Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

MicroC

- Para poder explorar outras formas de efeitos colaterais e estruturas de controle, vamos mudar o foco para uma linguagem imperativa simples
- MicroC tem sintaxe parecida com a de fun, mas abandona funções anônimas e tem apenas um único tipo de valor, números inteiros
- MicroC também não tem referências de primeira classe: toda variável pode ser usada como lado esquerdo da atribuição. Isso também vale para parâmetros de funções, mas a passagem ainda é por valor:

```
fun troca(a, b)
  let tmp = a in
    a = b;
    b = tmp
  end
end -- não vai trocar os valores de x e y!
let x = 1, y = 2 in troca(x, y) end
```

Ponteiros

- Para compensar a falta de referências de primeira classe, MicroC tem ponteiros
- Um ponteiro é um número inteiro tratado como um endereço na memória
- MicroC tem dois operadores para lidar com ponteiros:
 - * (Deref) trata o valor de sua expressão como um endereço, e o dereferencia; ele também pode ser usado esquerdo de uma atribuição
 - & (Ender) pode ser usado com variáveis, e dá o endereço da variável

Passagem por referência com ponteiros

• Usando ponteiros podemos escrever uma função troca que funciona:

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

- Quando promovemos esse padrão para a linguagem temos a passagem por referência presente em linguagems como C++ e Pascal
- Também podemos usar ponteiros para ter estruturas de dados em MicroC, como vetores

Ambientes, Memória e Ações

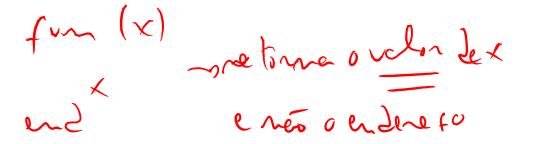
- MicroC só tem um tipo de valor, então expressões relacionais funcionam como em C, produzindo 0 no caso de falso e 1 se verdadeiro
- Como toda variável pode ser atribuída, o ambiente é um mapa de nomes para endereços na memória
- O fato de todas as variáveis serem endereços vai afetar tanto let quanto as chamadas de função
- Também vamos tirar o parâmetro de tipo na Acao de MicroC, e fazer toda Acao produzir um valor de MicroC, ou seja, um número inteiro
- Uma Acao em MicroC é uma função que recebe o próximo endereço livre e a memória e retorna um valor, o próximo endereço livre e uma nova memória

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 = deref(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 em MicroC

- Podemos adicionar exceções a MicroC do mesmo modo que fizemos em Fun, com um tipo algébrico Talvez e primitivas erro e trycatch
- Naturalmente as exceções também serão valores MicroC (números), mas ao invés de códigos de erro elas podem ser ponteiros para estruturas de dados

```
fun erro(x)
   if 0 < x then
      throw(x)
   else
      -x
   end
end

let y = 0 in
      try
      erro((y = 2) + 2) + (y = 1)
   catch err
      err + 10
   end
end</pre>
```

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 escpo 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 *Ivalues*: apenas expressões que podem aparecer do lado esquerdo de uma atribuição
- Os parâmetros por referência são ponteiros dereferenciados implicitamente em cada uso