) \Rightarrow (le(0)(k_1))(sp, m + (0 \to 2))$

$k_1 \Rightarrow (sp, m) \Rightarrow k_1(2)(sp, m + (0 \to 2))$

$(sp, m) \Rightarrow (2, sp, m + (0 \to 2))$