4
Co-rotinas completas e continuações one-shot

A literatura disponível habitualmente descreve co-rotinas assimétricas (ou *semi co-rotinas*) como uma construção menos poderosa que co-rotinas simétricas [62, 69]. Além disso, co-rotinas são vistas como uma abstração restrita a usos específicos, e portanto menos expressiva que continuações de primeira classe [27, 41].

Segundo o conceito de expressividade desenvolvido por Felleisen [25], uma construção é "mais expressiva" que outra quando a tradução de um programa que utiliza a primeira dessas construções para uma implementação com a segunda ou não é possível ou exige uma reorganização de todo o programa. Fundamentados por esse conceito, demonstraremos neste capítulo que co-rotinas completas simétricas e assimétricas e continuações *one-shot* têm na verdade o mesmo poder expressivo.

Nossa idéia básica é mostrar que uma linguagem que oferece um mecanismo de corotinas completas assimétricas, como Lua, pode facilmente prover também co-rotinas simétricas e continuações *one-shot* tradicionais e parciais, e portanto qualquer estrutura de controle baseada nessas construções. As implementações de co-rotinas simétricas e continuações *one-shot* são oferecidas através de bibliotecas Lua que provêem os operadores básicos de cada uma dessas construções. Essas bibliotecas constituem um mecanismo de tradução que mantém inalterados não apenas a estrutura de um programa, mas inclusive seu próprio código. Podemos argumentar, assim, que nossa demonstração é baseada em uma noção de equivalência de expressividade mais forte que a de Felleisen.

Além de demonstrar a equivalência de poder expressivo entre co-rotinas simétricas e assimétricas, e entre co-rotinas completas e continuações *one-shot*, discutiremos também semelhanças e diferenças entre esses mecanimos.

4.1 Equivalência de co-rotinas completas simétricas e assimétricas

Quando co-rotinas são *completas*, ou seja, quando são oferecidas como construções *stackful* de primeira classe, mecanismos simétricos e assimétricos podem ser facilmente providos um pelo outro. As implementações apresentadas a seguir comprovam essa afirmação, e demonstram que co-rotinas simétricas e assimétricas, contrariando a noção comum, têm de fato o mesmo poder expressivo.

4.1.1 Implementação de co-rotinas simétricas com assimétricas

A partir de um mecanismo de co-rotinas assimétricas completas, podemos obter uma implementação bastante simples de co-rotinas simétricas. Uma co-rotina simétrica tipicamente representa uma linha de execução independente, e, portanto, não pode ser confinada a uma estrutura ou localização específicas em um programa. Como uma co-rotina assimétrica completa é um valor de primeira classe, é possível utilizá-la para representar uma co-rotina simétrica. A operação simétrica *transfer* pode ser simulada através de um par de operações assimétricas *yield–resume*, e de um um *dispatcher* que atua como um intermediário na transferência de controle entre duas co-rotinas.

A Figura 4.1 apresenta uma biblioteca Lua (`symcoro`) que oferece co-rotinas simétricas baseadas no mecanismo de co-rotinas assimétricas provido por essa linguagem. A função `symcoro.create` cria uma nova co-rotina simétrica. Ela recebe como parâmetro a função principal da co-rotina (`f`), e retorna uma função que permite (re)ativá-la, obtida pela chamada a `coroutine.wrap`. A função `symcoro.transfer` implementa a operação simétrica de transferência de controle. Seu primeiro argumento é uma referência para a co-rotina à qual o controle deve ser transferido (uma função retornada por `symcoro.create`). O segundo argumento, opcional, permite a troca de dados entre co-rotinas. Para que co-rotinas possam também transferir o controle ao programa principal, a biblioteca `symcoro` o representa simulando uma referência para uma co-rotina (o campo `main`). Um outro campo auxiliar (`current`) armazena uma referência para a co-rotina em execução.

Quando a função `symcoro.transfer` é chamada pelo programa principal, o *loop* que implementa o dispatcher é iniciado, e o controle é transferido à co-rotina designada através de uma operação assíncrona *resume* (indi-

```
  symcoro = {}    -- biblioteca de co-rotinas simétricas

  symcoro.main = function() end   --  programa principal
  symcoro.current = symcoro.main  --  co-rotina ativa

  -- criação de uma co-rotina simétrica
  function symcoro.create(f)
    local co = function(val)
                f(val)
                error("co-rotina não transferiu o controle")
               end
    return coroutine.wrap(co)
  end

  -- operação de transferência de controle
  function symcoro.transfer(co, val)
    if symcoro.current ~= symcoro.main then
      return coroutine.yield(co, val)
    end

    -- dispatcher
    while true do
      symcoro.current = co
      if co == symcoro.main then
        return val
      end
      co, val = co(val)
    end
end
```

Figura 4.1: Implementando co-rotinas simétricas com assimétricas

retamente invocada pela função retornada por `coroutine.wrap`). Quando chamada por uma co-rotina, a função `symcoro.transfer` utiliza a operação assíncrona *yield* para suspender a co-rotina e reativar o *dispatcher*, ao qual são repassados a referência para a próxima co-rotina a ser ativada e o valor a ser transferido. Como uma co-rotina Lua é *stackful*, é possível suspendê-la durante a execução de uma função aninhada. Essa característica viabiliza a implementação da operação *transfer* como uma função de uma biblioteca.

Se a referência para a próxima co-rotina representa o programa principal, o *loop* que implementa o *dispatcher* é encerrado, terminando a chamada original a `symcoro.transfer`. Caso contrário, essa referência é utilizada para a ativação da co-rotina correspondente. Se essa co-rotina foi suspensa em uma chamada a `symcoro.transfer`, essa chamada termina, retornando à co-rotina o valor repassado pelo *dispatcher* (recebido como resultado da invocação a `coroutine.yield`). Na primeira ativação da co-

rotina, esse valor será recebido como o argumento de sua função principal.

A semântica de terminação de co-rotinas simétricas é uma questão em aberto [69]. Em algumas implementações, quando a função principal de uma co-rotina termina, o controle é retornado implicitamente ao programa principal. Em nossa implementação, adotamos a semântica das co-rotinas de Modula-2 [90], que especifica que o término de uma co-rotina sem uma transferência explícita de controle constitui um erro de execução. Para implementar essa semântica, a função `symcoro.create` define como corpo de uma co-rotina simétrica uma função que emite um erro de execução, terminando o programa principal, quando a função principal da co-rotina retorna.

