2
Um sistema de classificação para co-rotinas

A literatura descreve o conceito de co-rotinas de forma bastante genérica; a definição consensualmente adotada baseia-se apenas na capacidade de uma co-rotina manter seu estado local entre chamadas sucessivas. Entretanto, as implementações de co-rotinas diferem bastante, tanto com respeito ao seu poder expressivo como quanto à sua conveniência, ou seja, a complexidade envolvida em seu uso e compreensão.

Analisando as diversas implementações de co-rotinas, podemos identificar três características responsáveis por essas diferenças:

- as operações de transferência de controle oferecidas;
- se co-rotinas são valores de primeira classe;
- se co-rotinas são construções *stackful*.

Neste capítulo propomos um sistema de classificação para co-rotinas baseado nessas três características, discutindo como cada uma delas influencia a conveniência e expressividade de um mecanismo de co-rotinas. Nessa discussão, e no restante deste trabalho, a noção de expressividade de uma construção está relacionada à sua capacidade em prover um dado comportamento de controle sem que seja necessário alterar a estrutura global de um programa. Embora definida informalmente, essa noção de expressividade é compatível com a formalização desenvolvida por Felleisen [25].

2.1
Co-rotinas simétricas e assimétricas

A distinção entre os conceitos de co-rotinas *simétricas* e *assimétricas* diz respeito, basicamente, às operações de transferência de controle oferecidas pelo mecanismo de co-rotinas. Apesar dessa distinção ser usualmente reconhecida, suas consequências são, muitas vezes, erroneamente entendidas.

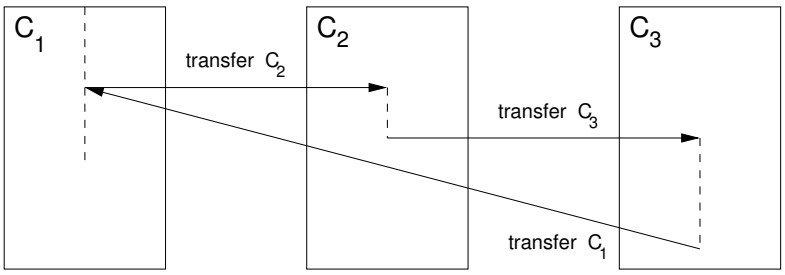

Figura 2.1: Transferência de controle com co-rotinas simétricas

O conceito de co-rotinas simétricas representa a abstração de controle originalmente proposta por Conway [14]. Mecanismos que implementam esse conceito provêem, tipicamente, uma única operação de transferência de controle, cujo efeito é a (re)ativação da co-rotina explicitamente nomeada. Como co-rotinas simétricas passam o controle arbitrariamente entre si, é comum descrevê-las como operando em um mesmo nível hierárquico [69]. A Figura 2.1 ilustra o uso da operação de transferência de controle implementada por um mecanismo de co-rotinas simétricas. Nessa ilustração, C_1, C_2 e C_3 denotam co-rotinas simétricas e a operação de transferência de controle é denominada *transfer*.

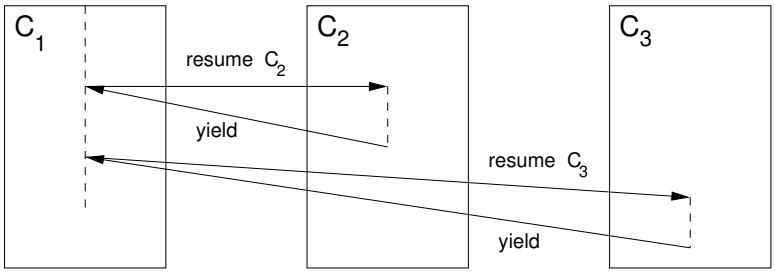

Figura 2.2: Transferência de controle com co-rotinas assimétricas

O conceito de co-rotinas assimétricas — mais comumente denominadas co-rotinas *semi-simétricas* ou *semi co-rotinas* — foi introduzido pela linguagem Simula [17]. Mecanismos de co-rotinas assimétricas provêem duas operações de transferência de controle: uma operação para invocar uma co-rotina e outra para suspendê-la. Como esta última operação retorna o controle implicitamente para o chamador da co-rotina, co-rotinas assimétricas são usualmente descritas como subordinadas a seus chamadores, comportando-se, de certa forma, como subrotinas ou funções [62]. A Figura 2.2 ilustra a disciplina de controle implementada por um mecanismo de co-rotinas assimétricas. Nessa ilustração, C_1, C_2 e C_3 denotam co-rotinas assimétricas, a operação de invocação de uma co-rotina é denominada *re-*

