Pontifícia Universidade Católica
do Rio de Janeiro

Ana Lúcia de Moura

Revisitando co-rotinas

Tese de Doutorado

Tese apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio como parte dos requisitos parciais para obtenção do título de Doutor em Informática

Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
Setembro de 2004

Ana Lúcia de Moura

Graduou–se em Matemática, modalidade Informática, na Universidade Federal do Rio de Janeiro em 1979. Obteve o título de Mestre em Informática pela PUC-Rio em 2000.

Ficha Catalográfica

Moura, Ana Lúcia de

Revisitando co-rotinas/ Ana Lúcia de Moura; orientador: Roberto Ierusalimschy. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2004.

v., ?? f: il. ; 29,7 cm

1. Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Linguagens de programação. 3. Construções de controle. 4. Co-rotinas. 5. Continuações. 6. Modelos de concorrência. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Ao Rafael

Agradecimentos

Ao Roberto, pela confiança, amizade e incentivo, e pelo muito que cresci e aprendi durante a realização deste trabalho.

À Noemi, pelos presentes que são seu exemplo, companhia e amizade.

Ao Rafael, grande companheiro, por seu carinho e compreensão. A ele, por quem tudo é possível, dedico esta tese.

Ao Paulo, pelo apoio sem o qual a conclusão deste trabalho não teria sido possível.

Aos muitos amigos que fiz nesses anos, e tenho grande alegria em manter. Júlia, Letícia, Isabel, Cristina, Silvana, Cecília, Clarissa, Carlos, Clínio, Renato, Tomás, André, Diego, muito obrigada por seu carinho e paciência.

A Hélia Ziller, Cláudia Lisboa e Ricardo Aquino, pelo suporte e incentivo para o fechamento desta etapa.

Ao Tecgraf, pelo privilégio de participar de seu grupo de pesquisa e desenvolvimento.

Ao Cnpq, pelo apoio financeiro.

Aos meus pais e à minha irmã Cristina, por seu amor.

Resumo

Moura, Ana Lúcia de; Ierusalimschy, Roberto. **Revisitando co-rotinas**. Rio de Janeiro, 2004. **??**p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O objetivo deste trabalho é defender o resgate do conceito de co-rotinas como uma construção de controle poderosa e conveniente, que pode substituir tanto continuações de primeira classe como *threads* com um conceito único e mais simples. Para suprir a ausência de uma definição precisa e adequada para o conceito de co-rotinas, propomos um novo sistema de classificação, e introduzimos o conceito de co-rotinas completas, para o qual provemos uma definição formal, baseada em uma semântica operacional. Demonstramos a seguir a equivalência de poder expressivo entre co-rotinas completas simétricas e assimétricas e entre co-rotinas completas e continuações *one-shot* tradicionais e parciais, discutindo as vantagens de co-rotinas completas assimétricas em relação a co-rotinas simétricas e continuações de primeira classe. Finalmente, analizamos os benefícios e desvantagens associados aos diversos modelos de concorrência, justificando a adoção de modelos alternativos a *multithreading* e o oferecimento de co-rotinas como uma construção básica de concorrência, adequada à implementação desses modelos.

Palavras–chave

co-rotinas; construções de controle; continuações; modelos de concorrência; threads.

Abstract

Moura, Ana Lúcia de; Ierusalimschy, Roberto. **Revisiting coroutines**. Rio de Janeiro, 2004. **??**p. PhD. Thesis — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The purpose of this work is to defend the revival of coroutines as a powerful and convenient control construct, which can replace both first-class continuations and threads with a single and simpler concept. In order to provide an adequate and precise definition of the concept of coroutines, we propose a new classifying system, and introduce the concept of complete coroutines, for which we provide a formal definition based on an operational semantics. We then demostrate that complete symmetric coroutines and complete asymmetric coroutines have equivalent expressive power, as well as complete coroutines and one-shot traditional and partial continuations. We also discuss the advantages of using complete asymmetric coroutines instead of symmetric coroutines or first-class continuations. Finally, we analyse the benefits and problems associated with different concurrency models, and argue in favor of the replacement of multithreading with alternative concurrency models and the provision of coroutines as a basic concurrency construct, adequate for the implementation of these alternative models.

Keywords

coroutines; control constructs; continuations; concurrency models; threads.

Conteúdo

Lista de Figuras

Temos a responsabilidade de trabalhar, de exercer nossos talentos e capacidades, de contribuir para a vida com nossa energia. Nossa natureza é criativa e, ao expressá-la, geramos constantemente mais entusiasmo e criatividade, estimulando um processo contínuo de contentamento no mundo à nossa volta. Trabalhar de bom grado, com toda a nossa energia e entusiasmo, é o modo que temos de contribuir para a vida.

Tarthang Tulku, *O caminho da habilidade.*

1
Introdução

O conceito de co-rotinas surgiu no início da década de 60, e constitui uma das propostas mais antigas de uma construção genérica de controle. Esse conceito é atribuído a Conway [14], que descreveu co-rotinas como "subrotinas que se comportam como se fossem o programa principal", e implementou essa construção para a estruturação de um compilador COBOL multi-fases, em uma arquitetura do tipo produtor–consumidor.

Durante a década de 70, e até meados da década de 80, a facilidade com que co-rotinas permitem expressar diversos comportamentos de controle foi intensamente explorada em contextos como simulação, inteligência artificial, programação concorrente, processamento de textos e manipulação de estruturas de dados [52, 62, 69]. No entanto, poucas linguagens incorporaram mecanismos de co-rotinas. Simula [8], BCPL [64], Modula-2 [90], CLU [60] e Icon [36] são alguns exemplos. Em geral, a conveniência de prover a um programador essa interessante construção de controle foi desconsiderada por projetistas de linguagens de programação.

A nosso ver, as principais críticas a co-rotinas e sua ausência em linguagens de programação *mainstream* decorrem, principalmente, da falta de um entendimento mais profundo sobre esse conceito. Essa falta de entendimento, deve-se, em grande parte, à ausência de uma definição precisa, e, consequentemente, de uma visão uniforme do conceito de co-rotinas. Adotada até hoje como a principal referência para esse conceito, a tese de doutorado de Marlin [62], de 1980, oferece como definição de uma co-rotina as seguintes características fundamentais:

- "dados locais a uma co-rotina tem seu valor preservado entre chamadas sucessivas";
- "a execução de uma co-rotina é suspensa quando o controle a deixa, sendo retomada no ponto de suspensão quando, em algum momento posterior, o controle retorna à co-rotina".

Pauli e Soffa [69], em um estudo comparativo de mecanismos de co-rotinas implementados na década de 70, oferecem uma descrição semelhante: "uma

co-rotina é uma generalização de uma *procedure* que permite que uma co-rotina seja temporariamente suspensa e subsequentemente retomada no ponto em que esteve ativa pela última vez".

Essas duas descrições correspondem, de fato, à noção consensual do conceito de co-rotinas, porém deixam em aberto diversas questões com respeito a essa construção, em especial três características relevantes:

- o mecanismo de transferência de controle, que pode prover co-rotinas *simétricas* ou *assimétricas*;
- se co-rotinas são valores de *primeira classe* (isto é, se podem ser invocadas em qualquer ordem e em qualquer lugar);
- se co-rotinas são construções *stackful* (isto é, se podem suspender sua execução dentro de um nível arbitrário de chamadas de funções).

Soluções particulares para essas questões resultaram em implementações de co-rotinas bastante diferentes, como as co-rotinas de Simula e Modula-2, os iteradores de CLU, os geradores de Icon e, mais recentemente, os geradores de Python [78]. Todas essas implementações satisfazem a caracterização genérica de Marlin, porém oferecem graus de expressividade significativamente diversos.

A introdução do conceito de *continuações de primeira classe* [28, 27], em meados da década de 80, contribuiu fortemente para o fim do interesse em co-rotinas como uma abstração genérica de controle. Ao contrário de co-rotinas, continuações de primeira classe possuem uma semântica bem definida e são reconhecidas por sua grande expressividade, explorada na implementação de diversas estruturas de controle como geradores, exceções, *backtracking* [22, 42], *multitasking* [88, 21], e também co-rotinas [41]. Contudo, à exceção de Scheme [51, 81], ML [40], Ruby [83] e uma implementação alternativa de Python [84], continuações de primeira classe não são oferecidas por linguagens de programação usuais [1].

Um obstáculo à incorporação de continuações de primeira classe em uma linguagem é a dificuldade de implementá-las de forma eficiente. Essa dificuldade é, em grande parte, resultante da necessidade de suporte a múltiplas invocações de uma mesma continuação. No entanto, na maioria de suas aplicações, continuações são invocadas apenas uma vez. Esse fato motivou a introdução do conceito de continuações *one-shot* [10], que, limitadas a uma única invocação, eliminam o *overhead* imposto por cópias de

[1] A linguagem Smalltalk [33] oferece *contextos de primeira classe*, a partir dos quais é possível oferecer continuações de primeira classe.

contextos de execução, utilizadas em implementações usuais de continuações *multi-shot* [45].

Uma outra dificuldade para o oferecimento de continuações de primeira classe como um recurso de programação é o próprio conceito de uma continuação como a representação do "resto de uma computação". Esse conceito não é compreendido e utilizado com facilidade, especialmente no contexto de linguagens procedurais. Parte dessa complexidade é eliminada com o uso de mecanismos baseados no conceito de *continuações parciais* [24, 50], cujo comportamento é, de certa forma, semelhante ao de uma função [71]. Esse tipo de comportamento favorece soluções mais simples e compreensíveis para as aplicações usuais de continuações, como demonstraram Danvy e Filinsky [18], Queinnec e Serpette [70] e Sitaram [79]. Em todas essas aplicações, podemos observar que uma única invocação de continuações é necessária, o que permite introduzir o conceito de continuações parciais *one-shot*. Apesar de apresentarem benefícios relevantes quando comparados a mecanismos de continuações tradicionais, mecanismos de continuações parciais permaneceram restritos a contextos de experimentação e pesquisa, não tendo sido incorporados nem mesmo a implementações de Scheme.

Um outro fator que muito contribuiu para a ausência de mecanismos de co-rotinas em linguagens de programação mais recentes foi a adoção do modelo conhecido como *multithreading* como um padrão "de fato" para a programação concorrente. Apesar dos diversos problemas associados ao uso do modelo de *multithreading* [68, 77], e dos vários esforços de pesquisa dedicados à investigação e exploração de modelos alternativos, linguagens *mainstream* modernas como Java [56], C# [3], Python [63] e Perl [87] ainda provêem *threads* como construção básica de concorrência.

Após um período de virtual esquecimento, observamos, em dois cenários, um ressurgimento do interesse em algumas formas de co-rotinas. O primeiro desses cenários corresponde ao desenvolvimento de aplicações concorrentes, onde alguns grupos de pesquisa têm explorado as vantagens de modelos de concorrência cooperativos como uma alternativa a *multithreading* [1, 5]. Nesse contexto, as construções utilizadas são tipicamente oferecidas por bibliotecas ou recursos do sistema — como, por exemplo, o mecanismo de *fibers* do Windows [75] — e seu uso é restrito ao suporte à programação concorrente. Interessantemente, apesar de os mecanismos descritos nesses trabalhos corresponderem essencialmente a implementações de co-rotinas, essas descrições sequer mencionam o termo "co-rotina".