4.1.2 Implementação de co-rotinas assimétricas com simétricas

Prover co-rotinas assimétricas completas a partir de um mecanismo de co-rotinas simétricas também não apresenta dificuldades. Para implementar a semântica de co-rotinas completas assimétricas, definida no Capítulo 3, é necessário apenas incluir na representação de uma co-rotina uma referência para o seu chamador e uma informação de estado; essa informação permitirá a detecção de operações de controle inválidas, como a reativação de uma co-rotina morta ou correntemente ativa.

O código da Figura 4.2 implementa uma biblioteca Lua (`asymcoro`) que oferece um mecanismo de co-rotinas completas assimétricas similar ao mecanismo nativo dessa linguagem, porém baseado no mecanismo de co-rotinas simétricas apresentado na seção anterior [1]. Nessa biblioteca, uma co-rotina assimétrica é representada por uma tabela que contém três campos: o estado da co-rotina (`state`), uma referência para seu chamador (`yieldto`) e uma referência para a co-rotina simétrica sobre a qual a co-rotina assimétrica é implementada (`scoro`). A variável auxiliar `current` salva a referência para a co-rotina assimétrica em execução. Essa variável recebe como valor inicial uma tabela que representa o programa principal (isto é, uma tabela que contém no campo `scoro` uma referência para a representação do programa principal como uma co-rotina simétrica). Essa referência permitirá o retorno do controle ao programa principal quando este é o chamador da co-rotina suspensa.

[1] Por questões de simplicidade, omitimos nessa implementação a facilidade de gerência de erros oferecida pelo mecanismo nativo de Lua.

```
asymcoro = {} -- biblioteca de co-rotinas assimétricas

local main = { scoro = symcoro.main }  -- programa principal
local current = main                   -- co-rotina em execução

-- criação de uma co-rotina assimétrica
function asymcoro.create(f)
  local body = function(val)
                 local res = f(val)
                 current.state = "dead"
                 current = current.yieldto
                 symcoro.transfer(current.scoro, res)
               end)

  return { state = "suspended",
           scoro = symcoro.create(body) }
end

-- operação de reativação
function asymcoro.resume(co, val)
  if co.state ~= "suspended" then
    if co.state == "dead" then
      error("é impossivel reativar co-rotina morta")
    else
      error("é impossivel reativar co-rotina ativa")
  end
  co.yieldto = current
  co.state = "running"
  current = co
  return symcoro.transfer(current.scoro, val)
end

-- operação de suspensão
function asymcoro.yield(val)
  if current == main then
    error("é impossivel suspender o programa principal")
  end
  current.state = "suspended"
  current = current.yieldto
  return symcoro.transfer(current.scoro, val)
end

-- função auxiliar wrap
function asymcoro.wrap(f)
  local co = asymcoro.create(f)
  return function(val) return asymcoro.resume(co, val) end
end
```

Figura 4.2: Implementando co-rotinas assimétricas com simétricas

A função `asymcoro.create` cria uma nova co-rotina assimétrica, implementando-a sobre uma co-rotina simétrica. A tabela que representa a nova co-rotina indica que ela foi criada no estado suspenso. Quando a função principal da co-rotina (`f`) retorna, o corpo da co-rotina simétrica modifica o estado da co-rotina para indicar que ela está morta (`dead`), e devolve o controle ao último chamador da co-rotina, transferindo o resultado retornado por `f`.

A função `asymcoro.resume` implementa a operação de (re)ativação de uma co-rotina, e tem dois argumentos: a tabela que representa a co-rotina e um valor opcional a ser transferido. Se a co-rotina designada está morta ou ativa (`running`), a operação é inválida, e sua invocação provoca um erro de execução. Se a co-rotina está suspensa, a operação é válida. Nesse caso, a tabela que representa a co-rotina é atualizada com seu novo estado (`running`) e com a referência para seu chamador (isto é, a co-rotina em execução). Em seguida, o controle é transferido para a co-rotina simétrica correspondente, para a qual é repassado o valor do segundo argumento de `asymcoro.resume`. Na primeira ativação da co-rotina, esse valor é recebido como o argumento de sua função principal. Nas ativações seguintes, esse valor será retornado pela chamada a `asymcoro.yield`.

A função `asymcoro.yield` implementa a operação de suspensão de uma co-rotina assimétrica. Ela atualiza o estado da co-rotina em execução (para `suspended`) e devolve o controle a seu chamador, repassando o valor de seu argumento. Esse valor será então retornado pela função `asymcoro.resume` responsável pela invocação da co-rotina suspensa. Quando o programa principal está em execução, a operação de suspensão é inválida, e sua invocação provoca um erro de execução.

Utilizando as funções básicas `create` e `resume`, podemos acrescentar à biblioteca `asymcoro` a função auxiliar `wrap`, oferecida pela biblioteca de co-rotinas de Lua. Essa função, descrita na Seção 3.3, recebe como parâmetro o corpo de uma co-rotina assimétrica, e retorna como resultado uma função que permite (re)ativar essa co-rotina.

4.2 Continuações one-shot

A introdução do conceito de continuações resultou de diversos esforços para descrever formalmente o comportamento de instruções como *jumps* e

`goto`'s, e de outras construções que alteram a sequência de execução de um programa [74]. Desde sua introdução, esse conceito vem sendo utilizado em diferentes cenários, como, por exemplo, a tradução de programas (transformação CPS) [29] e a definição formal de linguagens de programação [82].

Uma continuação é basicamente uma abstração que representa um contexto de controle como uma função. De uma forma mais intuitiva, uma continuação pode ser descrita como o "resto" de uma computação, ou o que falta ser executado a partir de um determinado ponto de um programa [81].

Algumas linguagens, como Scheme [51] e implementações de ML [40], oferecem continuações como valores de primeira classe. Ao permitir a manipulação explícita de contextos de controle, essas linguagens provêem um poderoso recurso de programação, possibilitando a implementação de diversas estruturas de controle não diretamente oferecidas pela linguagem [27]. Continuações de primeira classe podem ser usadas, por exemplo, para implementar *loops*, geradores, exceções, *backtracking*, *threads* e co-rotinas [88, 22, 41, 42, 21, 81].

