5
Co-rotinas completas como construção genérica de controle

No capítulo anterior nós mostramos que uma linguagem que oferece um mecanismo de co-rotinas completas pode facilmente prover também continuações *one-shot* tradicionais e parciais e, assim, qualquer estrutura de controle implementada com essas construções. Discutimos também como as mesmas vantagens de mecanismos de continuações parciais — maior facilidade de compreensão e suporte a implementações mais sucintas e estruturadas — são oferecidas por co-rotinas assimétricas.

Neste capítulo nós ilustramos essas vantagens e complementamos a demonstração do poder expressivo de um mecanismo de co-rotinas completas assimétricas apresentando implementações de estruturas de controle baseadas nesse mecanismo, incluindo alguns exemplos relevantes do uso de continuações[1]. A facilidade com que esses diferentes comportamentos são implementados é mais um argumento que nos permite contradizer a afirmação de que co-rotinas são uma abstração restrita a alguns usos específicos e muito menos expressiva que continuações [27]. A simplicidade e clareza de alguns desses exemplos contradiz também a afirmação de Knuth [52] de que é difícil encontrar exemplos pequenos, simples e ilustrativos do uso de co-rotinas (essa afirmação pode ser aplicada talvez a alguns mecanismos de co-rotinas simétricas, como o descrito por Knuth).

5.1
O problema do produtor–consumidor

O problema do produtor–consumidor é o exemplo mais paradigmático do uso de co-rotinas. Esse problema envolve, basicamente, a interação de duas computações independentes: uma que *produz* uma sequência de itens e outra que os consome, um a cada vez. Esse tipo de interação constitui um

[1]Por envolver uma discussão mais extensa, o uso de co-rotinas para implementar *multitasking* é apresentado em um capítulo à parte.

```
-- implementação do produtor
function produtor()
  return coroutine.wrap(function()
                          while true do
                            item = produz()
                            coroutine.yield(item)
                          end
                        end)
end

-- implementação do consumidor
function consumidor(prod)
  while true do
    local item = prod()
    consome(item)
  end
end
```

Figura 5.1: O padrão produtor–consumidor com co-rotinas assimétricas

padrão aplicável a diferentes cenários, como, por exemplo, processamento de textos, implementação de compiladores multi-fases, e de protocolos de comunicação.

Diversos exemplos de implementação do padrão produtor–consumidor com co-rotinas (incluindo o exemplo utilizado por Conway para motivar o conceito de co-rotinas [14]) utilizam co-rotinas simétricas, com a transferência explícita de controle entre produtor e consumidor. Co-rotinas assimétricas provêem, contudo, uma solução bem mais simples e estruturada, permitindo a implementação do consumidor como uma função convencional, que invoca o produtor (uma co-rotina assimétrica) quando o próximo ítem é necessário [2]. A Figura 5.1 mostra um esqueleto dessa solução, utilizando o mecanismo de co-rotinas assimétricas oferecido pela linguagem Lua.

Duas características bastante convenientes podem ser observadas na implementação do padrão produtor–consumidor com co-rotinas assimétricas. Em primeiro lugar, o produtor não precisa conhecer o consumidor, pois o controle é implicitamente retornado para este. Em segundo lugar, o código do consumidor independe do fato do produtor ser implementado como uma co-rotina, pois o produtor é utilizado como uma função convencional. O próximo exemplo ilustra a conveniência dessas duas características.

Uma extensão bastante interessante do padrão produtor–consumidor é

[2]Esse é um exemplo de um padrão *consumer-driven*. Quando conveniente, um padrão *producer-driven* pode ser utilizado.

```
-- implementação de um filtro
function transforma(prod, from, to)
  return coroutine.wrap(function()
    local c1, c2
    while true do
      c1 = prod()
      if c1 == from then -- é um possível início do par
        c2 = prod()
        if c2 == from then
          coroutine.yield(to) -- transfere o valor transformado
        else
          coroutine.yield(c1) -- transfere os dois caracteres
          coroutine.yield(c2) -- (um por vez)
        end
      else
        coroutine.yield(c1) -- transfere caracter (não é o par)
      end
    end
  end)
end

-- criação do pipeline
consumidor(transforma(transforma(io.read,"a","b"),"b","c"))
```

Figura 5.2: Implementação de um *pipeline* com co-rotinas assimétricas

a de um *pipeline*, ou seja, uma cadeia composta por um produtor inicial, um ou mais *filtros* que realizam alguma transformação nos itens transferidos e um consumidor final. Co-rotinas assimétricas provêem uma solução elegante e trivial para esse tipo de estrutura. Um filtro comporta-se tanto como um consumidor quanto como um produtor, e pode ser implementado por uma co-rotina assimétrica que ativa seu antecessor para obter um novo valor, transforma esse valor e em seguida suspende sua execução, transferindo o valor transformado ao seu chamador (o próximo consumidor na cadeia, que tanto pode ser um outro filtro como o consumidor final).