O segundo cenário diz respeito a linguagens de *scripting* de propósito geral, notavelmente Python, Perl e Lua. Python incorporou, em 2001, uma

forma restrita de co-rotinas que permite a implementação de *geradores* [78]; um mecanismo similar foi proposto para Perl 6 [15]. Contudo, a implementação dessas formas de co-rotinas impede sua utilização como uma abstração genérica de controle, e também o suporte a *multitasking*, que é provido nessas linguagens através de bibliotecas de *threads*. Ao contrário dessas linguagens, Lua [49, 65] oferece um mecanismo de co-rotinas com poder suficiente para expressar diferentes comportamentos de controle, inclusive *multitasking*.

1.1 Objetivos e organização do trabalho

O principal objetivo deste trabalho é defender o resgate do conceito de co-rotinas como uma abstração de controle poderosa e conveniente, que pode substituir tanto continuações de primeira classe quanto *threads* com um conceito único e mais simples. Além disso, desejamos suprir a ausência, na literatura, de uma descrição adequada para o conceito de co-rotinas, permitindo o alcance de um entendimento mais profundo desse conceito.

1.1.1 Formalização do conceito de co-rotinas

Para alcançar nossos objetivos, entendemos que, em primeiro lugar, é necessário suprir a ausência de uma definição precisa para o conceito de co-rotinas.

Propomos, no Capítulo 2, um novo sistema de classificação para co-rotinas baseado nas três características citadas anteriormente: o mecanismo de transferência de controle, se co-rotinas são valores de primeira classe ou estruturas "confinadas" e se co-rotinas são construções *stackful*. Esse sistema de classificação nos permite distinguir as diversas implementações de co-rotinas com respeito à sua conveniência e poder expressivo.

A partir de nosso sistema de classificação, introduzimos o conceito de uma co-rotina *completa*, e provemos, no Capítulo 3, uma definição formal para esse conceito, baseada no desenvolvimento de uma semântica operacional.

1.1.2 Equivalência de co-rotinas completas e continuações one-shot

Baseados no conceito de expressividade desenvolvido por Felleisen [25], demonstramos, no Capítulo 4, que co-rotinas completas simétricas e assimétricas e continuações *one-shot* têm o mesmo poder expressivo. Discutimos, também, as similaridades entre continuações tradicionais e co-rotinas simétricas, e entre continuações parciais e co-rotinas assimétricas, mostrando que os mesmos benefícios associados ao uso de mecanismos de continuações parciais podem ser obtidos através de co-rotinas completas assimétricas.

No Capítulo 5, apresentamos implementações de diversos comportamentos de controle baseadas em um mecanismo de co-rotinas completas assimétricas. Esse conjunto de implementações inclui as aplicações mais relevantes de continuações de primeira classe, com a exceção de *multitasking*, que mereceu um capítulo à parte.

1.1.3 Co-rotinas completas como construção de concorrência

A parte final deste trabalho é dedicada à defesa de modelos de concorrência alternativos a *multithreading*, e de co-rotinas completas assimétricas como um suporte adequado à implementação desses modelos alternativos.

Em primeiro lugar, classificamos os diversos modelos de concorrência a partir da combinação de três características básicas: a presença ou não de diferentes linhas de execução (modelos mono e multi-tarefa), a política de escalonamento de tarefas (preemptiva ou não preemptiva) e o tipo de mecanismo utilizado para a interação entre tarefas (memória compartilhada ou troca de mensagens). Com base nessa classificação, analisamos os benefícios e desvantagens associados a cada modelo de concorrência, justificando a adoção de modelos alternativos a *multithreading* como processos, gerência cooperativa de tarefas e programação orientada a eventos.

Finalmente, mostramos que mecanismos de co-rotinas completas assimétricas permitem uma implementação trivial de gerência cooperativa de tarefas, além de prover facilidades bastante convenientes para a programação orientada a eventos. Concluimos, portanto, que uma linguagem que oferece co-rotinas completas não precisa oferecer *threads* ou qualquer outra construção de concorrência adicional para o suporte básico a implementações simples, adequadas e eficientes de *multitasking*.

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.

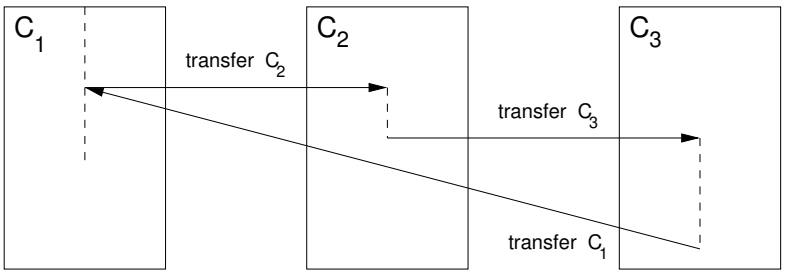

Figura 2.1: Transferência de controle com co-rotinas simétricas

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*.

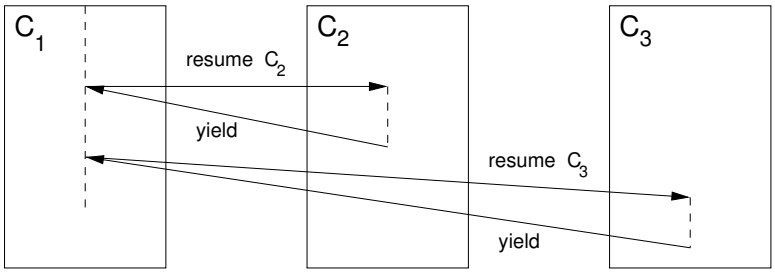

Figura 2.2: Transferência de controle com co-rotinas assimétricas

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; ofer-

ecer 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.

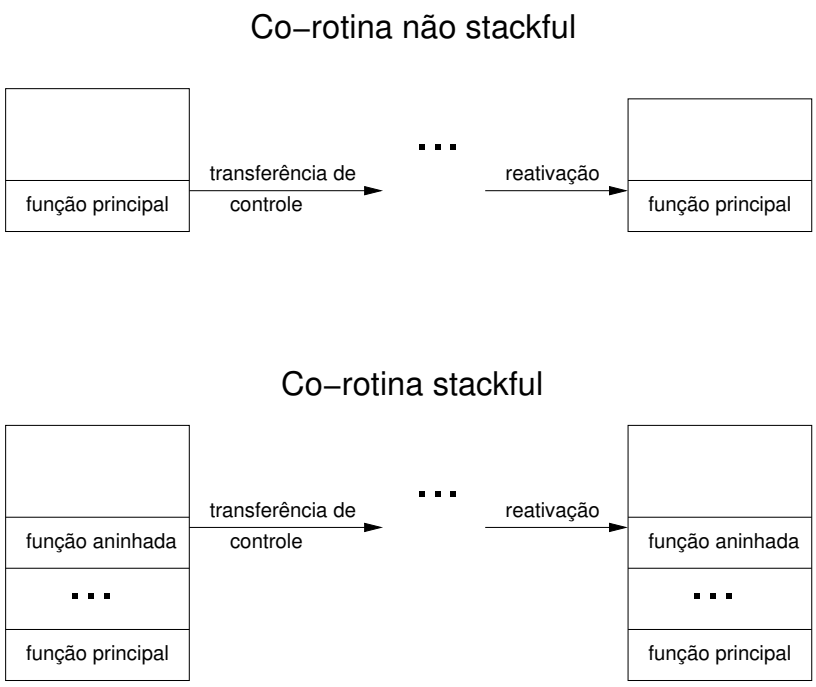

Figura 2.3: Co-rotinas stackful e não stackful

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.

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:

$$\theta : \mathit{variáveis} \;\rightarrow\; \mathit{valores}$$

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:

$$\theta : (\mathit{variáveis} \cup \mathit{rótulos}) \ \rightarrow \ \mathit{valores}$$

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:

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

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:

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

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

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

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

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.

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.

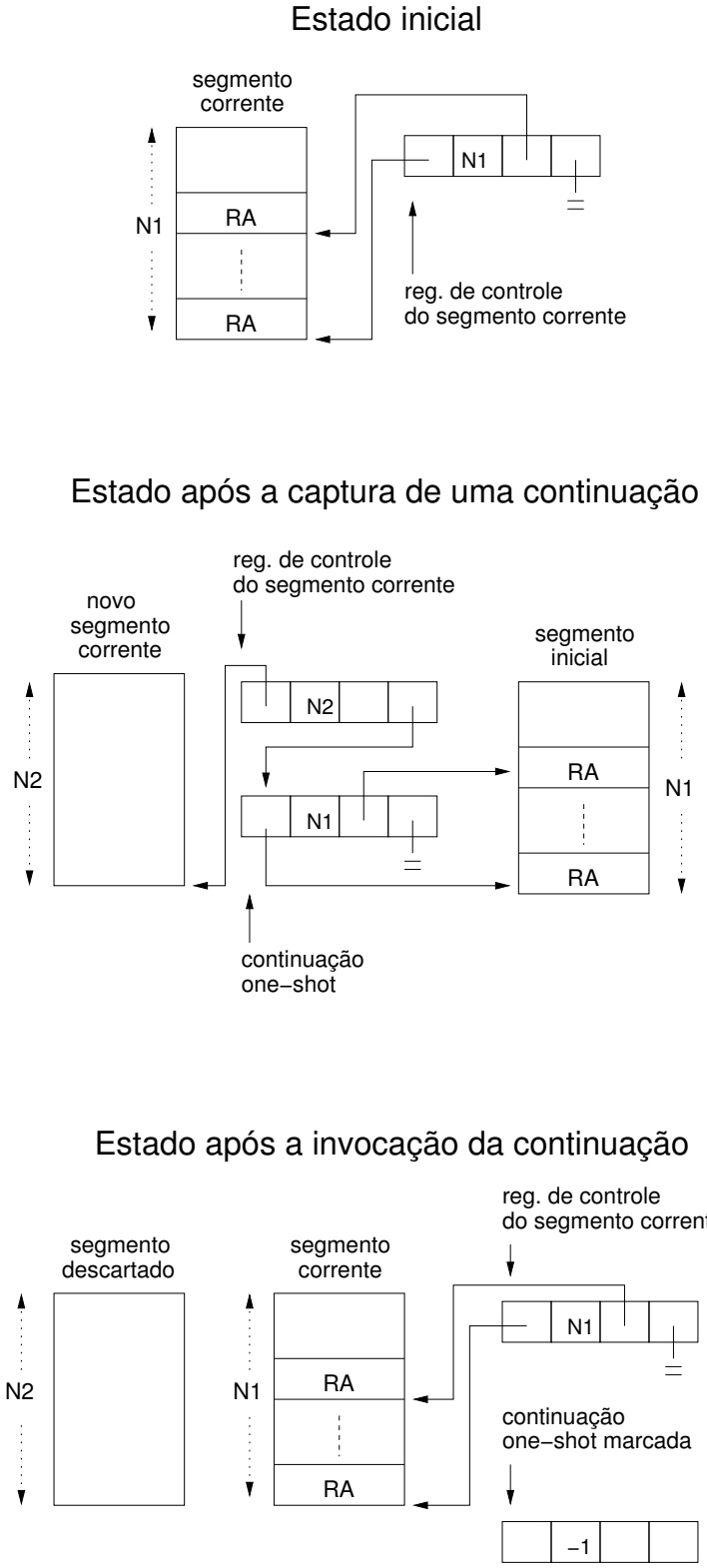

Figura 4.3: Implementação de continuações *one-shot*

```
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

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

```
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

mesmo em aplicações que utilizam continuações tradicionais *multi-shot*, como geradores [70] e *backtracking* [79]. O limite de uma única invocação 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

```
 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

processo é repetido até que a raiz da subcomputação "alvo" seja alcançada, incluindo na subcontinuação capturada todas as subcomputações 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

```
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

discutiremos a seguir, essas construções nem sempre são equivalentes com respeito à questões de conveniência e, em alguns casos, também de eficiência.