Em Scheme, o operador `call/cc` (*call-with-current-continuation*) captura a continuação corrente como um objeto de primeira classe, que pode ser salvo em uma variável, passado como parâmetro ou retornado por uma função. O argumento do operador `call/cc` é uma função cujo parâmetro é a continuação capturada. Se essa função termina sem invocar a continuação, o valor por ela retornado é o resultado da aplicação de `call/cc`. Se, a qualquer momento, a continuação é aplicada a um valor, o contexto da aplicação original de `call/cc` é restaurado, e o valor passado à continuação é retornado como resultado dessa aplicação.

Como uma ilustração bastante simples desse mecanismo, suponhamos que a linguagem Lua oferece uma função `callcc`, cujo comportamento é similar ao do operador `call/cc` de Scheme. Após a execução do trecho

```
x = 1 + callcc(function(k)
                 return 1 - k(3)
               end)
```

a variável `x` terá o valor 4, pois a invocação da continuação `k` durante a execução da função passada a `callcc` provocará o abandono do contexto corrente (que inclui a subtração), e o retorno imediato do valor 3 à continuação da chamada a `callcc`, que inclui a adição do valor retornado ao inteiro 1, e a atribuição do resultado dessa adição à variável `x`.

Mecanismos de continuações de primeira classe, como o implementado por Scheme, permitem que uma mesma continuação seja invocada múltiplas vezes. O exemplo apresentado não utiliza essa facilidade; a continuação

capturada é invocada uma única vez, produzindo um efeito semelhante ao de um comando do tipo *abort*.

Com um outro exemplo, mais complexo, podemos ilustrar o real poder de uma continuação de primeira classe. Suponhamos o seguinte trecho de código:

```
a = callcc(function(k) return k end)
```

Nesse caso, a chamada a `callcc` tem como resultado um valor que representa a própria continuação capturada. Note que o retorno da função passada a `callcc` provoca uma primeira invocação (implícita) dessa continuação. Como o valor retornado é salvo em uma variável, a continuação pode ser reinvocada em qualquer momento posterior da execução do programa, em um comando como `a()`. Essa invocação produz um efeito similar ao de um comando do tipo *goto*, porém muito mais poderoso, pois reconstrói todo o contexto (ou seja, a pilha de execução) da chamada original a `callcc`.

No entanto, em praticamente todas as suas aplicações relevantes, continuações de primeira classe são invocadas uma única vez. Essa constatação motivou então a introdução do conceito de continuações *one-shot* e de seu operador `call/1cc` [10]. Uma continuação *one-shot* pode ser invocada apenas uma vez, seja implicitamente (no retorno da função passada como parâmetro a `call/1cc`) ou explicitamente (pela invocação da continuação criada por `call/1cc`). Essa restrição evita a necessidade de cópias de continuações (isto é, de pilhas de execução), tipicamente necessárias para a implementação de continuações *multi-shot* [12, 45], e permite ganhos significativos de desempenho, especialmente na implementação de *multitasking*.

Na implementação de continuações *one-shot* desenvolvida por Bruggeman et al [10], a pilha de controle de um programa é representada por uma lista encadeada de segmentos de pilha. Cada um desses segmentos é uma pilha de registros de ativação; um registro de controle associado ao segmento contém informações como o tamanho e localização do segmento, e um ponteiro para o segmento anterior. Quando uma continuação *one-shot* é capturada, o segmento de pilha corrente é salvo, e um novo segmento de pilha é alocado para a execução da função passada à `call1/cc`. Quando uma continuação *one-shot* é invocada, o segmento de pilha corrente é descartado, e o segmento correspondente à continuação é restaurado. Para permitir a detecção de uma tentativa de reinvocação dessa continuação, o registro de controle que a representa é "marcado" (o tamanho do segmento é alterado para o valor -1). A Figura 4.3 ilustra esses procedimentos.

A descrição da implementação de continuações *one-shot* revela as similaridades dessa construção com co-rotinas completas simétricas. Assim como