A Figura 5.2 ilustra o uso de um pipeline construído com co-rotinas assimétricas. O código mostrado implementa um programa que consome uma sequência de caracteres lidos da entrada padrão. Antes de alcançar o consumidor final, essa sequência sofre duas transformações, realizadas por dois filtros. Na primeira transformação, um par de caracteres `"aa"` é transformado em um único caracter `"b"`. Na segunda transformação, um par de caracteres `"bb"` é transformado em um único caracter `"c"`. O produtor inicial do *pipeline* é a função `read`, oferecida pela biblioteca de entrada e saída de Lua. Note como nessa solução um único comando estabelece o

pipeline, criando e conectando os componentes na sequência apropriada, e ativando o consumidor final.

A solução original desse problema com co-rotinas foi apresentada por Grune [37]. Entretanto, essa solução utiliza co-rotinas simétricas, e não aplica o conceito de um *pipeline*.

5.2 Problemas multi-partes

A solução de problemas multi-partes ilustra um outro uso interessante de co-rotinas assimétricas. Esse tipo de problema, conforme descrito por Friedman et al [27], tem um enunciado como "assumindo resultados das partes a e b, prove c". A solução do problema não implica, necessariamente, na obtenção de todos, ou mesmo algum dos resultados das partes a e b; no caso típico, apenas resultados parciais são suficientes.

A solução do problema multi-partes tem assim uma estrutura onde a execução de c é entremeada com invocações das partes a e b, na busca de resultados parciais que possam auxiliar a solução global. Friedman et al implementam essa solução utilizando "pilhas" de continuações para transferir o controle entre o solucionador do problema e as partes individuais. Entretanto, essa estrutura pode ser vista como uma generalização do problema do produtor–consumidor, na qual um consumidor obtem itens de múltiplos produtores.

A Figura 5.3 mostra a implementação de um problema multi-partes com co-rotinas completas assimétricas. A função `multipart` recebe como parâmetros as funções que implementam as duas partes (`a` e `b`) e a função responsável por tentar resolver o problema (`c`). Para cada uma das partes (os produtores de resultados) é criada uma co-rotina assimétrica; as referências para essas co-rotinas são passadas ao solucionador do problema (o consumidor). Quando o solucionador deseja obter um resultado parcial de uma das partes, ele utiliza a referência correspondente para ativá-la, invocando-a como uma função convencional. Quando uma parte alcança um resultado parcial, ela suspende sua execução (através de uma chamada a `coroutine.yield`), retornando esse resultado ao solucionador.

```
-- monta a resolução do problema
function multipart(a,b,c)
  local part_a = coroutine.wrap(a)
  local part_b = coroutine.wrap(b)
  return c(part_a, part_b)
end

-- esqueleto de um problema
function c(part_a,part_b)
    ..
    val1 = part_a(x)  -- obtém resultado parcial de a
    val2 = part_b(y)  -- obtém resultado parcial de b
    if ... then
      ...
      return res -- problema resolvido
    end
    ...
    val3 = part_b(z)  -- obtém próximo resultado parcial de b
    ...
end
```

Figura 5.3: Resolução de um problema multi-partes

5.3 Geradores

Um gerador é uma estrutura de controle que produz uma sequência de valores. É simples perceber que essa estrutura é, novamente, apenas uma instância particular do padrão produtor–consumidor: um gerador é um *produtor* de itens *consumidos* por seu usuário. Co-rotinas assimétricas são, como vimos, uma construção bastante conveniente para a implementação desse padrão e, conseqüentemente, para a implementação de geradores.

É interessante observar que a recorrência do padrão produtor–consumidor em diferentes cenários, e a conveniência do uso de co-rotinas completas para implementar esse padrão, condizem com a conjectura de concisão de Felleisen [25], segundo a qual programas em linguagens que provêem abstrações expressivas contêm um número menor de padrões que programas equivalentes em linguagens que não oferecem a mesma expressividade.