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 con-

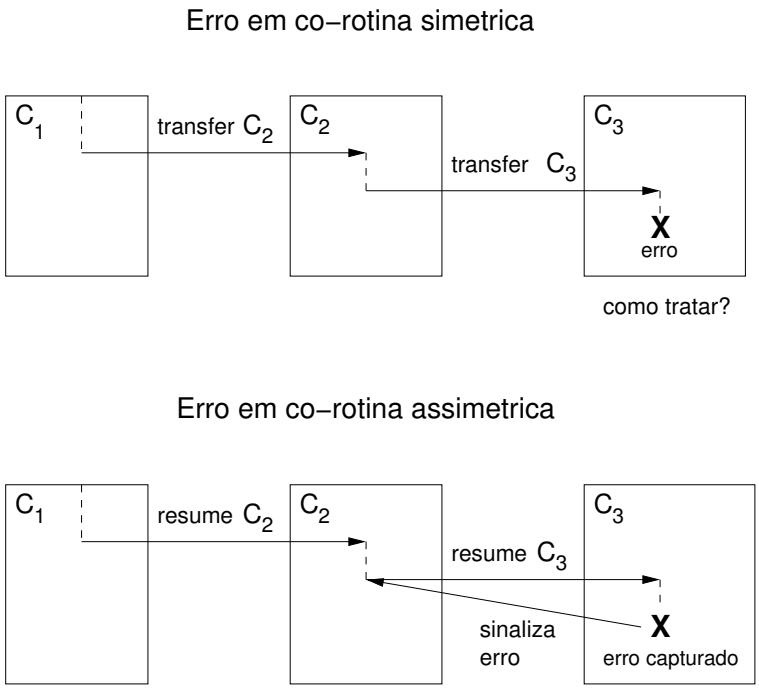

Figura 4.8: Erros de execução em co-rotinas simétricas e assimétricas

trole 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 — 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

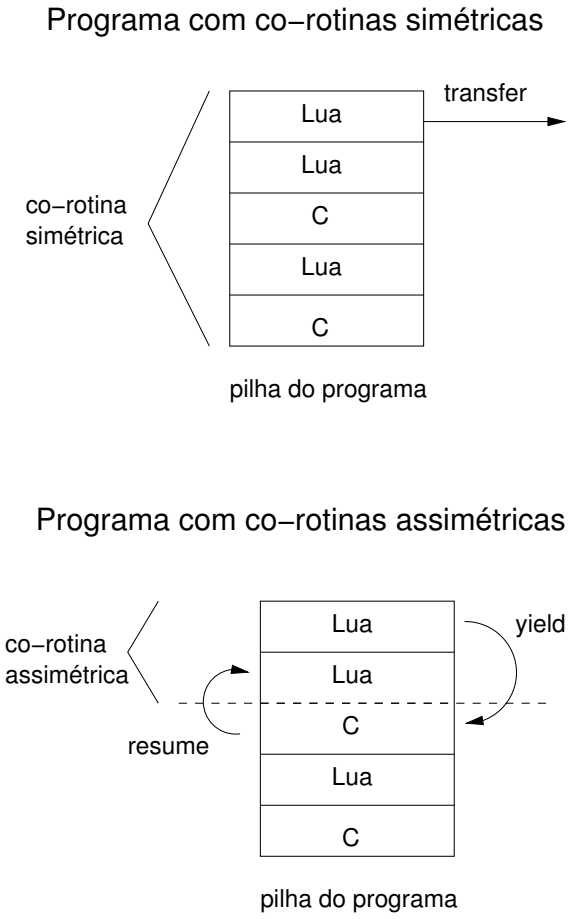

Figura 4.9: Pilha de um programa com co-rotinas simétricas e assimétricas

se não há uma chamada ativa a uma função C no segmento de pilha que representa a co-rotina.

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

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

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.

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`.

6
Co-rotinas completas e programação concorrente

Originado no contexto de sistemas operacionais, há mais de três décadas [20, 38, 39], o uso de concorrência é cada vez mais presente no desenvolvimento de sistemas e aplicações em diferentes domínios. Exemplos de cenários onde a programação concorrente é observada incluem serviços WEB e aplicações distribuídas em geral, sistemas de interface gráfica, aplicações paralelas, jogos, simulação de eventos discretos, e outros tipos de aplicações onde a decomposição em tarefas paralelas — isto é, executadas em concorrência real ou simulada — visa atender uma combinação de requisitos como desempenho, responsividade e simplicidade de projeto.

Atualmente, o modelo conhecido como *multithreading* é praticamente um padrão para o desenvolvimento de aplicações concorrentes. A maioria das linguagens *mainstream* modernas, como Java [56], C# [3], Perl [87] e Python [63], e bibliotecas largamente utilizadas como pThreads [67], provêem *threads* como construção básica de concorrência. A noção de que *threads* representam uma ferramenta mais simples e eficiente para o suporte à implementação de programação concorrente contribuiu de forma decisiva para virtual abandono do interesse em co-rotinas como construção de concorrência.

Multithreading envolve tipicamente a execução de tarefas concorrentes que compartilham memória e são sujeitas a preempção — isto é, à perda involuntária do controle do processador. Essas características são essenciais no contexto que originou a programação concorrente — o desenvolvimento de sistemas operacionais — onde requisitos de responsividade e desempenho são extremamente rigorosos. Entretanto, os requisitos de aplicações concorrentes são, em geral, bastante diferentes dos requisitos de um sistema operacional. Além disso, programadores de aplicações, diferentemente de desenvolvedores de sistemas operacionais, são muitas vezes relativamente inexperientes, ou pouco expostos aos problemas e soluções relacionados ao uso de concorrência. Nesse cenário, o uso do modelo de *multithreading* é claramente inadequado, e pode neutralizar os benefícios potenciais da programação con-

corrente, produzindo aplicações desnecessariamente complexas, incorretas e de baixo desempenho.

Nos últimos anos, os diversos problemas relacionados ao uso de *multithreading* têm motivado a investigação e proposta de ambientes baseados em modelos de concorrência alternativos, destacando-se a programação orientada a eventos [47, 89, 7, 16, 91] e, em menor escala, a gerência cooperativa de tarefas [1, 5]. Ao evitar a união de mecanismos de preempção e memória compartilhada, esses modelos permitem o desenvolvimento de aplicações menos complexas, de melhor desempenho, e com maiores garantias de correção.

Um outro modelo alternativo, usualmente desconsiderado, é o modelo de *processos* — um modelo de concorrrência baseado na interação de tarefas através de troca de mensagens. O pouco investimento em implementações eficientes desse tipo de modelo, especialmente em mecanismos para a interação entre processos em uma mesma máquina, é em parte responsável pela rejeição ao uso de processos para a implementação de aplicações concorrentes, privilegiando a adoção de modelos baseados em memória compartilhada, particularmente *multithreading*. Entretanto, o modelo de processos é bastante adequado para o desenvolvimento de aplicações naturalmente decompostas em módulos fracamente acoplados. Além de favorecer uma melhor estruturação desse tipo de aplicações, o uso de processos facilita a transposição dessas aplicações para ambientes distribuídos. O modelo de processos constitui também uma opção conveniente para ambientes multiprocessadores, especialmente quando combinado a um dos outros modelos alternativos.

Neste capítulo, nós analisamos benefícios e desvantagens associados a cada tipo de modelo de concorrência, justificando a adoção de modelos alternativos a *multithreading*. Mostramos também que co-rotinas completas assimétricas constituem uma construção bastante apropriada como suporte a modelos baseados em gerência cooperativa de tarefas e programação orientada a eventos e, portanto, mais adequadas que *threads* como uma construção básica de concorrência para ambientes de memória compartilhada.

6.1 Análise de modelos de concorrência

Apesar de introduzir novas oportunidades, o uso de concorrência introduz também problemas não encontrados no projeto, desenvolvimento e manutenção de aplicações seqüenciais. A escolha de um determinado mod-

elo de concorrência influi consideravelmente na minimização, ou potencialização, desses problemas.

6.1.1 Caracterização de modelos de concorrência

Para efeito da análise que vamos apresentar, utilizamos uma caracterização de modelos de concorrência baseada na combinação de três mecanismos básicos:

Mono/multi-tarefa: O uso de concorrência envolve a decomposição de uma aplicação em um conjunto de tarefas, ou fluxos de controle, independentes. Quando todas as tarefas executam sem transferência de controle até o seu final, em qualquer instante uma única linha de execução é definida. Denominamos este tipo de mecanismo *mono-tarefa*. Por outro lado, quando diversas tarefas executam simultaneamente, em um determinado instante múltiplas linhas de execução poderão estar definidas. Denominamos esse mecanismo *multi-tarefa*.

Escalonamento: Um modelo multi-tarefa implica em um entrelaçamento de diferentes linhas de execução; a política de escalonamento utilizada determina de que formas esse entrelaçamento poderá ocorrer. Quando essa política é *preemptiva* — seja pela adoção de mecanismos de prioridade, de *time slicing*, ou ambos — uma alternância entre tarefas pode ocorrer a qualquer momento. Nesse caso, o entrelaçamento de linhas de execução é arbitrário, ou não-determinístico. Quando o escalonamento é *não preemptivo*, uma transferência de controle ocorre somente em pontos bem definidos da execução de uma tarefa. Essa transferência de controle pode ser implícita ou explícita. Uma transferência implícita ocorre, por exemplo, quando uma tarefa inicia uma operação de entrada e saída, ou quando se coloca à espera de uma condição. Uma transferência explícita ocorre quando uma tarefa cede voluntariamente o controle do processador.

Interação entre tarefas: A decomposição de uma aplicação em um conjunto de tarefas envolve a necessidade de cooperação entre elas. Essa cooperação pode ser obtida através de dois mecanismos básicos: *memória compartilhada* ou *troca de mensagens* [2]. Um mecanismo de memória compartilhada permite que tarefas independentes troquem informações através do uso de estruturas de dados armazenadas em

um espaço de endereçamento comum. Um mecanismo de troca de mensagens permite a transferência de informações através da provisão de *canais de comunicação* e de operações básicas para envio e recepção de mensagens através desses canais.

Principais modelos de concorrência

Em nossa abordagem, um modelo de concorrência é o resultado da combinação dos mecanismos descritos. Entretanto, somente algumas dessas combinações resultam em modelos relevantes. Esses modelos, e exemplos típicos de suas implementações, são identificados a seguir:

- mono-tarefa, com memória compartilhada: *orientação a eventos*
- multi-tarefa, com preempção e memória compartilhada: *multithreading*
- multi-tarefa, sem preempção e com memória compartilhada: *gerência cooperativa de tarefas*
- multi-tarefa, com/sem preempção e com troca de mensagens: *processos*

Uma consideração importante diz respeito à categorização do modelo conhecido como *multithreading*. As implementações mais conhecidas e utilizadas desse modelo — como a de Java e pThreads — não oferecem qualquer garantia quanto à política de escalonamento adotada. Para garantir a portabilidade de uma aplicação, seu desenvolvedor deve então a assumir o comportamento de um modelo preemptivo, e implementar soluções para os problemas associados a esse comportamento, mesmo que em algumas plataformas a política de escalonamento não seja preemptiva.