sume e a operação de suspensão é denominada *yield*.

De forma geral, observa-se que o uso pretendido para o mecanismo de co-rotinas determina a opção por um ou ambos os tipos de transferência de controle. Mecanismos de co-rotinas desenvolvidos para dar suporte a programação concorrente geralmente provêem co-rotinas simétricas para representar unidades de execução independentes. As co-rotinas simétricas de Modula-2 [90] e o mecanismo de *fibers* do Windows [75] são exemplos desse tipo de uso. Mecanismos de co-rotinas voltados para a implementação de *geradores* — construções que produzem sequências de valores — provêem, usualmente, co-rotinas assimétricas. CLU [60], Sather [66], Icon [36] e Python [63] são exemplos de linguagens que oferecem esse tipo de construção. Por sua vez, mecanismos de co-rotinas de propósito geral, como os implementados pelas linguagens Simula e BCPL [64] e, mais recentemente, por algumas bibliotecas desenvolvidas para linguagens como C e C++ [85, 43], oferecem, tipicamente, os dois tipos de transferência de controle. Uma exceção é a linguagem BETA [61], uma sucessora de Simula, que implementa um mecanismo genérico de co-rotinas que oferece apenas co-rotinas assimétricas. BETA, contudo, implementa uma construção diferente para o suporte a programação concorrente.

A ausência de uma definição precisa e completa para o conceito de co-rotinas levou ao uso de descrições informais das implementações disponíveis como referências para esse mecanismo e, especialmente, à adoção da implementação de Simula, a mais conhecida (e também a mais complexa), como a principal referência para um mecanismo de co-rotinas de propósito geral. Como Simula oferece os dois tipos de transferência de controle, estabeleceu-se uma percepção de que co-rotinas simétricas e assimétricas não tem poder equivalente, e que mecanismos genéricos devem, portanto, prover as duas construções [69]. O próprio mecanismo proposto na tese de Marlin [62] (a principal referência para o conceito de co-rotinas) é baseado em Simula; apesar de menos complexo, esse mecanismo também oferece co-rotinas simétricas (denominadas apenas co-rotinas) e assimétricas (denominadas geradores).

Contudo, como mostraremos no Capítulo 4, é trivial implementar co-rotinas simétricas em termos de assimétricas — e vice-versa — desde que ambas sejam construções *stackful* de primeira classe (características que discutiremos a seguir). Dessa forma, qualquer comportamento implementado com um desses mecanismos pode ser provido pelo outro, sem necessidade de qualquer alteração em um programa. Um mecanismo genérico de co-rotinas pode prover assim apenas um dos tipos de transferência de controle; ofere-

cer os dois somente complica a semântica do mecanismo. Em Simula, por exemplo, a união de mecanismos de transferência de controle simétricos e assimétricos introduziu uma grande dificuldade para a compreensão do comportamento de uma co-rotina, que, dependendo de como é invocada, ora pode comportar-se como uma co-rotina simétrica, ora como assimétrica. Essa dificuldade é refletida nos esforços para descrever a semântica das co-rotinas de Simula; muitas dessas descrições são incompletas ou mesmo inconsistentes [62].

Apesar de equivalentes em termos de poder expressivo, co-rotinas simétricas e assimétricas não são equivalentes com respeito à sua conveniência, isto é, à facilidade oferecida para a implementação de diferentes comportamentos. Para a implementação de geradores, por exemplo, a semelhança entre co-rotinas assimétricas e funções é bastante apropriada: um gerador não precisa conhecer seu usuário, pois o controle será implicitamente retornado para ele. O uso de uma co-rotina simétrica nesse contexto é bem menos adequado, pois requer uma gerência explícita da transferência de controle entre o gerador e seu usuário.