No capítulo 4, mostramos que a implementação de geradores com continuações requer a manutenção explícita do estado de um gerador através da captura e gerência de continuações. O uso de co-rotinas assimétricas permite uma implementação bem mais simples e direta, pois o estado de uma co-rotina é automaticamente mantido quando ela é suspensa. Essa maior

```
function makegenfact()
  return coroutine.wrap(function()
    local f,i = 1,1        -- inicializa próximo fatorial
    while true do
      coroutine.yield(f)   -- retorna fatorial corrente
      i = i + 1; f = f * i -- calcula o próximo
    end
  end)
end

-- cria o gerador
genfact = makegenfact()

-- usa o gerador para imprimir os 5 primeiros fatoriais
for i = 1,5 do
  print(genfact())
end
```

Figura 5.4: Gerador de fatoriais com co-rotinas assimétricas

simplicidade, bastante adequada especialmente num contexto de linguagens procedurais, pode ser comprovada pela comparação das implementações do gerador de números fatoriais com subcontinuações (Figura 4.5) e com co-rotinas assimétricas (Figura 5.4).

Um uso bastante comum de geradores é para a implementação de iteradores de estruturas de dados. Um exemplo típico — a travessia de uma árvore binária — foi apresentado no Capítulo 3 (Figura 3.1). Entretanto, geradores não são apropriados apenas para esse tipo de aplicação. Na próxima seção apresentamos um exemplo de uso de geradores em um cenário bastante distinto.

5.4 Programação orientada por metas

A programação orientada por metas (*goal-oriented programming*) envolve basicamente a solução de um problema através de um mecanismo de *backtracking*. O problema a ser resolvido é tipicamente expresso como uma *disjunção* de metas alternativas, e o mecanismo de *backtracking* é responsável por tentar satisfazer essas metas até que um resultado adequado seja encontrado. Cada meta alternativa pode ser tanto uma meta *primitiva* como uma *conjunção* de submetas que devem ser satisfeitas em sequência, cada uma delas contribuindo com uma parte do resultado final.

A solução de problemas de *pattern matching* [36] e a implementação de linguagens como Prolog [13] são dois exemplos de uso desse tipo de mecanismo. Em *pattern matching*, por exemplo, o casamento de uma sequência de caracteres com uma *string* é uma meta primitiva, o casamento com algum dos padrões de um conjunto representa uma disjunção, e o casamento com uma sequência de subpadrões é uma conjunção de submetas. Na solução de *queries* em Prolog, o processo de unificação é uma meta primitiva, uma *relação* é uma disjunção e *regras* são basicamente conjunções de submetas.

A implementação dessa forma de *backtracking* é citada como um dos poucos exemplos onde o uso de continuações *one-shot* é impossível [10, 53]. De fato, implementações desse comportamento com continuações tradicionais, como a desenvolvida por Haynes [42], utilizam continuações *multi-shot* para receber e testar as soluções alternativas. Entretanto, a programação orientada por metas pode ser vista como uma simples aplicação de geradores: uma meta é basicamente um gerador que produz uma sequência de soluções. Dessa forma, tanto o uso de continuações parciais *one-shot* (como mostra Sitaram [79]), como o uso de co-rotinas assimétricas, são possíveis e também mais adequados, pois simplificam consideravelmente a estrutura da implementação [3].

A implementação de programação orientada por metas com co-rotinas assimétricas é bastante simples. A implementação de uma meta como o corpo de uma co-rotina assimétrica (um gerador) permite que um simples *loop* seja capaz de receber e testar as soluções alternativas para um problema. Uma meta primitiva é uma função que produz um resultado a cada invocação. Uma disjunção é uma função que invoca suas metas alternativas em sequência. Uma conjunção de duas submetas pode ser definida como uma função que itera sobre a primeira submeta, invocando a segunda para cada resultado retornado (a extensão dessa solução para uma conjunção de três ou mais submetas pode ser obtida com a combinação de conjunções).