6.1.2 Complexidade de Programação

Uma noção bastante comum é a de que quanto mais distante do modelo seqüencial — mais familiar, ou intuitivo, à maioria dos desenvolvedores — mais difícil é o projeto e programação de uma aplicação concorrente.

Na programação orientada a eventos, uma aplicação é tipicamente estruturada como um conjunto de tarefas tratadoras de eventos (*event handlers*), ativadas por um *loop* à medida que os eventos correspondentes são observados. Um dos maiores obstáculos ao uso desse modelo é o estilo

de execução tipicamente "reativo" das tarefas, e a conseqüente inversão no fluxo de controle da aplicação. Esse estilo de programação simplifica, contudo, o projeto e programação de aplicações e sistemas inerentemente assíncronos, como, por exemplo, sistemas de interface gráfica e aplicações distribuídas [86, 58, 7, 76].

Para garantir um nível adequado de concorrência à aplicação, modelos mono-tarefa, como o de orientação a eventos, requerem tarefas compostas por pequenos trechos de código, necessariamente não bloqueantes. Em algumas situações, essa exigência pode implicar na quebra de uma tarefa em duas ou mais partes, e, consequentemente, na necessidade de preservação e recuperação de contextos de execução — isto é, na gerência "manual" da pilha de execução [1]. Mecanismos de continuações de primeira classe e co-rotinas (como veremos na Seção 6.3) provêem suporte para a manutenção do estado de tratadores de eventos, simplificando a implementação de aplicações que adotam modelos mono-tarefa.

A aparência de programação sequencial é mais facilmente obtida em modelos multi-tarefa. Nesses modelos, a granularidade das tarefas é significativamente maior, pois é a alternância entre elas que determina o nível de concorrência da aplicação. Sob esse aspecto, modelos *preemptivos* são usualmente considerados mais simples, pois essa alternância é definida pelo escalonador. Por outro lado, como veremos a seguir, a união de mecanismos de preempção e de memória compartilhada potencializa os demais problemas enfrentados por desenvolvedores de aplicações concorrentes. Dessa forma, a aparente maior simplicidade de modelos que apresentam essa união pode, na verdade, envolver um nível de complexidade de programação considerável.

O uso de modelos de concorrência onde a interação entre tarefas é baseada em canais de comunicação simplifica o projeto e desenvolvimento de aplicações onde as tarefas são fracamente acopladas, favorecendo uma modularização adequada dessas aplicações. Essa característica pode ser observada, por exemplo, em aplicações paralelas, um cenário onde espaços de tuplas [32] são bastante utilizados. Apesar de esse mecanismo ser geralmente descrito como uma implementação de memória compartilhada, a leitura e escrita de tuplas é realizada somente através de operações explícitas para o envio e recepção de informações (`out`, `in`), que implementam um acesso serializado — ou seja, não simultâneo — às informações compartilhadas. Dessa forma, podemos categorizar espaços de tuplas como um mecanismo baseado em troca de mensagens. Veremos mais tarde que modelos baseados em troca de mensagens são também apropriados para a implementação de arquite-

turas baseadas em computação por estágios, uma solução conveniente para ambientes multi-processadores.

Interações entre tarefas fortemente acopladas, que envolvem o compartilhamento de estruturas de dados, são em geral mais simples e mais eficientes com o uso de mecanismos de memória compartilhada. O uso de memória compartilhada favorece também a preservação da aparência de uma programação seqüencial convencional. Além disso, soluções que envolvem a transferência de estruturas de dados compartilhadas em menssagens podem gerar problemas de inconsistência e serialização.

6.1.3 Garantia de correção

A maior dificuldade introduzida pelo uso de concorrência é garantir a *correção* da aplicação. Essa correção está associada ao atendimento de dois tipos de propriedades: *safety* e *liveness* [2].

Propriedades de *safety* garantem que uma aplicação jamais alcance um estado irremediavelmente inconsistente ou incorreto. Essa garantia exige, em primeiro lugar, a coordenação do acesso aos recursos compartilhados pelas diversas tarefas, evitando que uma interferência entre elas comprometa a consistência do estado da aplicação. Essa coordenação envolve, tipicamente, o uso de mecanismos de sincronização por *exclusão mútua* e por *condições* como semáforos, *locks*, monitores e guardas [9, 6, 2, 56].

Uma outra exigência relacionada à correção da aplicação é a ausência de *deadlocks*, situações onde um conjunto de tarefas, ciclicamente dependentes, é bloqueado à espera de condições que jamais ocorrerão. A ocorrência de um *deadlock* pode ser definida como um estado inconsistente da aplicação e, portanto, como uma falha no atendimento a propriedades de *safety*. Alguns autores porém, como Lea [56], consideram a ocorrência de *deadlocks* como falhas no atendimento a propriedades de *liveness*.

Excluindo-se as situações de *deadlock*, o conceito de *liveness* diz respeito à justiça (*fairness*) na alocação do processador entre as tarefas. Uma alocação injusta pode levar a uma situação onde uma ou mais tarefas jamais conseguem o controle do processador (*starvation*). Contudo, qualquer situação que impeça uma ou mais tarefas de progredir adequadamente pode ser considerada uma falha de justiça.

Em modelos mono-tarefa, a coordenação do acesso a recursos compartilhados é trivial, pois cada tarefa executa sem cessão de controle até o seu final. Além disso, como não há bloqueio de tarefas, situações de *deadlock*

jamais ocorrerão. Contudo, se tratadores de eventos são decompostos em diversas partes, mecanismos de sincronização podem ser necessários quando tratadores diferentes compartilham recursos. Nesse caso, o comportamento é semelhante ao de um modelo multi-tarefa não preemptivo, discutido mais adiante.

A justiça na alocação do processador em modelos mono-tarefa depende, basicamente, de dois fatores. O primeiro está relacionado à granularidade das tarefas, que deve estar de acordo com os requisitos específicos da aplicação. O segundo fator diz respeito ao escalonamento das tarefas — isto é, a determinação do próximo evento a ser tratado. Neste caso, se algum mecanismo de prioridades for adotado, cuidados especiais devem ser tomados para evitar situações de *starvation*.

A ausência de problemas de consistência e de *deadlocks* simplifica muito a depuração de aplicações concorrentes que adotam um modelo mono-tarefa. Comportamentos inadequados dizem respeito, apenas, a problemas de responsividade, cuja detecção é elementar. Cenários de execução são facilmente reproduzidos, pois dependem exclusivamente da ordenação de eventos. Dessa forma, a depuração desse tipo de aplicações não oferece grandes dificuldades.

Em modelos multi-tarefa não preemptivos, como os momentos onde uma tarefa cede o controle do processador são bem definidos, a manutenção da consistência da aplicação é bastante facilitada. Muitas vezes, mecanismos de sincronização podem ser dispensados ou simplificados, o que diminui a possibilidade de ocorrência de *deadlocks*. Entretanto, para que essa estratégia não comprometa a consistência da aplicação, é indispensável que a semântica das operações que podem acarretar uma cessão implícita de controle — por exemplo, operações de entrada e saída — seja bem definida.

Na ausência de preempção, o sequenciamento de tarefas é determinístico. Dessa forma, cenários de execução são mais facilmente reproduzidos, o que simplifica a deteção e correção de problemas.

O uso de uma política de escalonamento não preemptiva pode, contudo, reduzir o nível de concorrência, ou justiça, da aplicação. Uma tarefa que executa durante muito tempo até ceder o controle, implícita ou explicitamente, impede o progresso das demais tarefas. Neste caso, o nível de justiça adequado pode ser obtido através da redução da granularidade "interna" da tarefa, com a inserção de operações de cessão voluntária de controle. Essa estratégia pode introduzir alguma complexidade na aplicação. Por outro lado, problemas de justiça não são difíceis de identificar, e, como vimos, tem solução trivial.

Modelos baseados em troca de mensagens em geral não oferecem grande dificuldade de coordenação. Em primeiro lugar, o próprio mecanismo de troca de mensagens oferece facilidades de sincronização (Lauer e Needham [55], por exemplo, demonstram a dualidade desse mecanismo e mecanismos de sincronização como monitores). Além disso, esse modelo favorece a decomposição de uma aplicação em tarefas fracamente acopladas, o que reduz o uso de recursos compartilhados. Essa característica simplifica a sincronização de tarefas, reduzindo a probabilidade de erros de inconsistência e de ocorrência de *deadlocks*.

Com respeito à justiça na alocação do processador, o uso de modelos preemptivos oferece alguma facilidade, pois a responsabilidade pela distribuição de recursos é delegada ao mecanismo de escalonamento. Contudo, situações de *starvation* podem ocorrer, especialmente quando mecanismos de prioridade são adotados.

Em modelos preemptivos baseados em memória compartilhada, a coordenação de tarefas é muito mais complexa [9]. Mesmo a semântica de construções simples da linguagem, como o incremento de uma variável compartilhada, pode ser afetada [56]. Erros sutis resultantes da dificuldade em identificar as regiões críticas de uma tarefa são bastante comuns, e, na maioria das vezes, muito difíceis de detetar.

A maior preocupação com a consistência do estado da aplicação, e a dificuldade em delimitar adequadamente suas regiões críticas, provoca um uso intenso — e muitas vezes indiscriminado — de mecanismos de sincronização. Dessa forma, o número de *deadlocks* potenciais aumenta significativamente. Apesar de exaustivamente exploradas na literatura, soluções que visam garantir a consistência do estado da aplicação evitando, ao mesmo tempo, a ocorrência de *deadlocks* são de difícil compreensão e implementação, mesmo para desenvolvedores com grande experiência.

Finalmente, o não-determinismo no sequenciamento de tarefas em modelos preemptivos dificulta, ou mesmo impede, a reprodução de cenários de execução. Como aplicações baseadas em modelos que combinam preempção com o uso de memória compartilhada são bem mais suscetíveis a problemas de correção, um esforço consideravelmente maior é dispendido na depuração dessas aplicações.

6.1.4 Desempenho

Além de influenciar a complexidade de projeto e a dificuldade em garantir a correção de uma aplicação, as características de um modelo de concorrência interferem diretamente em seu desempenho. O uso de mecanismos de sincronização, a gerência de múltiplos contextos de execução e a política de escalonamento adotada introduzem *overheads* que podem prejudicar o atendimento a requisitos como disponibilidade, tempo de resposta e escalabilidade.

De forma geral, modelos mono-tarefa oferecem um bom desempenho e escalabilidade [47, 16]. Essa característica deve-se à inexistência de *overheads* decorrentes de trocas de contexto, ao uso de memória compartilhada e à ausência de mecanismos de sincronização.

O desempenho de uma aplicação concorrente é significativamente afetado por mecanismos de sincronização. O uso intenso desses mecanismos, além de envolver um custo de processamento, aumenta a probabilidade de ocorrência de situações de contenção, reduzindo o nível de concorrência da aplicação. Dessa forma, problemas de desempenho são muito mais facilmente observados em modelos multi-tarefa preemptivos que utilizam memória compartilhada, como *multithreading* [77]. Modelos não preemptivos ou baseados em mecanismos de troca de mensagens podem oferecer melhores resultados [1, 89, 54].

Em modelos multi-tarefa, trocas de contexto são inevitáveis. Entretanto, o custo associado é maior quando uma política de escalonamento preemptiva é adotada, pois a alternância de tarefas é em geral muito mais frequente. Além disso, a própria implementação de mecanismos de *time slicing* ou prioridades representa um adicional de processamento. Para garantir um nível de desempenho adequado, muitas vezes é necessário limitar o número de tarefas simultâneas [77, 47]. Essa necessidade é maior quando a implementação de um modelo de concorrência é baseada em mecanismos oferecidos pelo sistema operacional, pois o custo de criação de novas tarefas pode ser significativo.