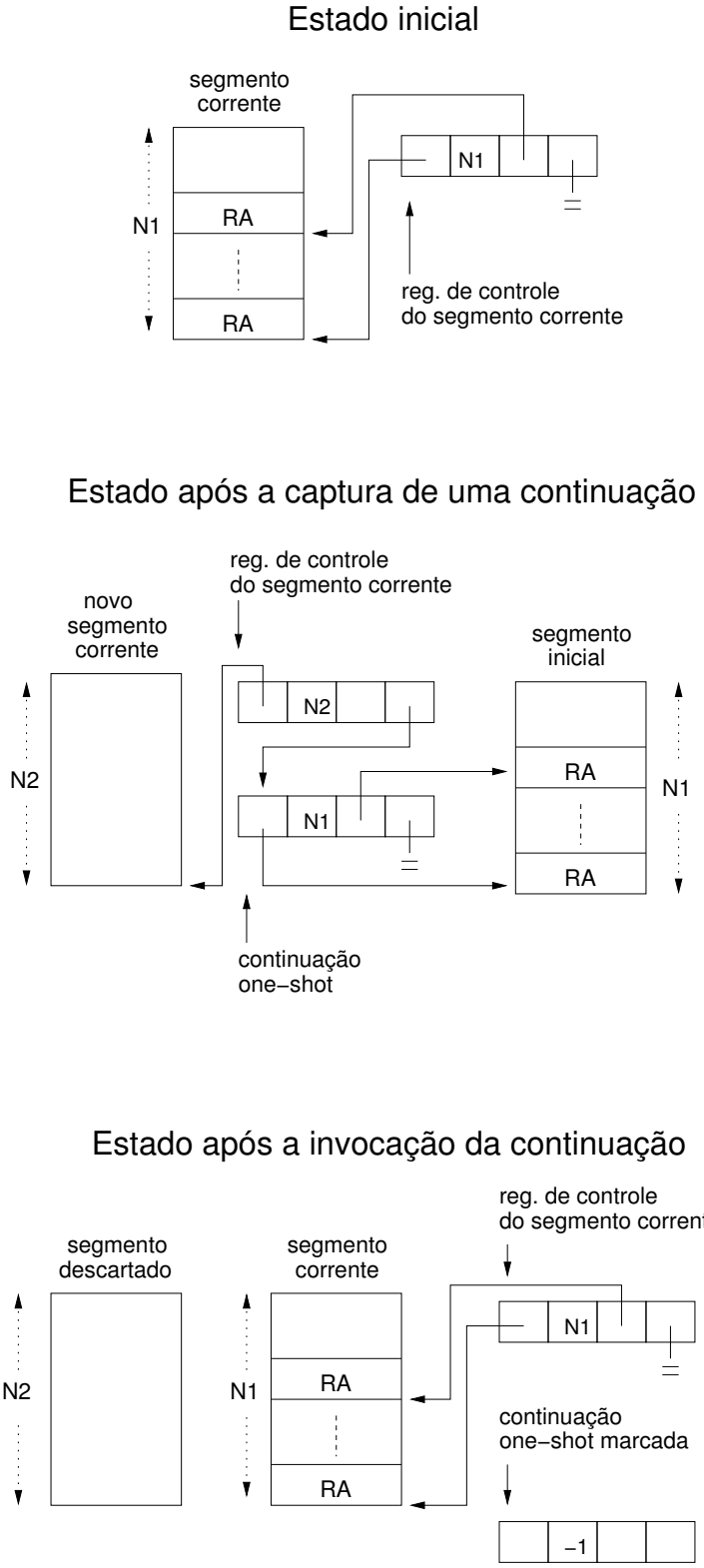

Figura 4.3: Implementação de continuações *one-shot*

```
function call1cc(f)
  -- salva o criador da continuação
  local ccoro = symcoro.current

  -- a invocação de uma continuação transfere
  -- o controle a seu criador
  local cont = function(val)
    if ccoro == nil then error("continuação já invocada") end
    symcoro.transfer(ccoro, val)
  end

  -- para capturar uma continuação, uma nova
  -- co-rotina é criada (e ativada)
  local val
  val = symcoro.transfer(symcoro.create(function()
                           local v = f(cont)
                           -- o retorno da função provoca uma
                           -- invocação implícita da continuação
                           cont(v)
                         end))

  -- quando o controle retorna ao criador,
  -- a continuação foi invocada ("shot")
  -- e deve ser invalidada
  ccoro = nil

  -- o valor passado à continuação é retornado
  -- pela chamada a call1/cc
  return val
end
```

Figura 4.4: Continuações *one-shot* com co-rotinas simétricas

uma continuação, uma co-rotina completa é, basicamente, um segmento de pilha. A captura de uma continuação *one-shot* através da invocação do operador `call/1cc` é semelhante à criação de uma nova co-rotina simétrica, seguida por uma transferência de controle para essa co-rotina. Por sua vez, a invocação de uma continuação *one-shot* restaura um segmento de pilha anterior, da mesma forma que a transferência de controle para uma co-rotina simétrica. A Figura 4.4 mostra como essas similaridades permitem-nos desenvolver uma implementação trivial de continuações *one-shot* em Lua a partir da biblioteca de co-rotinas simétricas descrita na Seção 4.1.

Ao oferecer a função `call1cc`, permitimos que a linguagem Lua dê suporte a qualquer aplicação desenvolvida com continuações *one-shot* de primeira classe. Como esse suporte independe de qualquer modificação nessas aplicações, podemos afirmar que co-rotinas completas têm poder

expressivo equivalente ao de continuações *one-shot*.

4.3 Continuações parciais one-shot

Apesar de seu grande poder expressivo, mecanismos de continuações de primeira classe tradicionais são difíceis de usar e compreender, o que explica, em parte, a ausência desses mecanismos em linguagens *mainstream*. Muito da complexidade envolvida no uso de continuações tradicionais deve-se ao fato que uma continuação representa *todo* o resto de uma computação. A invocação de uma continuação implica assim no abandono de todo um contexto de controle para a restauração de um contexto anterior. À exceção de algumas aplicações simples (como a implementação de uma operação *abort*), esse comportamento complica consideravelmente a estrutura de um programa. A implementação de um gerador, por exemplo, requer a criação e gerência de dois tipos de continuação: uma continuação para salvar o estado corrente do gerador, capturada a cada invocação, e outra para retornar o controle ao usuário desse gerador.

A conveniência de se limitar a extensão de continuações, e, assim, localizar o efeito de suas operações de controle, motivou a introdução do conceito de *continuações parciais* [24, 50] e a proposta de diversas abstrações baseadas nesse conceito [71]. Ao invés de representar todo o resto de uma computação, uma continuação parcial representa apenas uma parte dessa computação (isto é, uma parte de uma continuação). Desse modo, uma continuação parcial é *composta* com um contexto de controle, ao invés de abortá-lo.

Na seção anterior, observamos que a invocação de uma continuação tradicional tem um efeito similar ao de um comando *goto*. A invocação de uma continuação parcial, por sua vez, é semelhante a uma chamada de função, pois estabelece um ponto de retorno claramente identificado: o ponto de composição da continuação parcial com um contexto de controle. Esse tipo de comportamento simplifica o fluxo de controle de um programa e permite implementações mais concisas e compreensíveis de diversas aplicações usuais de continuações, como demonstram Danvy e Filinski [18] e Sitaram [79, 80].

