3
Co-rotinas completas assimétricas

O objetivo deste capítulo é prover uma definição precisa para o nosso conceito de co-rotinas completas assimétricas. Em primeiro lugar, descrevemos os operadores básicos desse modelo de co-rotinas. Em seguida formalizamos a semântica desses operadores descrevendo uma semântica operacional para uma linguagem simplificada que os incorpora. Apresentamos, por fim, como uma ilustração do conceito, um mecanismo de co-rotinas que segue, basicamente, a semântica descrita. Usaremos esse mecanismo e a linguagem que o oferece em todos os exemplos de programação apresentados neste trabalho.

3.1
Operadores de co-rotinas completas assimétricas

Nosso modelo de co-rotinas completas assimétricas oferece três operadores básicos: *create*, *resume* e *yield*. O operador *create* cria uma nova co-rotina. O argumento desse operador é uma função que corresponde ao corpo da co-rotina; o valor por ele retornado é uma referência para a co-rotina criada. Uma nova co-rotina não inicia automaticamente sua execução; a co-rotina é criada no estado suspenso, e sua *continuação* é sua função principal.

O operador *resume* (re)ativa uma co-rotina. Seu primeiro argumento é uma referência para uma co-rotina, retornada por uma operação *create* anterior. A co-rotina retoma então sua execução a partir de sua continuação, permanecendo ativa até ser suspensa ou sua função principal terminar. Nos dois casos, o controle é retornado ao ponto de invocação da co-rotina, encerrando a operação *resume*. Quando a função principal de uma co-rotina termina, a co-rotina é considerada morta e não pode mais ser reativada.

O operador *yield* suspende a execução de uma co-rotina, salvando seu estado local, ou continuação; na próxima reativação da co-rotina, seu estado local é restaurado e sua execução é retomada no ponto em que foi suspensa.

Nossos operadores implementam uma facilidade bastante conveniente, permitindo que uma co-rotina e seu chamador troquem dados. Como veremos mais tarde, essa facilidade simplifica a implementação de diversas estruturas de controle, especialmente geradores.

Na primeira vez em que uma co-rotina é ativada, um segundo argumento passado ao operador *resume* é passado como parâmetro para a função principal da co-rotina. Nas reativações seguintes, esse segundo argumento é passado à co-rotina como valor de retorno do operador *yield*. Por outro lado, quando uma co-rotina é suspensa, o argumento passado ao operador *yield* é retornado pelo operador *resume* responsável pela invocação da co-rotina. Quando uma co-rotina termina, o valor retornado pelo operador *resume* é o valor retornado pela função principal da co-rotina.

A facilidade de transferência de dados entre uma co-rotina e seu chamador é ilustrada em um exemplo apresentado na Seção 3.3, onde descrevemos um mecanismo que implementa nosso conceito de co-rotinas completas assimétricas.

3.2 Semântica operacional

Para formalizar nosso conceito de co-rotinas completas assimétricas, descrevemos a seguir uma semântica operacional para seus operadores. As várias similaridades entre co-rotinas completas assimétricas e *subcontinuações* (que discutiremos no próximo capítulo) permitem-nos basear essa semântica na semântica operacional descrita em [46]. Partimos da mesma linguagem básica, uma variante *call-by-value* de λ-cálculo estendida pela incorporação de atribuições. Nessa linguagem básica, o conjunto de expressões (denotado por e) inclui constantes (c), variáveis (x), definições de funções, aplicações de funções e atribuições:

$$e \ \rightarrow \ c \mid x \mid \lambda x.e \mid e\, e \mid x := e$$

As expressões que denotam valores (v) são constantes e funções:

$$v \ \rightarrow \ c \mid \lambda x.e$$

Para permitir efeitos colaterais é incluída na definição da linguagem uma memória (θ), mapeando variáveis para valores:

$$\theta : variáveis \ \rightarrow \ valores$$

A avaliação da linguagem básica é definida por um conjunto de regras de re-escrita, que são aplicadas sucessivamente a pares expressão–memória até que um valor seja obtido. Essas regras de re-escrita são expressas em termos de *contextos de avaliação* [23], que especificam, a cada passo, a próxima subexpressão a ser avaliada. Um contexto é uma expressão que contém um espaço vazio, denotado por $\Box$; $C[e]$ denota a expressão obtida ao se preencher o contexto C com a expressão e. Os contextos de avaliação (C) definidos para a linguagem básica são

$$C \ \rightarrow \ \Box \mid C\, e \mid v\, C \mid x := C$$

Como um argumento está contido em um contexto de avaliação apenas quando o termo na posição da função é um valor, essa definição especifica que aplicações de funções são avaliadas da esquerda para a direita.

As regras de re-escrita para avaliação da linguagem básica são definidas a seguir:

$$\langle C[x],\, \theta\rangle \ \Rightarrow \ \langle C[\theta(x)],\, \theta\rangle \tag{3-1}$$

$$\langle C[(\lambda x.e)v],\, \theta\rangle \ \Rightarrow \ \langle C[e],\, \theta[x \leftarrow v]\rangle,\ x \notin \mathrm{dom}(\theta) \tag{3-2}$$

$$\langle C[x := v],\, \theta\rangle \ \Rightarrow \ \langle C[v],\, \theta[x \leftarrow v]\rangle,\ x \in \mathrm{dom}(\theta) \tag{3-3}$$

A regra 3-1 determina que a avaliação de uma variável tem como resultado o valor dessa variável armazenado na memória θ. A regra 3-2 descreve a avaliação de aplicações; nesse caso, assume-se uma substituição α para garantir a introdução de uma nova variável na memória. Na regra 3-3, que descreve a semântica de atribuições, assume-se a existência prévia de um mapeamento para a variável na memória (isto é, a variável foi previamente criada por uma aplicação).

Para incorporar co-rotinas à linguagem básica, acrescentamos rótulos (l), expressões rotuladas ($l : e$) e os operadores de co-rotinas ao conjunto de expressões:

$$e \ \rightarrow \ c \mid x \mid \lambda x.e \mid e\, e \mid x := e \mid l \mid l : e \mid \mathit{create}\, e \mid \mathit{resume}\, e\, e \mid \mathit{yield}\, e$$

Nessa linguagem estendida, rótulos representam referências para co-rotinas e uma expressão rotulada representa uma co-rotina ativa. Como veremos a seguir, ao rotularmos um contexto somos capazes de identificar a co-rotina a ser suspensa em consequência da avaliação do operador *yield*. Como rótulos referenciam co-rotinas, devemos incluí-los no conjunto de expressões que

denotam valores:

$$v \ \rightarrow \ c \mid \lambda x.e \mid l$$

Estendemos também a definição da memória para permitir mapeamentos de rótulos para valores:

$$\theta : (variáveis \cup rótulos) \ \rightarrow \ valores$$

Além disso, a definição de contextos de avaliação deve incluir as novas expressões. Nessa definição, especificamos que o operador *resume* é avaliado da esquerda para a direita, pois seu primeiro argumento (a referência para uma co-rotina) deve ser reduzido para um rótulo antes que o argumento adicional seja examinado:

$$\begin{array}{lll} C & \rightarrow & \Box \mid C\ e \mid v\ C \mid x := C \mid \\ & & create\ C \mid resume\ C\ e \mid resume\ l\ C \mid yield\ C \mid l : C \end{array}$$

Em nossa semântica, utilizamos, de fato, dois tipos de contexto de avaliação: contextos *completos* (denotados por C) e *subcontextos* (denotados por C'). Um subcontexto é um contexto de avaliação que não contém contextos rotulados ($l : C$), e corresponde à co-rotina ativa mais interna (ou seja, uma co-rotina que não contém qualquer outra co-rotina aninhada) [1].