Como ilustração, vamos considerar um problema de *pattern-matching*. A meta a ser satisfeita é o casamento de uma *string* `S` com um padrão `patt`, que pode combinar subpadrões alternativos (denotados por `<subpadrao1>|<subpadrão2>`) e sequências de subpadrões (denotadas por `<subpadrao1>.<subpadrão2>`). Um exemplo desse tipo de padrão é

```
("abc"|"de")."x"^
```

[3] Na verdade, mesmo formas restritas de co-rotinas, como os geradores de Icon, provêem suporte a esse estilo de programação.

```
-- meta primitiva (casamento com uma string literal)
function prim(str)
  local len = string.len(str)   -- tamanho do padrão
  return function(S, pos)
           if string.sub(S, pos, pos+len-1) == str then
             coroutine.yield(pos+len)
           end
         end
end

-- disjunção (padrões alternativos)
function alt(patt1, patt2)
  return function(S, pos)
           patt1(S, pos)
           patt2(S, pos)
         end
end

-- conjunção (sequência de subpadrões)
function seq(patt1, patt2)
  return function(S, pos)
           local btpoint = coroutine.wrap(function()
                                            patt1(S, pos)
                                          end)
           for npos in btpoint do patt2(S, npos) end
         end
end
```

Figura 5.5: Progamação orientada por metas: *pattern-matching*

A Figura 5.5 mostra a implementação desse tipo de problema utilizando co-rotinas assimétricas de Lua. Nessa implementação, cada uma das funções responsáveis por tentar um casamento recebe como parâmetros a *string* em teste e uma posição inicial. Para cada casamento obtido, a próxima posição a ser verificada é retornada. Quando nenhum casamento pode mais ser obtido, o valor `nil` é retonado.

A função `prim` constrói uma meta primitiva. Ela recebe como parâmetro um valor do tipo *string* e retorna uma função que tenta casar esse valor com a *substring* de `S` que começa na posição indicada.

A função `alt` é responsável por construir uma disjunção. Ela recebe como parâmetros duas metas alternativas e retorna uma função que invoca essas metas para tentar obter um casamento da *substring* de `S` que inicia na posição indicada. Note que para cada casamento obtido, a próxima posição a ser testada é retornada diretamente ao chamador da função definida por `alt`.

Uma conjunção é construída pela função `seq`, que recebe como parâmetros as duas submetas a serem satisfeitas. A função retornada por `seq` cria uma co-rotina auxiliar (`btpoint`) para iterar sobre a primeira submeta. Cada casamento obtido pela invocação dessa submeta produz uma posição em `S` a partir da qual a segunda submeta deve ser satisfeita. Se um casamento é obtido por essa segunda submeta, a próxima posição a ser testada é retornada diretamente ao chamador da função criada por `seq`.

Utilizando as funções que acabamos de descrever, podemos definir o padrão `("abc"|"de")."x"` como

```
patt = seq(alt(prim("abc"), prim("de")), prim("x"))
```

Uma função que verifica se uma *string* casa com um padrão especificado pode ser implementada como a seguir:

```
function match(S, patt)
  local len = string.len(S)
  local m = coroutine.wrap(function() patt(S, 1) end)
  for pos in m do
    if pos == len + 1 then -- sucesso quando fim da string
      return true          -- é alcançado
    end
  end
  return false
end
```

5.5 Tratamento de Exceções

Um mecanismo de tratamento de exceções tipicamente implementa duas primitivas básicas: `try` e `raise` [29]. A primitiva `try` recebe como argumentos um corpo (uma sequência de comandos a serem executados) e um tratador de exceções. Quando a execução do corpo termina normalmente, o valor produzido por essa execução é o resultado da invocação de `try`, e o tratador de exceções é ignorado. Se durante a execução do corpo a primitiva `raise` é invocada, uma exceção é lançada. Nesse caso, a exceção é imediatamente enviada ao tratador de exceções e a execução do corpo é abandonada. Um tratador de exceções pode tanto retornar um valor (que se torna o resultado da invocação de `try`) como lançar uma outra exceção, que será então enviada ao próximo tratador de exceções, isto é, o tratador correspondente à estrutura `try` mais externa.

```
-- nivel de aninhamento corrente
local handler_level = 0

-- primitiva try
function try(body, handler)
  handler_level = handler_level + 1
  local res = coroutine.wrap(body)()     -- executa o corpo
  handler_level = handler_level - 1

  if is_exception(res) then
    return(handler(get_exception(res)))  -- exceção foi lançada
  else
    return res                           -- retorno normal
  end
end

-- lançamento de uma exceção
function raise(e)
  if handler_level > 0 then
    coroutine.yield(create_exception(e)) -- retorna a try
  else
    error "exceção não tratada"  -- emite erro de execução
  end
end
```