6.2 Gerência cooperativa de tarefas

Gerência cooperativa de tarefas (*cooperative task management*) é o melhor exemplo de um modelo de concorrência multi-tarefa não preemptivo baseado em memória compartilhada. Na maioria das situações, a gerência

```
tasks = {}  -- lista de tarefas vivas

-- cria uma tarefa
function create_task(f)
  local co = coroutine.wrap(function()
               f()
               return "task ended" -- sinaliza fim da tarefa
             end)

  -- insere a nova tarefa na lista
  table.insert(tasks, co)
end

-- escalonador
function dispatcher()
  while true do
    local n = table.getn(tasks)  -- número de tarefas vivas
    if n == 0 then break end
    for i = 1, n do
      local res = tasks[i]()     -- reativa tarefa
      if res == "task ended" then
        table.remove(tasks, i)   -- tarefa terminou
        break
      end
    end
  end
end
```

Figura 6.1: Gerência cooperativa de tarefas com co-rotinas assimétricas

cooperativa pode substituir com vantagens ambientes de *multithreading*, pois além de preservar a aparência de programação sequencial, esse tipo de modelo minimiza a necessidade de sincronização, favorecendo o desenvolvimento de aplicações mais simples e menos suscetíveis a incorreções.

Ambientes de gerência cooperativa de tarefas podem ser implementados com facilidade a partir de mecanismos de co-rotinas completas assimétricas. Assim como uma *thread*, uma co-rotina representa uma unidade de execução independente, associada a um estado local privado, compartilhando dados e outros recursos globais com outras co-rotinas. Porém, enquanto o conceito de uma *thread* é tipicamente associado a um escalonamento preemptivo — ou seja, a cessões involuntárias de controle — o escalonamento de co-rotinas é essencialmente cooperativo, pois uma co-rotina deve explicitamente suspender sua execução para permitir que outra co-rotina possa executar.

A Figura 6.1 apresenta uma implementação de um ambiente de

gerência cooperativa de tarefas baseado no mecanismo de co-rotinas completas assimétricas provido pela linguagem Lua (descrito na Seção 3.3). Nessa implementação, uma tabela (`tasks`) representa a lista de tarefas vivas, armazenando as referências para as co-rotinas correspondentes. A função `create_task` é responsável pela criação de novas tarefas. Ela recebe como parâmetro uma função que implementa a tarefa, cria uma co-rotina cujo corpo invoca essa função e insere a referência para essa co-rotina na lista de tarefas vivas.

A função `dispatcher` implementa o escalonador de tarefas. Esse escalonador é simplesmente um *loop* que itera sobre a lista de tarefas, reativando as tarefas vivas (que executam até terminar ou solicitar sua suspensão) e removendo da lista as tarefas que terminam. O término de uma tarefa (ou seja, o término da sua função principal) é sinalizado pelo corpo da co-rotina através do retorno de um valor pré-estabelecido (a *string* `"task ended"`) ao escalonador. A suspensão de uma tarefa é obtida pela invocação da função `coroutine.yield`. Na ausência de um parâmetro, essa função retornará o valor `nil` ao escalonador, que assim saberá que a tarefa correspondente ainda está viva.

Em aplicações que associam uma nova tarefa a cada requisição de serviço, o custo de criação de tarefas deve ser minimizado. Nesse caso, a utilização de um *pool* de co-rotinas reutilizáveis pode ser conveniente. O Apêndice A mostra uma implementação dessa facilidade.

Uma implementação trivial de um ambiente de gerência cooperativo, como a que acabamos de apresentar, pode implicar em um comportamento inaceitável para algumas aplicações. Se, por exemplo, uma co-rotina é bloqueada ao invocar uma operação de entrada ou saída, nenhuma outra co-rotina receberá o controle e, portanto, toda a aplicação será bloqueada até o término dessa operação. Dessa forma, o nível de concorrência da aplicação é afetado, o que pode comprometer consideravelmente o seu desempenho.

Contudo, uma solução bastante simples para esse problema pode ser implementada. As bibliotecas de entrada e saída providas pela maioria das plataformas oferecem usualmente facilidades que permitem associar um tempo máximo de espera (um *timeout*) para o término de uma operação, e aguardar mudanças de estado em um conjunto de recursos, que podem representar arquivos ou canais de comunicação como *sockets*. Exemplos dessas facilidades são as funções `poll` e `select`, providas em plataformas UNIX. Operações de entrada e saída assíncronas oferecidas em plataformas Windows podem também ser utilizadas para implementar essas facilidades.

Com essas facilidades, podemos oferecer uma biblioteca auxiliar de

```
function io(resource, <parâmetros adicionais>)
  while true do
    status = basic_io(resource, timeout, <parâmetros>)
    if status == "timeout" then
      coroutine.yield(resource)
    else
      return status
    end
end
```

Figura 6.2: Operação de entrada ou saída não bloqueante

funções de entrada e saída que suspendem a co-rotina em execução se a operação desejada não pode ser imediatamente satisfeita. Nesse caso, um valor que representa o recurso associado à operação invocada é retornado ao chamador da co-rotina. Quando a co-rotina é reativada, a função de entrada ou saída é retomada, finalizando a operação ou novamente suspendendo a co-rotina se essa operação não se completa no tempo determinado. A Figura 6.2 mostra uma possível estrutura para esse tipo de função.

Podemos então modificar a implementação do escalonador para que quando nenhuma tarefa possa ser ativada — ou seja, quando todas as tarefas vivas estão à espera de operações de entrada ou saída — o escalonador se bloqueie à espera de uma mudança de estado em algum dos recursos envolvidos. No Apêndice A apresentamos essa nova implementação do escalonador. Ierusalimschy [49] apresenta também a implementação de uma aplicação concorrente que utiliza esse tipo de solução para a transferência simultânea de arquivos através do protocolo HTTP.

Mecanismos básicos de sincronização para ambientes de gerência cooperativa baseados em co-rotinas completas assimétricas podem ser oferecidos com bastante facilidade. Um mecanismo de exclusão mútua pode ser obtido através de uma implementação trivial de semáforos binários [19, 6]. Mecanismos de sincronização por condições também têm implementação bastante simples. O Apêndice A mostra também uma possível implementação desses mecanismos.

Comentamos anteriormente que um interesse em ambientes de concorrência baseados em gerência cooperativa de tarefas como uma alternativa a *multithreading* começa a ser observado. Adya et al [1] descrevem implementações desse tipo de ambiente para o desenvolvimento de dois tipos de aplicações concorrentes. Na primeira implementação, um sistema de arquivos distribuído, o ambiente de gerência cooperativa é baseado no mecanismo de *fibers* do Windows [75] (que constitui, na verdade, uma im-

plementação de co-rotinas simétricas). Na segunda implementação, uma aplicação para comunicação *wireless* em dispositivos do tipo PDA, o ambiente de gerência cooperativa é simulado a partir de um mecanismo de *threads* e da associação de uma variável de condição a cada uma das tarefas, e ao escalonador. Para (re)ativar uma tarefa, o escalonador sinaliza a condição associada à tarefa, e se bloqueia em sua própria condição. Para suspender sua execução, uma tarefa sinaliza a condição do escalonador, e se bloqueia à espera da sinalização de sua condição. É claro perceber que esses procedimentos correspondem também a uma implementação de co-rotinas simétricas. Contudo, o trabalho de Adya et al sequer menciona o termo "co-rotina", e denomina sua proposta de "gerência automática da pilha de execução" (*automatic stack management*).

Behren et al [5] propõem um modelo multi-tarefa não preemptivo — essencialmente, um ambiente de gerência cooperativa de tarefas — como uma alternativa para a implementação de servidores com um alto nível de concorrência. Segundo os autores, esse tipo de modelo é tão eficiente quanto a programação orientada a eventos, porém favorece um estilo de programação mais natural. Para validar sua proposta, implementam um ambiente cooperativo baseado em uma biblioteca de co-rotinas desenvolvida para a linguagem C [85]. Interessantemente, esse trabalho não considera que co-rotinas são a construção básica do ambiente de concorrência, que é caracterizado como uma implementação de *multithreading*.

Ganz, Friedman e Wand [31] descrevem um estilo de programação — denominado *trampolined* — onde um programa é organizado como um *loop* que escalona diferentes computações, cuja execução progride em passos discretos. A aplicação desse estilo de programação para a implementação de *multitasking* corresponde, basicamente, à implementação de gerência cooperativa de tarefas baseada em corotinas completas assimétricas, que apresentamos nesta Seção. De fato, esse trabalho menciona a semelhança desse estilo com o uso de continuações parciais, cuja similaridade com co-rotinas completas assimétricas foi discutida no Capítulo 4. A semelhança com co-rotinas é também mencionada, porém os autores associam o conceito de co-rotinas apenas a co-rotinas simétricas, e, assim, argumentam que a construção de concorrência por eles introduzida (denominada *threads*) não corresponde a co-rotinas.

Modelos de concorrência baseados em gerência cooperativa de tarefas não são, em geral, adequados a ambientes multi-processadores. Nesse tipo de ambiente, modelos que permitem a execução simultânea de tarefas alocadas a diferentes processadores — baseados em *threads* ou em processos —

favorecem um maior nível de concorrência para as aplicaçôes, permitindo um melhor aproveitamento dos recursos disponíveis. Conforme discutimos na seção anterior, o modelo de *multithreading*, apesar de ser usualmente considerado mais simples e mais eficiente, envolve uma maior necessidade de mecanismos de sincronização, o que pode comprometer tanto o desempenho de uma aplicação concorrente como também sua correção. O modelo de processos constitui uma alternativa natural para ambientes multi-processadores, podendo oferecer melhores resultados, especialmente para aplicações que podem ser decompostas em módulos fracamente acoplados.

A combinação de gerência cooperativa de tarefas com modelos baseados em troca de mensagens, como o de processos, representa também uma opção bastante interessante para ambientes multi-processadores. Welsh, Culler e Brewer [89], por exemplo, propõem uma arquitetura para a implementação de servidores WEB onde o atendimento a um serviço é decomposto em uma sequência de estágios. Cada estágio é implementado em um processo separado, e a comunicação entre estágios adjacentes é realizada através de filas de mensagens. Dentro de cada estágio, o atendimento aos "sub-serviços" a ele correspondentes é implementado em um ambiente *multi-threaded*. Comparações de desempenho apresentadas pelos autores mostram que em diversos cenários essa arquitetura, denominada SEDA, permite obter resultados significativamente melhores do que arquiteturas baseadas unicamente no modelo de *multithreading*. A natureza assíncrona da comunicação entre estágios é responsável pela caracterização da arquitetura SEDA como um ambiente de orientação a eventos. Contudo, essa arquitetura é, na verdade, um modelo híbrido, baseado na combinação de processos, orientação a eventos e *multithreading*. A implementação de estágios através de gerência cooperativa de tarefas, ao invés de *multithreading*, pode apresentar um desempenho ainda melhor, por evitar, ou reduzir, a necessidade de sincronização intra-estágios. Além disso, como veremos na próxima seção, co-rotinas oferecem um suporte bastante conveniente para a programação orientada a eventos.