Assim como continuações tradicionais, continuações parciais são tipicamente invocadas uma única vez. Essa característica pode ser observada mesmo em aplicações que utilizam continuações tradicionais *multi-shot*, como geradores [70] e *backtracking* [79]. O limite de uma única invocação

```
function makegenfact(controller)
  local f,i = 1,1 -- inicializa o próximo fatorial

  -- aplica a função recebida ao fatorial corrente
  -- e calcula o próximo fatorial
  while true do
    controller(function(k) return k end)(f)
    i = i +1; f = f * i
  end
end

-- cria o gerador e obtem a primeira subcontinuação
genfact = spawn(makegenfact)

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

Figura 4.5: Gerador de fatoriais com subcontinuações

pode ser então ser aplicado também a continuações parciais, gerando o conceito de continuações parciais *one-shot*.

O mecanismo de *subcontinuações* [46] é um dos exemplos de implementação do conceito de continuações parciais. Uma subcontinuação representa o resto de uma computação parcial independente (uma subcomputação), estabelecida pelo operador de controle `spawn`. Esse operador recebe como parâmetro a função que representa a subcomputação, e a invoca passando como parâmetro um *controlador*. Se esse controlador não é invocado, a subcomputação eventualmente termina, e o resultado da aplicação de `spawn` é o valor retornado pela função. Se a função invoca o controlador, a continuação da subcomputação é capturada e abortada; essa continuação é uma *subcontinuação* (uma continuação parcial) cuja extensão é limitada pela raiz da subcomputação. A função passada ao controlador é então invocada, recebendo como parâmetro a subcontinuação capturada. Um controlador é válido apenas quando a raiz da subcomputação correspondente está contida na continuação do programa. Dessa forma, somente quando uma subcontinuação é invocada, e, portanto, composta com a continuação corrente, o controlador pode ser utilizado.

Para ilustrar o mecanismo de subcontinuações, vamos supor que a linguagem Lua oferece uma função com comportamento similar ao do operador `spawn`. No exemplo da Figura 4.5, a função `makegenfact` é uma subcomputação que implementa um gerador de números fatoriais. A

chamada à função `spawn` ativa a subcomputação, que inicia seu estado local e invoca o controlador. Essa invocação interrompe a execução do gerador e captura sua continuação (uma subcontinuação), que é passada a uma função que retorna a subcontinuação como resultado da aplicação de `spawn`. A continuação do gerador, quando invocada, recebe uma função (`print`, no nosso exemplo), e a aplica ao valor corrente do fatorial. Em seguida, a continuação calcula o valor do próximo fatorial e reinvoca o controlador. Essa invocação captura uma nova subcontinuação, que reflete o novo estado do gerador.

Podemos perceber que os comportamentos de subcontinuações *one-shot* e de co-rotinas completas assimétricas têm várias semelhanças. Co-rotinas completas podem ser vistas como subcomputações independentes. A invocação do controlador de uma subcomputação é similar a uma operação de suspensão de uma co-rotina assimétrica, pois retorna o controle ao último ponto de reativação da subcomputação. Por sua vez, a invocação de uma subcontinuação *one-shot* corresponde à reativação de uma co-rotina assimétrica.

Uma diferença relevante entre o mecanismo de subcontinuações e outros tipos de continuações parciais é o fato de que uma subcontinuação, assim como uma co-rotina *completa*, é composta com o seu ponto de invocação (ou seja, com o contexto de controle onde a subcontinuação é invocada). Em outros mecanismos, a continuação parcial é composta com o contexto de criação da subcomputação correspondente, o que impõe uma restrição semelhante à observada em mecanismos de co-rotinas *confinadas*, descritos na Seção 2.2. Essa restrição, como vimos, implica em uma redução de poder expressivo, o que pode ser observado alguns exemplos de uso desse tipo de continuações parciais confinadas. Um desses exemplos é a solução para o problema *same-fringe* (a comparação entre duas árvores binárias para determinar se elas possuem a mesma sequência de folhas [44]). Nas soluções implementadas pelos mecanismos desenvolvidos por Queinnec e Serpette [70] e Sitaram [79], a obtenção de cada folha de uma árvore requer a criação de uma nova subcomputação.

A principal diferença entre subcontinuações *one-shot* e co-rotinas completas assimétricas é o fato de uma subcontinuação não ser restrita à subcomputação mais interna. O mecanismo de subcontinuações permite que uma subcomputação "aninhada" invoque o controlador de qualquer subcomputação mais externa. Nesse caso, a subcontinuação capturada estende-se do ponto de invocação do controlador na computação aninhada até a raiz da subcomputação externa, podendo incluir diversas outras

subcomputações nesse caminho. Esse tipo de comportamento é oferecido por variações de alguns mecanismos de continuações parciais que utilizam *marcas* [70] ou *tags* [79] para especificar a extensão da continuação parcial a ser capturada. Como veremos a seguir, essa facilidade pode ser também implementada com co-rotinas completas assimétricas.

A Figura 4.6 mostra uma implementação do operador `spawn` baseada no mecanismo de co-rotinas assimétricas completas de Lua. A função `spawn` cria uma co-rotina Lua para representar uma subcomputação. O corpo dessa co-rotina invoca o argumento de `spawn` (uma função `f`), passando como parâmetro uma função definida no escopo local; essa função (`controller`) implementa o controlador da subcomputação. A variável `validC` indica se a invocação do controlador é válida; seu valor é `true` apenas quando a co-rotina correspondente à subcomputação está ativa.

A função `subK` é responsável por iniciar e restaurar uma subcomputação. Ela tem dois argumentos: a identificação da subcontinuação invocada e o valor a ser passado à subcontinuação. A identificação de uma subcontinuação é usada para testar se sua invocação é válida, isto é, se a subcontinuação já foi invocada anteriormente. Como um controlador é válido apenas quando a subcomputação correspondente está ativa, uma nova subcontinuação somente pode ser capturada após a invocação da subcontinuação anterior. Dessa forma, podemos identificar subcontinuações através de uma sequência crescente de valores numéricos, mantendo em uma variável local (`validK`) a identificação da próxima subcontinuação válida.

Uma subcomputação é restaurada através da reativação da co-rotina correspondente (linha 25). Quando a subcomputação termina sem invocar o controlador, a co-rotina retorna dois valores: o valor de retorno de `f` e `nil` (linha 7). Nesse caso, a função `subK` termina, repassando o retorno de `f` a seu chamador (linha 29).

Quando um controlador é invocado, a co-rotina em execução é suspensa, retornando a seu chamador a função a ser aplicada à subcontinuação (`fc`) e uma referência para o controlador utilizado (linha 15). O uso dessa referência permite-nos expressar uma subcontinuação composta por um número arbitrário de subcomputações. Se a referência retornada pela co-rotina corresponde ao controlador local, a função passada ao controlador é aplicada à subcontinuação (linha 36). Se o controlador invocado corresponde a uma subcomputação externa, a função `coroutine.yield` é chamada para que o controle seja encaminhado a essa subcomputação (linha 41); esse processo é repetido até que a raiz da subcomputação "alvo" seja alcançada, incluindo na subcontinuação capturada todas as subcomputações