Figura 5.6: Implementação de um mecanismo de tratamento de exceções

Co-rotinas completas assimétricas provêem suporte a uma implementação bastante simples de um mecanismo de exceção, como mostra a Figura 5.6. A primitiva `try` pode ser implementada por uma função que executa seu primeiro argumento (uma função que representa o corpo) em uma co-rotina assimétrica. Quando o resultado da invocação da co-rotina é uma exceção, a função `try` invoca o tratador de exceções (seu segundo argumento), repassando o valor da exceção recebida. Caso contrário, a função `try` retorna o valor recebido da co-rotina. A primitiva `raise` é simplesmente uma função que suspende a execução da co-rotina, transferindo a representação do valor de seu argumento como uma exceção. O uso de uma variável que indica o nível de aninhamento corrente (`handler_level`) permite detetar se o lançamento de uma exceção é válido (isto é, se a primitiva `raise` foi chamada dentro de uma estrutura `try`).

Para obter uma forma segura de distinguir exceções de retornos normais (que podem ser valores de qualquer tipo), podemos representar uma exceção como uma tabela Lua com dois índices: a *string* `"value"` (que indexa a posição da tabela que contém o valor da exceção) e uma referência

```
-- índice usado em exceções
local ekey = {}

-- cria uma exceção
local function create_exception(v)
  return { [ekey] = true, value = v }
end

-- verifica se é representação de uma exceção
local function is_exception(e)
  return type(e) == "table" and e[ekey]
end

-- obtem o valor correspondente a uma exceção
local function get_exception(e)
  return e.value
end
```

Figura 5.7: Criação e manipulação de exceções

para um tabela específica; a presença desse segundo índice nos permite distinguir uma exceção de uma tabela Lua normal. A Figura 5.7 ilustra essa representação, e mostra funções auxiliares para a criação e manipulação de exceções.

5.6 Evitando interferências entre ações de controle

O aninhamento de diferentes estruturas de controle pode às vezes causar interferências indesejáveis entre essas estruturas. Uma interferência indesejável pode ocorrer, por exemplo, se um iterador utilizado em um corpo de comandos executado por uma primitiva `try` lança uma exceção. Nesse caso, em vez do envio da exceção ao tratador correspondente (pelo retorno do controle à primitiva `try`), o controle é retornado ao usuário do iterador, que, erradamente, interpreta a exceção como um valor produzido pelo iterador.

Esse tipo de interferência pode ser evitado pela associação explícita de pares de operações de controle. Essa associação pode ser implementada através da atribuição de um *tag* diferente (uma *string*, por exemplo). a cada tipo de estrutura de controle. Dessa forma, a solicitação de suspensão de uma co-rotina pode identificar a que ponto de invocação (isto é, a que operação *resume*) o controle deve ser retornado. É interessante notar que a idéia básica dessa solução é similar à utilizada na implementação

```
-- salva a definição original de coroutine.wrap
local wrap = coroutine.wrap

-- redefinição de coroutine.wrap
function coroutine.wrap(tag, f)

   -- cria uma co-rotina "tagged"
   local co = wrap(function(v) return tag, f(v) end)
   return function(v)
            local rtag, ret = co(v) -- ativa co-rotina

            while (rtag ~= tag) do
              -- reativa co-rotina externa se os tags diferem
              v = coroutine.yield(rtag, ret)

              -- na reinvocação, reativa co-rotina interna
              tag, ret = co(v)
            end

            -- se tags iguais, retorna o controle ao chamador
            return ret
          end
end
```

Figura 5.8: Evitando interferências entre ações de controle

de subcontinuações *one-shot* com co-rotinas assimétricas, para casar uma subcomputação com o controlador correspondente (veja a Seção 4.3).

Esse tipo de solução pode ser implementado em Lua com uma redefinição da função `coroutine.wrap`, que passa a receber dois parâmetros: o *tag* a ser associado à nova co-rotina, e a função que implementa seu corpo. Adicionalmente, a função `coroutine.yield` passa a requerer como um primeiro parâmetro obrigatório um *tag* que identifica a que estrutura de controle a operação de suspensão se refere. A função retornada pela nova versão de `coroutine.wrap`, antes de retornar o controle a seu chamador, verifica se o *tag* retornado pela co-rotina é o mesmo ao qual ela foi associada. Se é, o controle é retornado ao chamador. Se não, o *tag* se refere a uma co-rotina externa; nesse caso, o controle é retornado à próxima co-rotina (através de uma nova invocação a `coroutine.yield`). Esse procedimento se repete até que a operação *resume* relacionada ao *tag* seja atingida. A Figura 5.8 mostra essa nova versão de `coroutine.wrap`.