Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

Coletando o lixo

- Em *fun* com referências, a memória reservada para uma referência não pode ser reciclada facilmente; uma referência pode ser capturada por uma função anônima, ou copiada para outra variável

```
let cont =
  let n = ref 0 in
    fun ()
      n := !n + 1
    end
  end
in
  (cont)();
  (cont)();
  (cont)()
end
```

Coletando o lixo

- MicroC é mais simples: o tempo de vida de uma variável está ligado diretamente ao seu escopo, e quando ela sai de escopo a memória ocupada por aquela variável pode ser reciclada
- Isso só não vale quando usamos &, mas vamos assumir que, do mesmo modo que em C, o ponteiro criado por & é válido apenas no escopo da variável
- Como os escopos são aninhados, essa reciclagem pode ser bem simples: basta decrementarmos o ponteiro que indica qual a próxima posição de memória livre!
- Podemos fazer o mesmo com as parâmetros em uma chamada de função

Pilha de execução

- O interpretador de MicroC recria a estrutura de *pilha de execução*, usada em todos os processadores como uma maneira de gerenciar chamadas de procedimentos

- O ponteiro que indica o próximo endereço livre é o análogo do *stack pointer*

- A pilha é ótima para linguagens como MicroC e C, mas não funciona quando o tempo de vida das variáveis não está ligado a seu escopo, e a execução não segue o aninhamento dado pelo escopo e pelas chamadas de função

- Já vimos que funções de primeira classe furam o modelo de pilha; *threads* também fazem isso

Exceções e a pilha

- Quando abandonamos cada escopo a pilha retorna para o ponto onde estava quando entramos naquele escopo
- Mas o mecanismo atual só faz isso se a execução chegou até o final do escopo
- Quando introduzimos exceções em MicroC isso não é mais verdade: as exceções vazam memória!
- Podemos corrigir isso se *trycatch* fizer esse retorno da pilha caso alguma exceção tenha acontecido
- Para isso quebramos a linearidade do *stack pointer*, isso quer dizer que uma implementação imperativa precisa guardar o valor atual de *sp* antes de um bloco *try*

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 escopo onde foram definidos

Passagem por referência

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

```
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