As regras de re-escrita que descrevem a semântica dos operadores de co-rotina são definidas a seguir:

$$\langle C[create\ v],\ \theta\rangle \ \Rightarrow \ \langle C[l],\ \theta[l \leftarrow v]\rangle,\ l \notin \mathrm{dom}(\theta) \qquad (3\text{-}4)$$

$$\langle C[resume\ l\ v],\ \theta\rangle \ \Rightarrow \ \langle C[l : \theta(l)\ v],\ \theta[l \leftarrow \bot]\rangle \qquad (3\text{-}5)$$

$$\langle C_1[l : C_2'[yield\ v]],\ \theta\rangle \ \Rightarrow \ \langle C_1[v],\ \theta[l \leftarrow \lambda x.C_2'[x]]\rangle \qquad (3\text{-}6)$$

$$\langle C[l : v],\ \theta\rangle \ \Rightarrow \ \langle C[v],\ \theta\rangle \qquad (3\text{-}7)$$

A regra 3-4 descreve a criação de uma co-rotina. Essa regra especifica que um novo rótulo, mapeado para a função principal da co-rotina (a *continuação* da co-rotina) deve ser introduzido na memória.

A regra 3-5 define que a (re)ativação de uma co-rotina produz uma expressão rotulada, que corresponde à continuação da co-rotina, obtida da memória. Essa continuação é então invocada com o argumento adicional passado ao operador *resume*. Para evitar que a co-rotina seja reativada, seu rótulo é mapeado para um valor inválido, denotado por $\bot$.

[1] A definição de subcontextos é trivialmente obtida a partir da definição de contextos completos e, portanto, foi omitida.

A regra 3-6 descreve a suspensão de uma co-rotina. Ela especifica que a avaliação do operador *yield* deve, obrigatoriamente, ocorrer dentro de um subcontexto rotulado (C_2'), resultante da avaliação da operação *resume* que reativou a co-rotina. O objetivo dessa restrição é garantir que a co-rotina retorne o controle a seu ponto de invocação. O argumento passado ao operador *yield* torna-se, então, o valor obtido pela reativação da co-rotina (ou seja, o resultado da operação *resume* correspondente). Além disso, a continuação da co-rotina é salva na memória, substituindo o mapeamento do seu rótulo.

A última regra define a semântica da terminação de uma co-rotina. Nesse caso, o valor obtido pela última reativação da co-rotina é o valor retornado por sua função principal. O mapeamento do rótulo da co-rotina para $\perp$, estabelecido naquela última reativação, evita que uma co-rotina morta seja reativada.

3.3 Co-rotinas em Lua

Lua [48, 49] é uma linguagem de *scripting* que suporta um estilo de programação procedural e oferece facilidades de descrição de dados, escopo léxico e gerência automática de memória. Desde sua versão 5.0, Lua oferece um mecanismo de co-rotinas que segue, basicamente, a semântica que acabamos de descrever.

Lua é uma linguagem tipada dinamicamente, e oferece oito tipos básicos para valores: *nil*, *boolean*, *number*, *string*, *userdata*, *thread*, *function* e *table*. Os tipos *nil*, *boolean*, *number* e *string* têm seu significado usual. O tipo *userdata* permite que dados definidos na linguagem hospedeira (C, por exemplo) sejam armazenados em variáveis Lua. O tipo *thread* representa uma linha de controle independente, e é usado para implementar co-rotinas.

Em Lua, funções são valores de primeira classe, e podem ser, portanto, armazenadas em variáveis, passadas como argumentos e retornadas como resultados de outras funções. Funções Lua são sempre anônimas; a sintaxe

```
function foo(x) ... end
```

é, simplesmente, um açúcar sintático para

```
foo = function (x) ... end
```

Tabelas em Lua são *arrays* associativos, e podem ser indexadas por qualquer tipo de valor. Tabelas Lua podem representar diversos tipos de

estruturas de dados como, por exemplo, *arrays* convencionais, tabelas de símbolos, conjuntos e *records*. Para permitir uma representação conveniente para *records*, Lua usa um nome de campo como índice, suportando `a.name` como um açúcar sintático para `a["name"]`. A criação de tabelas Lua é realizada através de construtores de tabelas. O construtor mais simples (`{}`) cria uma tabela vazia. Construtores de tabelas podem também especificar valores iniciais para campos selecionados, como em `{x = 1, y = 2}`.

