Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

Ações sem efeitos colaterais

- Podemos retirar os efeitos colaterais de *fun*, mas manter o arcabouço das ações, com uma definição simples:

```
type Acao[T] = T
```

- Naturalmente precisamos ajustar as primitivas, e remover os recursos imperativos da linguagem, e ter de volta uma linguagem idêntica à *fun* original, só que usando *bind* como primitiva de composição, ao invés de usar os mecanismos de Scala

- Isso nos permite explorar outros modelos de como definir e compor ações; por exemplo, podemos fazer `type Acao[T] = Talvez[T]` e introduzir exceções sem outros efeitos colaterais

Não-determinismo

- Uma mudança mais interessante que podemos fazer é introduzir *não-determinismo* a *fun*:

```
type Acao[T] = Stream[T]
```

- A definição das primitivas é simples; de fato, não precisamos nem definir *bind, flatMap, map* e *filter*, pois o tipo Stream de Scala já tem essas operações!

- Só acrescentamos uma primitiva, *amb*:

```
def amb[T](a: Stream[Acao[T]]): Acao[T] = a.flatten
```

fun com não-determinismo

- Para usar a primitiva *amb* vamos ter uma expressão *amb* na linguagem, e com ela podemos escrever programas não-determinísticos

```
fun nums(a, b)
  if a < b then
    amb(a, nums(a+1, b))
  else
    amb()
  end
end

let a = nums(1, 100),
    b = nums(1, 100),
    c = nums(1, 100) in
  if a * a == b * b + c * c then
    a :: b :: c :: nil
  else
    amb()
  end
end
```

Sequências infinitas

- Como o interpretador não-determinístico usa *streams* ao invés de listas, podemos usar *amb* para construir sequências infinitas:

```
fun fibs(n1, n2)
  amb(n1, fibs(n2, n1 + n2))
end

fibs(1, 1)
```

- Se quisermos obter os dez primeiros valores para esse programa podemos usar o seguinte driver:

```
object driver extends App {
  println(parser.parseFile(args(0)).eval.take(10).toList)
}
```

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; toda variável pode ser usada como lado esquerdo da atribuição

- Além disso, MicroC possui 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

- O operador & pode ser usado com variáveis, e dá o endereço da variável

Ambientes e Memória

- MicroC só tem um 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

- Não há passagem por nome em MicroC, mas depois adicionaremos a *passagem por referência*

- O fato de todas as variáveis serem endereços vai afetar tanto *let* quanto as chamadas de função