Outras formas de decomposição que permitem explorar a combinação de processos e gerência cooperativa de tarefas para um melhor aproveitamento de recursos em ambientes multi-processadores podem ser desenvolvidas. Um exemplo é o atendimento simultâneo a tipos de serviço distintos em processos diferentes; dentro de cada processo, a execução concorrente de tarefas similares pode ser implementada através de um ambiente de gerência cooperativa.

6.3 Programação orientada a eventos

A programação orientada a eventos é um exemplo paradigmático de um modelo de concorrência mono-tarefa. Como observamos anteriormente, esse modelo é bastante adequado a aplicações inerentemente assíncronas, e vem sendo utilizado há bastante tempo para a construção de sistemas de interface gráfica e, mais recentemente, para a implementação de servidores WEB de alto desempenho [47, 89, 16] e de aplicações distribuídas [86, 58, 11, 76].

Uma das maiores dificuldades na implementação desse tipo de modelo é decompor tratadores de eventos em uma ou mais partes, para impedir que tratamentos excessivamente longos comprometam a responsividade da aplicação. Essa decomposição pode ser necessária, por exemplo, para evitar o bloqueio de um tratador de eventos à espera de uma operação de entrada ou saída. Uma situação semelhante ocorre em aplicações distribuídas, quando o tratamento de um evento envolve uma chamada remota síncrona [58, 76]. Nos dois casos, é necessário suspender o tratamento do evento, retomando-o quando a resposta à operação solicitada (um novo evento) for recebida. Para que o tratamento de um evento possa ser suspenso e posteriormente retomado, é necessário algum mecanismo que permita preservar, e restaurar, o contexto de execução desse tratamento.

Lima [58] e Fuchs [30] mostram que continuações de primeira classe provêem suporte a esse tipo de mecanismo. O uso de continuações de primeira classe permite que o contexto de execução de um tratador de eventos — isto é, a *continuação* desse tratador — seja salvo em alguma variável, ou estrutura de dados, para que possa ser recuperado e restaurado quando o resultado que permite o prosseguimento do tratador é obtido.

Rossetto [76] mostra que co-rotinas completas assimétricas também provêem essa facilidade. O trabalho de Rossetto descreve a implementação de um ambiente de desenvolvimento para aplicações distribuídas com mobilidade. Esse ambiente utiliza para os componentes de uma aplicação distribuída um modelo de concorrência baseado em orientação a eventos; a interação entre esses componentes é realizada através de um espaço de tuplas [57]. Operações de escrita de tuplas representam tanto solicitações de serviço como respostas a essas solicitações; essas operações produzem eventos que são enviados aos componentes que registraram seu interesse nas tuplas correspondentes através de operações de leitura. Dessa forma, a inserção de uma tupla associada à solicitação de um determinado serviço,

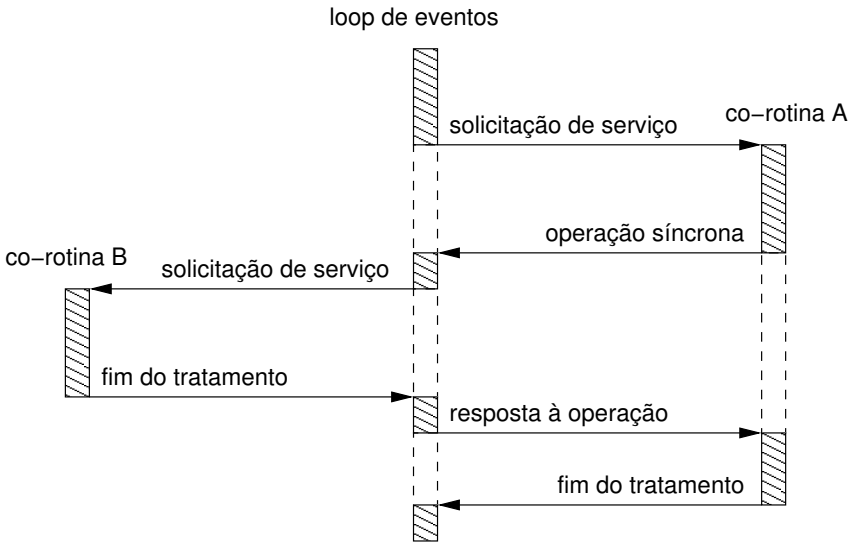

Figura 6.3: Orientação a eventos com suporte de co-rotinas

seguida pela leitura da tupla que contém o resultado desse serviço representa uma operação remota síncrona, e não deve bloquear um componente da aplicação.

Para permitir que tratadores de eventos sejam suspensos enquanto executam uma operação remota síncrona, a execução desses tratadores é realizada por co-rotinas completas assimétricas, como mostra a Figura 6.3. Quando um tratador solicita uma operação síncrona, a suspensão da co-rotina correspondente permite o retorno do controle ao *loop* de eventos, mantendo o estado do tratador. A recepção da tupla que contém o resultado da operação é associada a uma função de *callback* que reativa a co-rotina para que o tratador de eventos possa prosseguir sua execução. Caso seja necessário minimizar o custo de criação de tratadores de eventos, um *pool* de co-rotinas, como o descrito no Apêndice A, pode ser utilizado.

É importante observar que o uso de co-rotinas na implementação de aplicações orientadas a eventos não impõe um novo paradigma, ou estilo de programação. Nesse contexto, co-rotinas representam apenas uma facilidade conveniente para a implementação de tratadores de eventos que precisam ser decompostos, permitindo que o estado desses tratadores seja preservado e restaurado sem que seja necessário abrir mão de uma aparência de programação sequencial. O uso dessa facilidade é opcional, e não afeta a programação de tratadores de eventos que possam ser implementados por pequenos trechos de código ou por funções convencionais.

Um outro exemplo de combinação de processos, orientação a eventos e co-rotinas é a arquitetura proposta por Larus e Parkes [54] para a imple-

mentação de servidores em ambientes multi-processadores. Assim como na arquitetura SEDA, descrita na seção anterior, uma aplicação concorrente é decomposta em processos que representam uma sucessão de estágios. Cada estágio é responsável por um conjunto de operações assíncronas, e as solicitações e respostas a essas operações são enviadas através de mensagens. A distribuição de operações pelos diferentes estágios (denominada *cohort scheduling*) é feita de forma a agrupar computações que referenciam uma mesma região de código e dados, possibilitando ganhos de desempenho por executá-las em sequência e por reduzir a necessidade de sincronização. Dentro de cada estágio, a execução de operações é realizada de forma não preemptiva, porém uma operação pode ser suspensa — por exemplo, durante uma operação de entrada e saída — e retomada posteriormente. Os autores dessa arquitetura sugerem o uso de mecanismos como *fibers* para a manutenção do estado de operações suspensas. Contudo, não associam esse tipo de mecanismo ao conceito de co-rotinas, e o denominam de mecanismo de "continuações implícitas" (*implicit continuations*).

6.4 Co-rotinas versus threads

Neste capítulo, argumentamos que modelos de concorrência baseados em gerência cooperativa de tarefas e em orientação a eventos são alternativas vantajosas para o modelo de *multithreading*, permitindo o desenvolvimento de aplicações menos complexas, mais eficientes e muito mais fáceis de depurar. Discutimos também que em ambientes multi-processadores, a combinação de mecanismos de troca de mensagens com esses modelos pode substituir arquiteturas baseadas unicamente em *multithreading* com benefícios de maior simplicidade e melhor desempenho, apontando trabalhos que comprovam essa afirmação.

Podemos concluir, então, que para a maioria das aplicações concorrentes, seja em ambientes mono ou multi-processadores, o modelo de *multithreading* é inadequado, tanto com respeito à complexidade de programação quanto em relação a requisitos de correção e desempenho. À exceção de componentes de sistemas operacionais, a união de mecanismos de preempção e memória compartilhada pode ser conveniente apenas em cenários com requisitos rigorosos de justiça e responsividade, como aplicações de tempo real. Entretanto, a maioria das implementações de *multithreading* não oferece garantias reais com respeito a esses requisitos; em algumas implementações, como a de Java [56], a própria especificação do mecanismo de *multithreading*

não exige essas garantias. Dessa forma, mesmo no contexto de aplicações de tempo real, o uso de *multithreading* pode não oferecer vantagens.

Mostramos também neste capítulo que mecanismos de co-rotinas completas, além de permitir uma implementação trivial de gerência cooperativa de tarefas, provêem facilidades bastante convenientes para a implementação de aplicações orientadas a eventos. Podemos concluir, portanto, que uma linguagem que oferece co-rotinas completas não precisa oferecer *threads* ou qualquer outra construção adicional para prover um suporte básico adequado à programação concorrente independente de mecanismos do sistema operacional.

7
Conclusão

A ausência de uma definição formal para o conceito de co-rotinas e a diversidade de implementações de mecanismos de co-rotinas (algumas delas desnecessariamente complexas) impediram o reconhecimento da expressividade dessa construção de controle, e de sua conveniência como um recurso de programação.

Ao desenvolver este trabalho, buscamos alcançar um entendimento adequado do conceito de co-rotinas e de seu poder expressivo. Nossas principais contribuições são:

- a proposta de um novo sistema de classificação que permite distinguir diversas implementações de co-rotinas com respeito à sua conveniência e expressividade;
- a introdução do conceito de co-rotinas completas, e uma definição formal para esse conceito, baseada no desenvolvimento de uma semântica operacional;
- a demonstração da equivalência de poder expressivo entre co-rotinas completas simétricas e assimétricas e entre co-rotinas completas e continuações *one-shot*;
- uma discussão que fundamenta o argumento de que co-rotinas completas assimétricas são mais convenientes que co-rotinas completas simétricas para a implementação de diversos comportamentos de controle;
- uma comparação de modelos de concorrência com respeito a seus benefícios e desvantagens, justificando a adoção de modelos alternativos a *multithreading* e o oferecimento de co-rotinas como uma construção básica de concorrência adequada à implementação desses modelos alternativos.

A partir desses resultados, obtivemos também argumentos que fundamentam nossa defesa de co-rotinas como um conceito simples, adequado

a linguagens procedurais, que pode ser implementado com eficiência, e que pode substituir, com vantagens, tanto continuações de primeira classe quanto *threads*.

Atualmente, uma série de trabalhos vêm defendendo o uso de continuações de primeira classe para o desenvolvimento de aplicações WEB [72, 34, 26, 73]. A idéia básica dessas propostas é utilizar continuações de primeira classe para manter o estado de uma aplicação ao longo de um número arbitrário de interações com um cliente, permitindo que essa aplicação seja desenvolvida como uma aplicação sequencial convencional, ao invés de um conjunto de *scripts* isolados. O uso de continuações de primeira classe para a manutenção do estado das aplicações é, contudo, independente das implementações do modelo de concorrência para essas aplicações, que utilizam *threads*.

Acreditamos que o uso de co-rotinas nesse cenário é um campo de investigação bastante promissor. Além de prover um suporte conveniente para a manutenção do estado da aplicação WEB, preservando um estilo de programação sequencial, o uso de co-rotinas elimina a necessidade de construções de concorrência adicionais, como *threads*. Ao prover essas duas facilidades através de um único conceito, co-rotinas podem não apenas simplificar a estrutura da aplicação como também favorecer ganhos de desempenho.

Bibliografia

[1] ADYA, A.; HOWELL, J.; THEIMER, M.; BOLOSKY, W. J. ; DOUCER, J. R.. **Cooperative Task Management without Manual Stack Management**. In: PROCEEDINGS OF USENIX ANNUAL TECHNICAL CONFERENCE, Monterey, CA, June 2002. USENIX.