Variáveis Lua podem ser globais ou locais. Variáveis globais não são declaradas e recebem `nil` como valor inicial. Variáveis locais devem ser declaradas explicitamente como tal, e tem escopo léxico.

Lua oferece um conjunto de comandos quase convencional, similar aos de Pascal ou C. Esse conjunto inclui atribuições, chamadas de funções e estruturas de controle tradicionais (`if`, `while`, `repeat` e `for`). Lua suporta também facilidades não tão convencionais, como atribuições múltiplas e múltiplos resultados.

O mecanismo de co-rotinas oferecido por Lua [65] implementa nosso conceito de co-rotinas completas assimétricas. Assim como a maioria das bibliotecas de Lua, a biblioteca `coroutine`, que implementa esse mecanismo, oferece suas funções como campos de uma tabela global.

A função `coroutine.create` recebe como argumento uma função Lua que representa o corpo da co-rotina, e retorna uma referência para a co-rotina criada (um valor do tipo *thread*, que corresponde a uma pilha Lua, alocada para a nova co-rotina). O argumento para `coroutine.create` é, frequentemente, uma função anônima, como em

```
co = coroutine.create(function() ... end)
```

Co-rotinas Lua, assim como funções, são valores de primeira classe. Além disso, não há uma operação explícita para destruir uma co-rotina; como qualquer outro valor em Lua, co-rotinas não referenciadas são descartadas pelo coletor de lixo.

As funções `coroutine.resume` e `coroutine.yield` seguem essencialmente a semântica dos operadores *resume* and *yield* descritos anteriormente, permitindo a transferência de dados entre uma co-rotina e seu chamador. Como funções Lua podem retornar múltiplos resultados, essa facilidade é provida também às co-rotinas. Isso significa que a função `coroutine.resume`, além da referência para a co-rotina a ser (re)ativada, pode receber um número variável de argumentos adicionais. Na primeira ativação da co-rotina, os argumentos adicionais recebidos por `coroutine.resume` são passados como parâmetros para a função principal

da co-rotina; em ativações subseqüentes, os argumentos adicionais são transferidos à co-rotina como valores de retorno da chamada correspondente à função `coroutine.yield`. Da mesma forma, quando uma co-rotina é suspensa, ou termina, a chamada a `coroutine.resume` retorna todos os argumentos passados a `coroutine.yield`, ou retornados pela função principal da co-rotina.

Para ilustrar essa facilidade, consideremos a co-rotina criada pelo seguinte trecho de código:

```
co = coroutine.create(function(a,b)
       local c, d  = coroutine.yield(a + b, a - b)
       return "fim", c * d
     end)
```

Se essa co-rotina for ativada por uma chamada como

```
coroutine.resume(co, 20, 10)
```

sua função principal receberá nos parâmetros `a` e `b` os argumentos adicionais passados a `coroutine.resume` (respectivamente, os valores numéricos 20 e 10). Quando a co-rotina solicitar sua suspensão, os valores passados a `coroutine.yield` (os valores numéricos 30 e 10, resultantes da avaliação de `a + b` e `a - b`) serão transferidos ao seu chamador, como resultados da chamada a `coroutine.resume`.

Se essa mesma co-rotina for re-ativada por uma chamada como

```
coroutine.resume(co, 8, 2)
```

os argumentos adicionais passados a `coroutine.resume` (os valores numéricos 8 e 2) serão recebidos pela co-rotina como resultados da chamada a `coroutine.yield`, e armazenados em suas variáveis locais `c` e `d`. Finalmente, quando a co-rotina terminar, os valores retornados por sua função principal (a *string* `"fim"` e o valor numérico 16) serão recebidos pelo chamador como resultados da última reativação da co-rotina (isto é, da última chamada a `coroutine.resume`).