Ainda que usualmente consideradas apenas para a implementação de geradores e estruturas similares, co-rotinas assimétricas são convenientes também para o suporte à programação concorrente. Ambientes de concorrência tipicamente utilizam um *dispatcher*, responsável pela política de escalonamento de tarefas. Quando uma tarefa é suspensa, o controle deve ser retornado ao *dispatcher*, que determina a próxima tarefa a ser ativada. A implementação dessa estrutura com base em um mecanismo de co-rotinas assimétricas é bastante natural: o *dispatcher* é apenas um *loop* responsável pela ativação das tarefas, e as tarefas são totalmente independentes umas das outras. A implementação baseada em co-rotinas simétricas é mais complicada; nesse caso, a responsabilidade de escalonamento ou é distribuída pelas diversas tarefas concorrentes ou é atribuída a uma co-rotina específica, que deve ser conhecida por todas as demais. No contexto de programação concorrente, a associação bastante usual do conceito de co-rotinas a mecanismos de transferência de controle simétricos muito contribuiu para fundamentar argumentos de que co-rotinas são um mecanismo inadequado, de difícil compreensão [56].

2.2 Co-rotinas de primeira classe e confinadas

Uma característica que influencia consideravelmente o poder expressivo de um mecanismo de co-rotinas é o fato de co-rotinas serem ou não valores de primeira classe. Em outras palavras, se co-rotinas podem ser livremente manipuladas e, portanto, invocadas em qualquer ordem e em qualquer ponto de um programa.

Em algumas implementações, co-rotinas são confinadas em contextos determinados e não podem ser diretamente manipuladas pelo programador. Um exemplo dessa forma restrita de co-rotina é a abstração de um iterador, originalmente proposta e implementada pelos projetistas da linguagem CLU [59, 60]. O objetivo dessa abstração é permitir que uma estrutura de dados seja percorrida independentemente de sua representação interna; a cada invocação, um iterador retorna o próximo item da estrutura de dados correspondente.

Como um iterador CLU preserva seu estado entre chamadas sucessivas, seus criadores o descreveram como uma co-rotina (essa característica corresponde, de fato, à definição genérica do conceito de co-rotina). Contudo, um iterador CLU é confinado em uma estrutura de iteração específica (um comando `for`). Essa estrutura admite o uso de um único iterador, que é invocado implicitamente a cada passo da iteração. Essas restrições garantem que iteradores ativos sejam sempre aninhados, e permitem, assim, uma implementação simples e eficiente, que utiliza uma única pilha de controle [4]. Contudo, essas restrições limitam também o poder dos iteradores, não permitindo, por exemplo, que duas ou mais estruturas de dados sejam percorridas em paralelo.

Os iteradores de Sather [66], inspirados em CLU, são também confinados em uma estrutura de iteração (um comando `loop`). Apesar do número de iteradores invocados em um *loop* não ser limitado, a invocação de cada iterador é restrita a um único ponto de chamada, e se um dos iteradores termina, o *loop* é encerrado. Em Sather, a manipulação simultânea de diversas estruturas de dados é possível, mas a implementação de manipulações assíncronas (onde a sequência de invocação de iteradores é arbitrária) não tem solução trivial. Esse tipo de manipulação é necessário, por exemplo, para combinar (*merge*) duas ou mais estruturas de dados.