[2] ANDREWS, G. R.. **Foundations of Multithreaded, Parallel and Distributed Programming**. Addison-Wesley, 2000.

[3] ARCHER, T.; WHITECHAPEL, A.. **Inside C#**. Microsoft Press, 2002.

[4] ATKINSON, R.; LISKOV, B. ; SCHELFLER, R.. **Aspects of Implementing CLU**. In: PROCEEDINGS OF THE ACM 1978 ANNUAL CONFERENCE, p. 123–129, Oct. 1978.

[5] BEHREN, R.; CONDIT, J. ; BREWER, E.. **Why Events are a Bad Idea (for high-concurrency servers)**. In: PROCEEDINGS OF THE 9TH WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS (HOTOS IX), Lihue, HI, May 2003.

[6] BEN-ARI, M.. **Principles of Concurrent and Distributed Programming**. Benjamin Cummings, 1990.

[7] BENTON, N.; CARDELLI, L. ; FOURNET, C.. **Modern Concurrency Abstractions for C#**. In: PROCEEDINGS OF THE SIXTEENTH EUROPEAN CONFERENCE ON OO PROGRAMMING ECOOP 2002, Spain, June 2002.

[8] BIRTWISTLE, G.; DAHL, O.-J.; MYHRHAUG, B. ; NYGAARD, K.. **Simula Begin**. Studentlitteratur, Sweden, 1980.

[9] BIRREL, A. D.. **An introduction to programming with threads**. Technical report 35, Digital Systems Research Center, Jan. 1989.

[10] BRUGGEMAN, C.; WADDELL, O. ; DYBVIG, R.. **Representing Control in the Presence of One-Shot Continuations**. In: PROCEEDINGS OF THE ACM SIGPLAN'96 CONF. ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), p. 99–107, Philadelphia, PA, May 1996. ACM. SIGPLAN Notices 31(5).

[11] CARZANIGA, A.; ROSENBLUM, D. ; WOLF, A.. **Design and evaluations of a wide-area event notification service**. ACM Transactions on Computer Systems, 19(3):332–383, 2001.

[12] CLINGER, W.; HARTHEIMER, A. ; OST, E.. **Implementation strategies for continuations**. In: PROCEEDINGS OF THE 1988 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, p. 124–131, Snowbird, Utah, 1988.

[13] CLOCKSIN, W.; MELLISH, C.. **Programming in Prolog**. Springer-Verlag, 1981.

[14] CONWAY, M.. **Design of a separable transition-diagram compiler**. Communications of the ACM, 6(7):396–408, July 1963.

[15] CONWAY, D.. **RFC 31: Subroutines: Co-routines**, 2000. `http://dev.perl.org/perl6/rfc/31.html`.

[16] DABEK, F.; ZELDOVICH, N.; KAASHOEK, F.; MAZIERES, D. ; MORRIS, R.. **Event-driven Programming for Robust Software**. In: PROCEEDINGS OF THE 10TH ACM SIGOPS EUROPEAN WORKSHOP, Sept. 2002.

[17] DAHL, O.-J.; DIJKSTRA, E. W. ; HOARE, C. A. R.. **Hierarchical program structures**. In: STRUCTURED PROGRAMMING, p. 175–220. Academic Press, London, England, 1972.

[18] DANVY, O.; FILINSKI, A.. **Abstracting control**. In: PROCEEDINGS OF THE 1990 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, p. 151–160, Nice, France, June 1990. ACM.

[19] DIJKSTRA, E. W.. **Cooperating Sequential Processes**. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, Jan. 1965.

[20] DIJKSTRA, E. W.. **The Structure of the THE Multiprogramming System**. Communications of the ACM, 11(5):341–346, 1968.

[21] DYBVIG, R.; HIEB, R.. **Engines from continuations**. Computer Languages, 14(2):109–123, 1989.

[22] FELLEISEN, M.. **Transliterating Prolog into Scheme**. Technical Report 182, Indiana University, Bloomington, IN, 1985.

[23] FELLEISEN, M.; FRIEDMAN, D.. **Control operators, the secd-machine, and the λ-calculus**. In: Wirsing, M., editor, FORMAL DESCRIPTION OF PROGRAMMING CONCEPTS-III, p. 193–217. North-Holland, 1986.

[24] FELLEISEN, M.. **The theory and practice of first-class prompts**. In: PROCEEDINGS OF THE 15TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'88, p. 180–190, San Diego, CA, Jan. 1988. ACM.

[25] FELLEISEN, M.. **On the expressive power of programming languages**. In: PROCEEDINGS OF 3RD EUROPEAN SYMPOSIUM ON PROGRAMMING ESOP'90, p. 134–151, Copenhagen, Denmark, May 1990.

[26] FELLEISEN, M.. **Developing Interactive Web Programs**. In: ADVANCED FUNCTIONAL PROGRAMMING: 4TH INTERNATIONAL SCHOOL, AFP 2002, volumen 2638 de **Lecture Notes in Computer Science**, p. 100–128. Springer-Verlag, 2003.

[27] FRIEDMAN, D.; HAYNES, C. ; KOHLBECKER, E.. **Programming with continuations**. In: Pepper, P., editor, PROGRAM TRANSFORMATION AND PROGRAMMING ENVIRONMENTS, p. 263–274. Springer-Verlag, 1984.

[28] FRIEDMAN, D.; HAYNES, C.; KOHLBECKER, E. ; WAND, M.. **Scheme 84 Interim Reference Manual**. Technical report 153, Indiana University, Computer Science Department, 1985.

[29] FRIEDMAN, D.; WAND, M. ; HAYNES, C.. **Essentials of Programming Languages**. MIT Press, London, England, second edition, 2001.

[30] FUCHS, M.. **Escaping the event loop: an alternative control structure for multi-threaded GUIs**. In: Unger, C.; Bass, L., editors, Engineering for the HCI, p. 69–87. Chapman and Hall, 1996.

[31] GANZ, S.; FRIEDMAN, D. ; WAND, M.. **Trampolined Style**. In: PROCEEDINGS OF THE 4TH ACM SIGPLAN INTERNATIONAL

CONFERENCE ON FUNCTIONAL PROGRAMMING, p. 18–27, Paris, France, 1999.

[32] GELERNTER, D.; CARRIERO, N.. **Generative Communications in Linda**. ACM Transactions on Programming Languages and Systems, 7(1):80–112, 1985.

[33] GOLDBERG, A.; ROBSON, D.. **Smalltalk-80: the Language and its Implementation**. Addison-Wesley, 1983.

[34] GRAUNKE, P.; KRISHNAMURTI, S.; HOEVEN, S. V. D. ; FELLEISEN, M.. **Programming the Web with High-Level Programming Languages**. In: EUROPEAN SYMPOSIUM ON PROGRAMMING, ESOP'2001, Apr. 2001.

[35] GRISWOLD, R.. **The Evaluation of Expressions in Icom**. ACM Transactions on Programming Languages and Systems, 4(4):563–584, 1982.

[36] GRISWOLD, R.; GRISWOLD, M.. **The Icon Programming Language**. Prentice-Hall, New Jersey, NJ, 1983. 3rd Edition, Peer to Peer Communications, 2000.

[37] GRUNE, D.. **A View of Coroutines**. ACM SIGPLAN Notices, 12(7):75–81, 1977.

[38] HANSEN, P. B.. **The Nucleus of a Multiprogramming System**. Communications of the ACM, 13(4):238–241 and 250, 1970.

[39] HANSEN, P. B.. **Operating System Principles**. Prentice-Hall, 1973.

[40] HARPER, R.; DUBA, B.; HARPER, R. ; MACQUEEN, D.. **Typing first-class continuations in ML**. In: PROCEEDINGS OF THE 18TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'91, p. 163–173, Orlando, FL, Jan. 1991. ACM.

[41] HAYNES, C. T.; FRIEDMAN, D. ; WAND, M.. **Obtaining coroutines with continuations**. Computer Languages, 11(3/4):143–153, 1986.

[42] HAYNES, C. T.. **Logic continuations**. J. Logic Programming, 4:157–176, 1987.

[43] HELSGAUN, K.. **A Portable C++ Library for Coroutine Sequencing**. Writings on Computer Science, Department of Computer Science, Roskilde University, Roskilde, Denmark, 1999.

[44] HEWITT, C.. **Viewing control structures as patterns of passing messages**. Artificial Intelligence, 8, 1977.

[45] HIEB, R.; DYBVIG, R. ; BRUGGEMAN, C.. **Representing Control in the Presence of First-Class Continuations**. In: PROCEEDINGS OF THE ACM SIGPLAN'90 CONF. ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), p. 66–77, White Plains, NY, June 1990. ACM. SIGPLAN Notices 25(6).

[46] HIEB, R.; DYBVIG, R. ; ANDERSON III, C. W.. **Subcontinuations**. Lisp and Symbolic Computation, 7(1):83–110, 1994.

[47] HU, J.; PYARALI, I. ; SCHMIDT, D. C.. **Applying the Proactor Pattern to High-Performance Web Servers**. In: PROCEEDINGS OF THE 10TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, IASTED, Las Vegas, Nevada, Oct. 1998.

[48] IERUSALIMSCHY, R.; FIGUEIREDO, L. ; CELES, W.. **Lua — an extensible extension language**. Software: Practice & Experience, 26(6):635–652, June 1996.

[49] IERUSALIMSCHY, R.. **Programming in Lua**. Lua.org, ISBN 85-903798-1-7, Rio de Janeiro, Brazil, 2003.

[50] JOHNSON, G.; DUGGAN, D.. **Stores and partial continuations as first-class objects in a language and its environment**. In: PROCEEDINGS OF THE 15TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'88, San Diego, CA, Jan. 1988. ACM.

[51] KELSEY, R.; CLINGER, W. ; REES, J.. **Revised^5 report on the algorithmic language Scheme**. ACM SIGPLAN Notices, 33(9):26–76, Sept. 1998.

[52] KNUTH, D. E.. **The Art of Computer Programming, Volume 1, Fundamental Algorithms**. Addison-Wesley, Reading, MA, 1968. 3rd Edition, 1997.

[53] KUMAR, S.; BRUGGEMAN, C. ; DYBVIG, R.. **Threads yield continuations**. Lisp and Symbolic Computation, 10(3):223–236, 1998.

[54] LARUS, J.; PARKES, M.. **Using Cohort Scheduling to Enhance Server Performance**. In: USENIX ANNUAL TECHNICAL CONFERENCE, June 2002.

[55] LAUER, H. C.; NEEDHAM, R. M.. **On the Duality of Operating System Structures**. In: PROC. SECOND INTERNATIONAL SYMPOSIUM ON OPERATING SYSTEMS, IRIA, Oct. 1978. Reprinted in Operating Systems Review, 13(2), pages 3–19, 1979.

[56] LEA, D.. **Concurrent Programming in Java, Design Principles and Patterns**. Addison-Wesley, Reading, MA, second edition, 2000.

[57] LEAL, M.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **LuaTS - a reactive event-driven tuple space**. J.UCS - Journal of Universal Computer Science, 9(8):730–744, 2003.

[58] DE LIMA, M. J. D.. **ORFEO: Programação Distribuída Orientada a Eventos com Funções e Continuações como Valores de Primeira Classe**. PhD thesis, Departamento de Informática, PUC–Rio, 2000.

[59] LISKOV, B.; SNYDER, A.; ATKINSON, R. ; SCHAFFERT, C.. **Abstraction mechanisms in CLU**. Communications of the ACM, 20(8):564–576, Aug. 1977.

