Linguagens de Programação

Fabio Mascarenhas - 2017.2

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

Mais controle de fluxo

- 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 *corotinas* ou *geradores*

Corotinas

- Uma *corotina* é como uma função que pode suspender a sua execução, retornando ao chamador mas permitindo que a execução seja retomada do ponto onde parou:

```
fun gen()
  let n = 10 in
    while 0 < n do
      yield n;
      n := n - 1
    end
  end
end
```

```
let c = coro gen in
  resume c +
  resume c +
  resume c
end
```

- A primitiva `coro` cria uma corotina a partir de uma função sem parâmetros; a primitiva `resume` inicia/retoma a execução da corotina, e a primitiva `yield` suspende a execução, passando um valor de volta para o chamador

Implementando corotinas

- Mesmo se `yield` só pode ser usado dentro do corpo da corotina não é óbvio como podemos implementar uma corotina, e é comum que yield possa ser usado por qualquer função chamada a partir da função principal da corotina

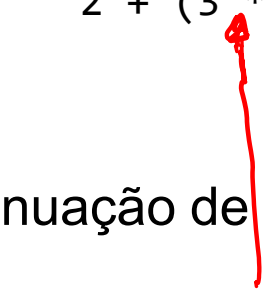
```
fun yielder(n)
  yield n
end

fun gen()
  let n = 10 in
    while 0 < n do
      yielder(n);
      n := n - 1
    end
  end
end
```

- Precisamos de alguma maneira de representar “o ponto atual da execução”

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”

$k \equiv x \Rightarrow (x * 5) + 2 - 10$

- 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
  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

- Corotinas são um exemplo ainda mais radical de mudança da continuação atual

Abstraindo a continuação

- Podemos representar uma continuação usando uma função:

```
type Cont[T] = T => Resp
```

- Onde `Resp` (de *resposta*) é parecido com o que o tipo `Acao` do nosso interpretador era. O tipo `Acao[T]` passa a ser:

```
type Acao[T] = Cont[T] => Resp
```

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

Primitivas

- A definição de `lift` para as novas ações é simples:

```
def lift[T](v: T): Acao[T] = k => k(v)
```

- Para as outras primitivas basta passar o resultado para a continuação, ao invés de retorná-lo

```
def le(r: End): Acao[Valor] = k => (sp, m) => k(m(r))(sp, m)
def escreve(r: End, v: Valor): Acao[Valor] =
        k => (sp, m) => k(v)(sp, m + (r -> v))
```

Bind

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

Continuações

```
def bind[T, U](a: Acao[T], f: T => Acao[U]): Acao[U] = k => a(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))
```