A linguagem Icon implementa um interessante paradigma, denominado *goal-directed evaluation* (avaliação direcionada por meta) [35]. Para implementar esse paradigma, Icon utiliza um mecanismo de *backtracking* e

uma forma restrita de co-rotinas, denominada geradores (*generators*) — expressões capazes de gerar uma sequência de valores, se isto é necessário para avaliar, com sucesso, a expressão onde estão contidas. Além de prover uma coleção de geradores pré-definidos, Icon também permite que um programador defina novos geradores, implementados por procedimentos que, ao invés de retornar, suspendem sua execução através do uso da expressão `suspend`. Essa expressão retorna o valor de seu argumento, mantendo, porém, o estado local do procedimento. Se o valor retornado não satisfaz a meta pretendida — o sucesso na avaliação da expressão que contém a chamada ao procedimento — o procedimento é reinvocado, e sua execução é retomada no ponto de suspensão. Apesar de não serem restritos a uma construção específica, como os iteradores de CLU e Sather, os geradores de Icon são confinados em uma expressão e seu uso é limitado à localização dessa expressão e à sequência de avaliação de um programa.

Co-rotinas confinadas, como os iteradores de CLU e Sather e os geradores de Icon, não permitem a implementação de comportamentos onde a sequência de invocações deve ser controlada explicitamente, ou seja, quando é necessário que uma co-rotina possa ser invocada a qualquer momento, e em qualquer lugar do programa. Esse controle explícito é necessário, como vimos, para implementar manipulações assíncronas de estruturas de dados, e, de forma geral, para prover suporte à implementação de estruturas de controle definidas pelo usuário, e, especialmente, *multitasking*.

O controle explícito sobre co-rotinas somente é possível quando elas são valores de primeira classe, livremente manipuladas por um programador. Essa característica é oferecida, por exemplo, pelas *co-expressions* de Icon. Uma *co-expression* é um objeto que "captura" uma expressão, e pode ser atribuído a uma variável, passado como argumento e retornado como resultado de um procedimento. Dessa forma, a expressão capturada pode ser (re)invocada em qualquer lugar do programa.

Co-rotinas de primeira classe são também oferecidas pelos mecanismos genéricos implementados por Simula, BETA e BCPL, e por mecanismos voltados para o suporte a programação concorrente como as co-rotinas de Modula-2 e as *fibers* do Windows.

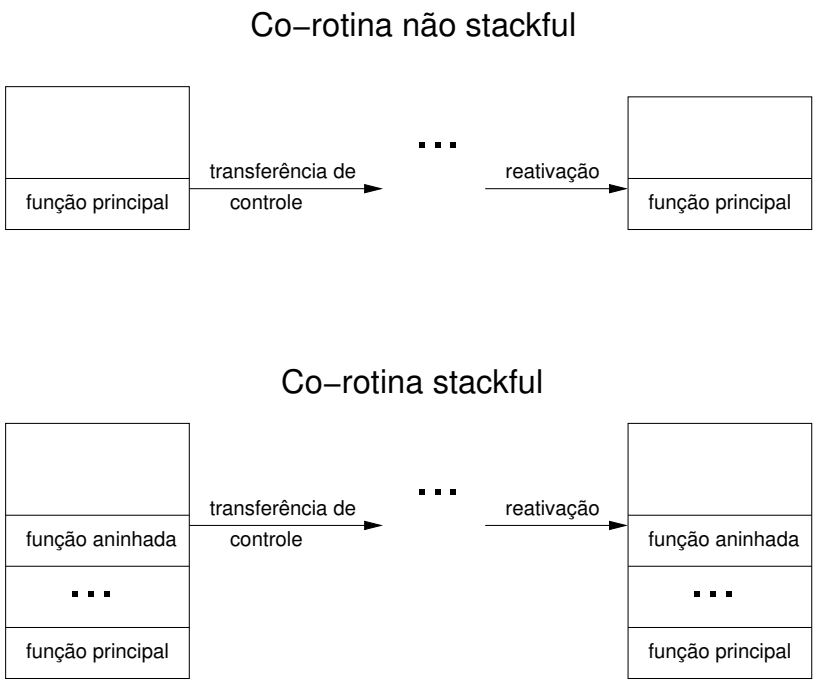

Figura 2.3: Co-rotinas stackful e não stackful

2.3 Co-rotinas stackful

Em nosso sistema de classificação, introduzimos o conceito de uma co-rotina *stackful* como uma co-rotina que pode ser suspensa enquanto executa uma função aninhada, isto é, uma função direta ou indiretamente chamada por sua função principal. Nesse caso, quando essa co-rotina é reinvocada, sua execução continua no ponto exato de suspensão, ou seja, a execução da função aninhada é retomada, com seu estado local restaurado. A Figura 2.3 ilustra essa capacidade. São *stackful*, por exemplo, as co-rotinas providas em Simula, BCPL e Modula-2. Não são *stackful* os geradores de Icon, os iteradores de CLU e Sather, e os geradores de Python [78].