```
 1 function spawn(f)
 2   local controller, validC, subK
 3   local validK = 0
 4
 5   -- subcomputação representada por uma co-rotina assimétrica
 6   local subc = coroutine.wrap(function()
 7                                 return f(controller), nil
 8                               end)
 9
10   -- implementação do controlador
11   function controller(fc)
12     if not validC then error("controlador inválido") end
13
14     -- suspende a subcomputação
15     val = coroutine.yield(fc, controller)
16     return val
17   end
18
19   -- inicia/reativa uma subcomputação
20   function subK(k,v)
21     if k ~= validK then error("subcontinuação já invocada") end
22
23     -- invoca uma subcontinuação
24     validC = true
25     local ret, c = subc(v)
26     validC = false
27
28     -- a subcomputação terminou ?
29     if c == nil then return ret
30
31     -- se o controlador local foi invocado,
32     --  aplica a função à subcontinuação
33     elseif c == controller then
34       validK = validK + 1
35       local k = validK
36       return ret(function(v) return subK(k,v) end)
37
38     -- o controlador invocado corresponde
39     -- a uma subcomputação externa
40     else
41       val = coroutine.yield(ret, c)
42       return subK(validK, val)
43     end
44   end
45
46   -- inicia a subcomputação
47   return subK(validK)
48 end
```

Figura 4.6: Subcontinuações *one-shot* com co-rotinas assimétricas

suspensas. Simetricamente, a invocação dessa subcontinuação provocará a reativação de todas as co-rotinas suspensas (linha 42), até que o ponto original de invocação do controlador seja alcançado.

Quando uma subcontinuação é invocada, a função `controller` que suspendeu a co-rotina correspondente retorna a seu chamador. O resultado da chamada ao controlador é o valor passado à continuação, retornado por `coroutine.yield` (linha 15).

A implementação do operador `spawn` baseada em um mecanismo de co-rotinas simétricas, apresentada na Figura 4.7, elimina a necessidade de suspensões e reativações sucessivas de co-rotinas aninhadas. O uso de co-rotinas simétricas permite uma transferência direta do controle à co-rotina correspondente à raiz de uma subcomputação quando o controlador correspondente é invocado. Para que essa transferência possa ser realizada, a variável `current`, definida no escopo da subcomputação, armazena a referência para a última co-rotina que restaurou essa subcomputação.

Da mesma forma, a co-rotina suspensa pela invocação de um controlador pode ser reativada através de uma transferência de controle direta quando a subcontinuação correspondente é invocada. Para que isso seja possível, o valor transferido por um controlador à sua raiz é uma tabela que armazena no campo `val` a função a ser aplicada à subcontinuação (`fc`), no campo `ended` o valor `false` (que indica que a subcomputação não terminou) e no campo `sc` uma referência para a co-rotina suspensa. Essa referência é utilizada na definição da função que representa a subcontinuação, e a função `subK`, passa a receber também essa referência como parâmetro, utilizando-a para transferir o controle diretamente à co-rotina apropriada.

Quando uma subcomputação termina sem invocar o controlador, o controle é transferido à última co-rotina que restaurou a subcomputação. O valor retornado a essa co-rotina é uma tabela que armazena no campo `val` o resultado da chamada à função que representa a subcomputação e no campo `ended` o valor `true`, que indica que a subcomputação terminou.

4.4 Questões de conveniência e eficiência

Nas seções anteriores, nós demonstramos que co-rotinas completas simétricas, co-rotinas completas assimétricas e e continuações *one-shot* são construções que oferecem o mesmo poder expressivo. Entretanto, conforme discutiremos a seguir, essas construções nem sempre são equivalentes com respeito à questões de conveniência e, em alguns casos, também de eficiência.

```
function spawn(f)
  local current, controller, validC, subK
  local validK = 0

  -- subcomputação representada por uma co-rotina simétrica
  local subc = symcoro.create(
                function()
                  local ret = { val = f(controller),
                                ended = true }
                  symcoro.transfer(current, ret)
                end)

  -- implementação do controlador
  function controller(fc)
    if not validC then error("controlador invalido") end

    -- suspende a computação
    local ret = { val = fc, sc = symcoro.current,
                  ended = false }
    ret = symcoro.transfer(current, ret)
    return ret
  end

  -- inicia/reativa uma subcomputação
  function subK(k,sc,v)
    if k ~= validK then error("subcontinuação já invocada") end

    -- invoca subcontinuação da subcomputação especificada (sc)
    validC = true
    current = symcoro.current    -- salva o chamador
    local ret = symcoro.transfer(sc, v)
    validC = false

    -- a subcomputação terminou?
    if ret.ended == true then return ret.val end

    -- aplica função à subcontinuação
    validK = validK + 1
    local k = validK
    return ret.val(function(v) return subK(k,ret.sc,v) end)
  end