Assim como `coroutine.create`, a função auxiliar `coroutine.wrap` cria uma nova co-rotina, mas ao invés de retornar uma referência para a co-rotina, retorna uma função que, quando chamada, (re)invoca a co-rotina. Os argumentos passados a essa função representam os argumentos adicionais para a operação *resume* correspondente. A função criada por `coroutine.wrap` também retorna a seu chamador todos os valores passados a `coroutine.yield` (ou retornados pela função principal da co-rotina, quando esta termina).

```
-- percorre uma árvore binária
function inorder(node)
  if node then
    inorder(node.left)
    coroutine.yield(node.key)
    inorder(node.right)
  end
end

-- criação do iterador
function make_iterator(tree)
  return coroutine.wrap(function()
                          inorder(tree)
                          return nil
                        end)
end
```

Figura 3.1: Implementando um iterador com co-rotinas Lua

De forma geral, a função `coroutine.wrap` oferece uma maior conveniência que `coroutine.create`; ela provê exatamente o que é usualmente necessário: uma função para reativar uma co-rotina. Por outro lado, o uso das funções `coroutine.create` e `coroutine.resume` permite o gerenciamento de erros. Quando uma co-rotina é reativada por `coroutine.resume`, o primeiro valor retornado por essa função é sempre um valor do tipo *boolean*. Quando uma co-rotina é suspensa, ou termina normalmente, a função `coroutine.resume` retorna o valor `true`, seguido pelos argumentos passados à `coroutine.yield`, ou retornados pela função principal. Na ocorrência de um erro durante a execução da co-rotina, a função `coroutine.resume` retorna o valor `false` e a mensagem de erro correspondente. A função retornada por `coroutine.wrap` não captura erros; qualquer erro provocado pela execução da co-rotina é propagado a seu chamador.

Para ilustrar o uso de co-rotinas em Lua, vamos utilizar um exemplo clássico: um iterador que percorre uma árvore binária de busca em in-ordem, apresentado na Figura 3.1. Nesse exemplo, os nós da árvore são representados por tabelas Lua que contém três campos: `key`, `left` e `right`. O campo `key` armazena o valor do nó (um valor numérico); os campos `left` e `right` contém referências para os nós filhos correspondentes. A função `make_iterator` recebe como argumento o nó raiz de uma árvore binária (`tree`) e retorna um iterador que produz, sucessivamente, os valores armazenados nessa árvore. Como co-rotinas Lua são completas, e, portanto, *stackful*, a implementação desse iterador é feita de forma concisa e elegante:

```
function merge(t1, t2)
  local it1 = make_iterator(t1)
  local it2 = make_iterator(t2)
  local v1 = it1()
  local v2 = it2()

  while v1 or v2 do
    if v1 ~= nil and (v2 == nil or v1 < v2) then
      print(v1); v1 = it1()
    else
      print(v2); v2 = it2()
    end
  end
end
```

Figura 3.2: Combinando duas árvores binárias

a travessia da árvore é executada por uma função recursiva auxiliar que provê o valor de um nó diretamente ao ponto de invocação do iterador. O final da iteração é sinalizado pelo valor `nil`, retornado pela função principal da co-rotina quando esta termina.

A Figura 3.2 apresenta um exemplo de uso do iterador: a combinação de duas árvores binárias de busca. A função `merge` recebe como argumentos as raízes das árvores binárias (`t1` e `t2`), cria iteradores para as duas árvores (`it1` e `it2`) e coleta seus menores elementos (`v1` e `v2`). Em seguida, essa função executa um *loop* que imprime o menor desses valores e reinvoca o iterador correspondente para obter seu próximo elemento. O *loop* é encerrado quando se esgotam os elementos das duas árvores. É interessante observar que a implementação do conceito de co-rotinas *completas* provida por Lua é essencial para o suporte a essa solução trivial para a combinação de duas árvores binárias.