Em Python, uma função que contém um comando `yield` é denominada uma função geradora. Quando chamada, essa função retorna um objeto que pode ser (re)invocado em qualquer ponto de um programa, retornando um novo valor a cada invocação. Nesse sentido, um gerador Python — o objeto retornado pela função geradora — comporta-se como uma co-rotina assimétrica de primeira classe. Um gerador Python não é, contudo, uma construção *stackful*, pois somente pode suspender sua execução enquanto executa sua função principal. Essa restrição simplifica a implementação de geradores, eliminando a necessidade de pilhas separadas. Uma facilidade semelhante foi proposta para a versão 6 de Perl [15]: a adição de um novo

tipo de comando de retorno (também denominado `yield`), que preserva o estado local de uma subrotina.

Geradores de Python e construções similares, não *stackful* porém de primeira classe, permitem o desenvolvimento de iteradores e geradores simples, e também a manipulação assíncrona de estruturas de dados. Entretanto, a impossibilidade de suspender a execução dentro de funções aninhadas complica a estrutura de implementações mais elaboradas. Se, por exemplo, uma geração de itens utiliza um algoritmo recursivo, é necessária a criação de uma hierarquia de geradores auxiliares, que sucessivamente suspendem sua execução até que o ponto original de invocação seja alcançado.

Co-rotinas que não são *stackful* não são também suficientemente poderosas para a implementação de *multitasking*. Um requisito bastante comum a ambientes de tarefas concorrentes é a possibilidade de suspender uma tarefa durante a execução de uma operação bloqueante — por exemplo, uma operação de entrada ou saída. Sem essa suspensão, que permite a execução de outras tarefas, a eficiência do ambiente pode ser seriamente comprometida. Operações de entrada e saída são tipicamente executadas por funções auxiliares, normalmente oferecidas por bibliotecas. Dessa forma, co-rotinas que não são *stackful* não provêem um suporte adequado para a implementação de um ambiente de *multitasking* genérico, pois uma solução para esse requisito exige uma reformulação da estrutura das tarefas concorrentes e da biblioteca de entrada e saída. Por outro lado, se co-rotinas são *stackful*, esse requisito pode ser atendido modificando-se apenas as funções que executam operações bloqueantes.

2.4 Co-rotinas completas

Os argumentos que acabamos de expor permitem-nos concluir que duas das características que baseiam nosso sistema de classificação determinam a expressividade de um mecanismo de co-rotinas: se co-rotinas são valores de primeira classe e se são construções *stackful*. Sem essas características, co-rotinas não provêem suporte a diversos tipos de comportamento, e não podem ser consideradas como uma construção genérica de controle.

No Capítulo 4, mostraremos que o fato de uma co-rotina ser uma construção *stackful* de primeira classe é um requisito necessário e suficiente para se expressar co-rotinas simétricas em termos de assimétricas, e vice-versa, garantindo a equivalência dessas duas construções. Podemos então associar a presença dessas duas características ao conceito de uma co-

rotina *completa*, que, como também mostraremos, tem poder expressivo equivalente ao de continuações *one-shot*.

Apesar de não limitar o poder expressivo de um mecanismo de co-rotinas, a escolha de uma determinada disciplina de transferência de controle influencia a conveniência desse mecanismo. Na maioria das aplicações de co-rotinas, co-rotinas assimétricas são mais convenientes que co-rotinas simétricas. Podemos argumentar, portanto, que co-rotinas completas assimétricas são mais apropriadas como uma construção de controle genérica, o que justifica nosso maior interesse por esse modelo de co-rotinas.