  -- inicia a subcomputação
  return subK(validK, subc)
end
```

Figura 4.7: Subcontinuações *one-shot* com co-rotinas simétricas

4.4.1 Co-rotinas assimétricas versus co-rotinas simétricas

No capítulo 2, ao destacar as diferenças entre mecanismos de corotinas simétricas e assimétricas, argumentamos que co-rotinas assimétricas constituem uma construção mais conveniente para a implementação de diferentes estruturas de controle. A semelhança entre o sequenciamento de controle de co-rotinas assimétricas e funções convencionais simplifica a compreensão de implementações baseadas em co-rotinas assimétricas, e permite o desenvolvimento de programas mais estruturados. Por sua vez, a gerência do fluxo de controle de um programa que utilize mesmo um número moderado de co-rotinas simétricas requer um esforço considerável de um programador.

Argumentos semelhantes, como vimos, são utilizados na comparação entre mecanismos de continuações tradicionais e parciais, e constituem uma das principais motivações para as propostas de abstrações baseadas no conceito de continuações parciais. De fato, as similaridades entre continuações tradicionais *one-shot* e co-rotinas simétricas, e entre continuações parciais *one-shot* e co-rotinas assimétricas, explicam por que as mesmas vantagens proporcionadas por continuações parciais são oferecidas por co-rotinas assimétricas, tanto em relação a co-rotinas simétricas como em relação a continuações tradicionais.

Além da maior simplicidade para a implementação de diversos tipos de comportamento de controle, duas outras questões relevantes contribuem para a maior conveniência de mecanismos de co-rotinas assimétricas em relação a co-rotinas simétricas. A primeira questão diz respeito ao tratamento de erros que ocorrem durante a execução de uma co-rotina. Para ilustrar essa questão, consideremos, em primeiro lugar, um programa estruturado como um conjunto de co-rotinas simétricas. Se a execução de uma dessas co-rotinas provoca algum tipo de erro, é virtualmente impossível prosseguir a execução do programa, mesmo que a linguagem ofereça facilidades para a captura de erros. Como o sequenciamento de controle é distribuído entre as co-rotinas, não se pode determinar, a princípio, a que co-rotina o controle deve ser devolvido. Se o mesmo programa utiliza co-rotinas assimétricas, a captura de um erro provocado pela execução de uma co-rotina pode ser sinalizada ao chamador dessa co-rotina, ao qual o controle é naturalmente transferido. Essa facilidade é oferecida, por exemplo, pelo mecanismo de co-rotinas assimétricas da linguagem Lua, descrito na Seção 3.3. A Figura 4.8 ilustra essas duas situações.

A segunda questão diz repeito à facilidade de integração de uma linguagem que oferece um mecanismo de co-rotinas com outras linguagens

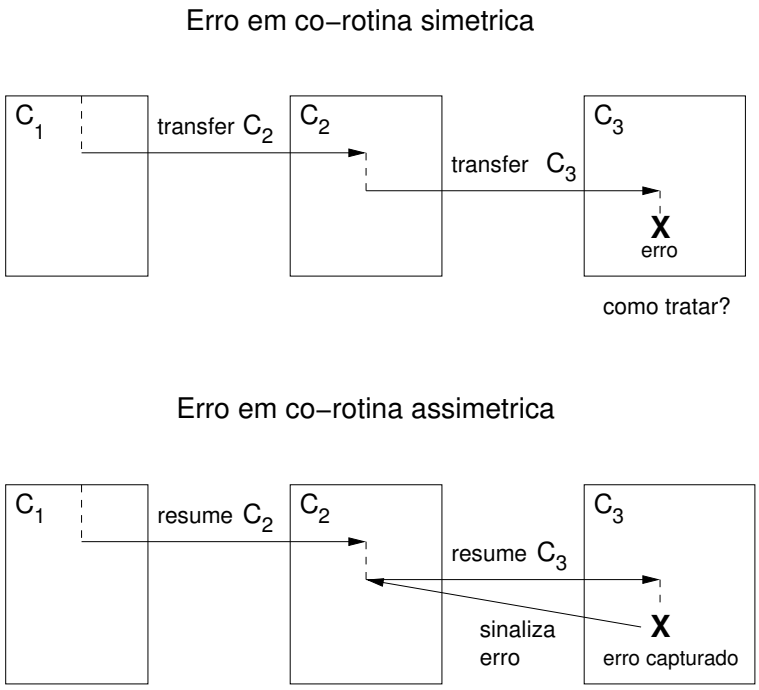

Figura 4.8: Erros de execução em co-rotinas simétricas e assimétricas

— por exemplo, a integração de uma linguagem de *extensão*, como Lua, com sua linguagem hospedeira. Em Lua, um código escrito em C pode chamar um código Lua, e vice-versa. Dessa forma, uma aplicação Lua pode ter em sua pilha de controle uma cadeia de chamadas de funções onde essas duas linguagens são livremente intercaladas. A implementação de um mecanismo de co-rotinas simétricas nesse cenário impõe a preservação do estado "C" quando uma co-rotina Lua é suspensa, pois a transferência de controle entre co-rotinas simétricas implica na substituição de toda a pilha de controle do programa. Isso somente é possível com o suporte de um mecanismo de co-rotinas também para a linguagem hospedeira. Contudo, implementações de co-rotinas para C não são disponíveis usualmente. Além disso, uma implementação portável de co-rotinas para C é inviável.

Por outro lado, o suporte às co-rotinas assimétricas de Lua não requer o uso de um mecanismo de co-rotinas para a linguagem hospedeira, pois uma co-rotina assimétrica envolve apenas uma porção da pilha de controle do programa, como ilustra a Figura 4.9. Dessa forma, é necessário apenas impor como restrição que uma co-rotina Lua somente possa ser suspensa se não há uma chamada ativa a uma função C no segmento de pilha que representa a co-rotina.

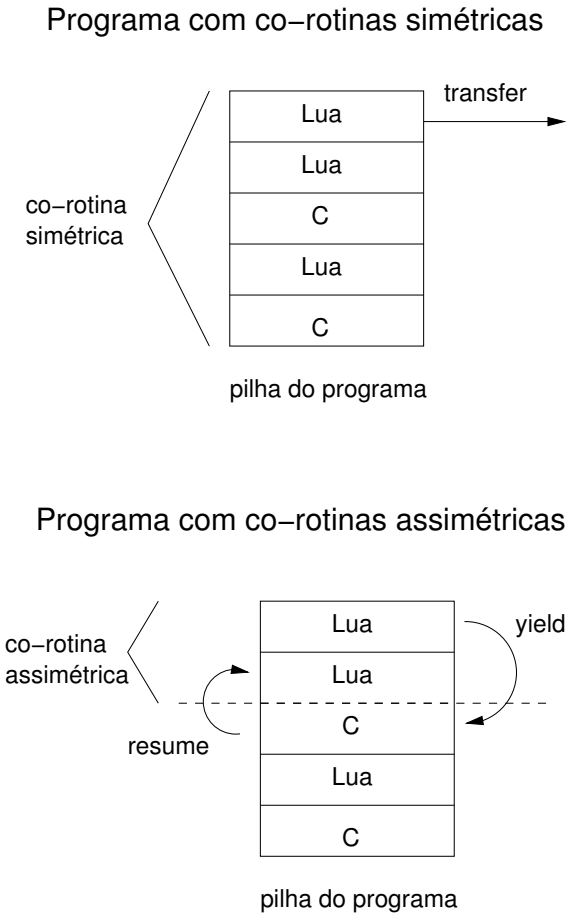

Figura 4.9: Pilha de um programa com co-rotinas simétricas e assimétricas

4.4.2 Continuaçoes one-shot versus co-rotinas completas

Haynes, Friedman e Wand [41] mostraram que continuações de primeira classe tradicionais podem ser usadas para implementar co-rotinas. Sitaram [80] mostrou que co-rotinas podem ser expressas também em termos de continuações parciais Neste capítulo nós mostramos que co-rotinas completas podem ser usadas para expressar tanto continuações tradicionais como continuações parciais *one-shot*. Contudo, a implementação de continuações com co-rotinas e a operação inversa não são equivalentes em termos de eficiência.

Numa implementação simples de continuações tradicionais *one-shot*, como a descrita na Seção 4.2, a criação de uma continuação envolve a alocação de um novo segmento de pilha. Na invocação de uma continuação, o segmento corrente é descartado, e o segmento correspondente à continuação é restaurado. A gerência da pilha de execução de uma aplicação é realizada através da alocação e manipulação de estruturas de controle associadas aos diversos segmentos, ou continuações. Uma implementação trivial

de co-rotinas completas envolve também a alocação de segmentos de pilha, associados a informações de controle. As operações de ativação e suspensão de co-rotinas têm um custo apenas um pouco maior que chamadas convencionais de funções.

Em nossa implementação de continuações tradicionais *one-shot*, mostramos que a criação de uma única co-rotina é suficiente para representar uma continuação, e que as operações de transferência de controle entre co-rotinas permitem simular a captura e invocação de continuações. A comparação dessa implementação com a implementação direta de continuações *one-shot* permite-nos argumentar que uma linguagem que oferece co-rotinas completas pode prover um mecanismo de continuações tradicionais *one-shot* tão eficiente quanto uma implementação direta dessa abstração.

Hieb, Dybvig e Anderson [46] descrevem uma possível implementação de subcontinuações baseada na representação da pilha de controle de um programa através de uma pilha de "segmentos de pilha rotulados". A ativação de uma nova subcomputação (pela invocação do operador `spawn`) resulta então na adição de um novo segmento ao topo da pilha do programa; a esse novo segmento é atribuído um rótulo que permite associá-lo ao controlador correspondente. Quando um controlador é invocado, todos os segmentos superpostos ao segmento associado ao rótulo do controlador são removidos da pilha de execução, e convertidos em uma subcontinuação. Quando a subcontinuação é invocada, esses segmentos são recolocados no topo da pilha de execução.

Em nossas implementações de subcontinuações *one-shot* com co-rotinas assimétricas e simétricas, o custo relativo à ativação de uma subcomputação (isto é, a criação e ativação de uma co-rotina), pode ser considerado equivalente ao custo da implementação direta proposta. Quando uma subcontinuação envolve uma única subcomputação (o caso mais usual), as duas implementações com co-rotinas executam a captura e invocação da subcontinuação com eficiência equivalente à da implementação direta. No caso mais complicado, quando uma subcontinuação envolve diversas subcomputações aninhadas, a implementação com co-rotinas simétricas é tão eficiente quanto a implementação direta. As reativações sucessivas de subcomputações na implementação com co-rotinas assimétricas impõem um certo *overhead*, porém com um custo não muito maior que uma sequência de chamadas de funções.

Por outro lado, as implementações de co-rotinas com continuações são bem menos eficientes que implementações diretas de co-rotinas. Essas implementações, tanto com continuações tradicionais como com continuações

parciais, requerem a captura de uma nova continuação quando uma co-rotina é suspensa. Esse maior consumo de processamento — e, em implementações mais simples, também de memória — representa uma desvantagem do uso de continuações em relação ao uso de co-rotinas, tanto para a implementação de *multitasking* como para a implementação de geradores, que devem manter seu estado entre invocações sucessivas. Além disso, o uso de co-rotinas simplifica a implementação desses comportamentos, evitando a captura "explícita" de continuações. Essa maior simplicidade, adequada especialmente a um contexto de linguagens procedurais, é evidenciada nos exemplos de programação apresentados nos próximos capítulos.

Uma implementação de subcontinuações *one-shot* em termos de *threads* foi proposta por Kumar, Bruggeman e Dybvig [53]. Sua idéia básica é de certa forma semelhante à utilizada em nossas implementações com co-rotinas, especialmente a implementação com co-rotinas simétricas. Uma subcomputação é representada por uma *thread* "filha", criada na invocação do operador `spawn`. Quando a subcomputação é ativada, a execução da *thread* "mãe" é suspensa, e sua reativação é condicionada à sinalização de uma condição (`done`) associada ao controlador da subcomputação. O controlador é implementado por uma função que sinaliza a condição `done`, cria uma nova condição (`continue`) e suspende a execução da *thread* corrente, que passa a aguardar essa nova condição. Quando uma subcontinuação é invocada, a *thread* correspondente é reativada pela sinalização de sua condição `continue`, e seu chamador é suspenso, com sua reativação condicionada por uma nova sinalização da condição `done` associada ao controlador da subcomputação [2].

Além do uso de condições para implementar a suspensão e reativação de *threads* (que, diferentemente de co-rotinas, não podem transferir o controle explicitamente) a implementação de subcontinuações com *threads* impõe a necessidade de um mecanismo de exclusão mútua para evitar que que a *thread* que executa a subcomputação sinalize a condição `done` antes que a *thread* que deve aguardar essa condição esteja efetivamente suspensa.

A implementação de subcontinuações com *threads*, assim como a implementação com co-rotinas simétricas, não requer suspensões e reativações sucessivas de subcomputações aninhadas. Contudo, o uso de *threads* introduz uma maior complexidade e *overhead*, pela necessidade de mecanismos de sincronização. Além de mais eficiente, a implementação de subcontinuações *one-shot* com co-rotinas simétricas é bem mais simples, e menos suscetível a erros.

[2] Essa é uma descrição simplificada da implementação apresentada em [53]. Consideramos apenas os requisitos para uma implementação de subcontinuações não concorrentes.