[60] LISKOV, B.; MOSS, E.; SNYDER, A.; ATKINSON, R.; SCHAFFERT, C.; BLOOM, T. ; SCHEIFLER, R.. **CLU Reference Manual**. Springer-Verlag, New York, NY, 1984.

[61] MADSEN, O.; M.-PEDERSEN, B. ; NYGAARD, K.. **Object-oriented programing in the BETA programming language**. ACM Press/Addison-Wesley, New York, NY, 1993.

[62] MARLIN, C. D.. **Coroutines: A Programming Methodology, a Language Design and an Implementation**. LNCS 95, Springer-Verlag, 1980.

[63] MARTELLI, A.. **Python in a Nutshell**. O'Reilly, 2003.

[64] MOODY, K.; RICHARDS, M.. **A coroutine mechanism for BCPL**. Software: Practice & Experience, 10(10):765–771, Oct. 1980.

[65] MOURA, A. L.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Coroutines in Lua**. In: 8TH BRAZILIAN SYMPOSIUM OF PROGAMMING LANGUAGES (SBLP), p. 89–101, Niteroi, RJ, Brazil, May 2004. SBC.

[66] MURER, S.; OMOHUNDRO, S.; STOUTAMIRE, D. ; SZYPERSKI, C.. **Iteration abstraction in Sather**. ACM Transactions on Progamming Languages and Systems, 18(1):1–15, Jan. 1996.

[67] NICHOLS, B.; BUTTLAR, D. ; FARRELL, J. P.. **Pthreads Programming**. O'Reilly & Associates, 1996.

[68] OUSTERHOUT, J.. **Why threads are a bad idea (for most purposes)**. In: USENIX TECHNICAL CONFERENCE, Austin, Texas, Jan. 1996. Invited Talk.

[69] PAULI, W.; SOFFA, M. L.. **Coroutine behaviour and implementation**. Software: Practice & Experience, 10(3):189–204, Mar. 1980.

[70] QUEINNEC, C.; SERPETTE, B.. **A dynamic extent control operator for partial continuations**. In: PROCEEDINGS OF THE 18TH ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES POPL'91, p. 174–184, Orlando, FL, Jan. 1991. ACM.

[71] QUEINNEC, C.. **A library of higher-level control operators**. ACM SIGPLAN Lisp Pointers, 6(4):11–26, Oct. 1993.

[72] QUEINNEC, C.. **The influence of browsers on evaluators, or continuations to program web servers**. In: INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING ICFP'2000, p. 23–33, Montreal, Canada, Sept. 2000.

[73] QUEINNEC, C.. **Inverting back the inversion of control, or Continuations versus page-centric programming**. ACM SIGPLAN Notices, 38(2):57–64, Feb. 2003.

[74] REYNOLDS, J. C.. **The Discoveries of Continuations**. Lisp and Symbolic Computation, 6(3/4):233–247, 1993.

[75] RICHTER, J.. **Advanced Windows**. Microsoft Press, Redmond, WA, third edition, 1997.

[76] ROSSETTO, S.; RODRIGUEZ, N. ; IERUSALIMSCHY, R.. **Abstrações para o desenvolvimento de aplicações distribuídas em**

ambientes com mobilidade. In: 8TH BRAZILIAN SYMPOSIUM OF PROGAMMING LANGUAGES (SBLP), p. 143–156, Niteroi, RJ, Brazil, May 2004. SBC.

[77] SCHMIDT, D.; STAL, M.; ROHNERT, H. ; BUSCHMANN, F.. **Pattern-Oriented Software Architecture, volume 2, Patterns for Concurrent and Networked Objects**. John Wiley & Sons, 2000.

[78] SCHEMENAUER, N.; PETERS, T. ; HETLAND, M.. **PEP 255 Simple Generators**, 2001. `http://www.python.org/peps/pep-0255.html`.

[79] SITARAM, D.. **Handling control**. In: PROCEEDINGS OF THE ACM SIGPLAN'93 CONF. ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI), Albuquerque, NM, June 1993. ACM. SIGPLAN Notices 28(6).

[80] SITARAM, D.. **Models of Control and Their Implications for Progamming Language Design**. PhD thesis, Rice University, Apr. 1994.

[81] SPRINGER, G.; FRIEDMAN, D. P.. **Scheme and The Art of Programming**. McGraw–Hill, 1994.

[82] STRACHEY, C.; WADSWORTH, C.. **Continuations: a Mathematical Semantics for Handling Full Jumps**. Higher-Order and Symbolic Computation, 13(1/2):135–152, 2000.

[83] THOMAS, D.; HUNT, A.. **Programming Ruby: The Pragmatic Programmer's Guide**. Addison-Wesley, 2001.

[84] TISMER, C.. **Continuations and Stackless Python**. In: PROCEEDINGS OF THE 8TH INTERNATIONAL PYTHON CONFERENCE, Arlington, VA, Jan. 2000.

[85] TOERNIG, E.. **C Coroutines (coro library)**, 2000. `http://www.goron.de/~froese/coro/coro.html`.

[86] URURAHY, C.; RODRIGUEZ, N.. **ALua: An event-driven communication mechanism for parallel and distributed programming**. In: PROC. 12TH INTL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS (PDCS'99), Fort Lauderdale, Florida, 1999.

[87] WALL, L.; CHRISTIANSEN, T. ; ORWANT, J.. **Programming Perl**. O'Reilly, third edition, 2000.

[88] WAND, M.. **Continuation-based multiprocessing**. In: PROCEEDINGS OF THE 1980 LISP CONFERENCE, p. 19–28, Stanford, CA, Aug. 1980. ACM.

[89] WELSH, M.; CULLER, D. ; BREWER, E.. **SEDA: An Architecture for Well-Conditioned, Scalable Internet Services**. In: PROCEEDINGS OF THE EIGHTEENTH SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP-18), Canada, Oct. 2001.

[90] WIRTH, N.. **Programming in Modula-2**. Springer-Verlag, third, corrected edition, 1985.

[91] ERIGHTS.ORG. **Overview: Concurrency in E**, 2003. `http://www.erights.org/elib/concurrency/overview.html`.

A
Gerência cooperativa de tarefas com co-rotinas completas assimétricas

No Capítulo 6, mostramos uma implementação bastante simples de um ambiente de gerência cooperativa de tarefas baseado em co-rotinas completas assimétricas (Figura 6.1). O objetivo deste apêndice é complementar essa implementação, adicionando facilidades de sincronização e mecanismos que permitem obter um melhor desempenho.

A primeira modificação ao nosso ambiente é a adição de um *pool* de co-rotinas reutilizáveis para minimizar o custo de criação de novas tarefas. Nessa nova implementação, apresentada na Figura A.1, a tabela `pool` armazena as co-rotinas disponíveis para a execução de uma nova tarefa. A função `create_task` somente cria uma nova co-rotina se esse *pool* está vazio; caso contrário, uma co-rotina disponível é removida do *pool*. Ao final da execução de sua tarefa, uma co-rotina se suspende, ao invés de terminar; em sua próxima reativação (invocada por `create_task`), a co-rotina recebe a próxima tarefa a executar. A nova função `release_task` é chamada pelo escalonador quando uma co-rotina sinaliza o término de sua tarefa. Se o *pool* está em seu tamanho máximo, essa co-rotina é descartada. Se não, a referência para essa co-rotina (suspensa à espera de uma nova tarefa) é salva no *pool* de co-rotinas.

Nossa segunda modificação evita o bloqueio de uma aplicação quando uma tarefa solicita uma operação de entrada ou saída. Para isso, assumimos que essas solicitações são realizadas através de funções como a apresentada na Figura 6.2. Nesse caso, quando uma operação não se completa num tempo pré-determinado, a co-rotina em execução é suspensa, retornando ao escalonador uma referência para o recurso correspondente à operação invocada. A referência para o recurso é salva em uma tabela, para que o escalonador, quando possível, possar aguardar uma mudança de estado. Quando todas as tarefas vivas estão à espera de operações de entrada ou saída, o escalonador se bloqueia, invocando uma função auxiliar (`select`) que permite aguardar uma mudança de estado em algum dos recursos

```
MAX_POOL = ...  -- tamanho máximo do pool
pool  = {}      -- pool de co-rotinas
tasks = {}      -- lista de tarefas vivas

-- cria uma tarefa
function create_task(f)
  local co
  if table.getn(pool) == 0 then -- pool vazio
    co = coroutine.wrap(
           function()
             while true do
               -- executa tarefa
               f()
               -- suspende execução aguardando nova tarefa
               f = coroutine.yield("task ended")
               -- aguarda ativação pelo escalonador
               coroutine.yield()
             end
           end)
  else
    co = table.remove(pool)  -- retira co-rotina do pool
    co(f)                    -- e envia nova tarefa
  end
  table.insert(tasks, co)
end

-- libera uma co-rotina
function release_task(co)
  if table.getn(pool) < MAX_POOL then
    table.insert(pool, co)
  end
end

-- escalonador
function dispatcher()
  while true do
    local n = table.getn(tasks)
    if n == 0 then break end
    for i = 1, n do
      local res = tasks[i]()      -- reativa tarefa
      if res == "task ended" then -- tarefa terminou
        release_task(tasks[i])    -- libera co-rotina
        table.remove(tasks, i)    -- remove tarefa
        break
      end
    end
  end
end
```

Figura A.1: Gerência cooperativa de tarefas com pool de co-rotinas

```
function dispatcher()
  while true do
    local n = table.getn(tasks)
    if n == 0 then break end
    resources = {}                -- tabela de recursos
    for i = 1, n do
      local res = tasks[i]()
      if res == "task ended" then -- tarefa terminou
        release_task(tasks[i])
        table.remove(tasks, i)
        break
      elseif res ~= nil then      -- tarefa aguarda E/S
        table.insert(resources, res)
      end
    end

    -- todas as tarefas vivas aguardam E/S ?
    if table.getn(resources) == table.getn(tasks) then
      select(resources)
    end
  end
end
```

Figura A.2: Gerência cooperativa de tarefas com E/S não bloqueante

referenciados na tabela de recursos. A Figura A.2 mostra a adição desses procedimentos ao escalonador.

Para completar nosso ambiente de gerência cooperativa de tarefas, podemos adicionar alguns mecanismos básicos de sincronização. Em primeiro lugar, implementamos um mecanismo de exclusão mútua baseado no conceito de semáforos binários [19, 6]. Nessa implementação, apresentada na Figura A.3, um semáforo é representado por uma tabela que armazena o valor s do semáforo (0 ou 1) e uma tabela de co-rotinas que representa a fila de tarefas que aguardam a liberação do semáforo. A operação *P(s)* é implementada por uma função que suspende a co-rotina ativa caso o semáforo não esteja liberado, retornando ao escalonador um valor que indica que a co-rotina deve ser inserida na fila do semáforo especificado, e removida da lista de tarefas vivas. Caso contrário, o valor do semáforo é alterado, e a co-rotina prossegue sua execução. A operação *V(s)* é implementada por uma função que, após atualizar o valor do semáforo, verifica se existem tarefas que o aguardam, consultando a fila de espera correspondente. Se essa fila não está vazia, a primeira co-rotina é removida da fila de espera e reinserida na lista de tarefas vivas.

Essa implementação pode causar uma distribuição injusta de recursos,

```
-- cria um semáforo
function create_sem() return {s = 0, blocked = {}} end

-- aguarda um semáforo
function P(sem)
  while sem.s > 0 do coroutine.yield("blocked", sem) end
  sem.s = 1
end

-- libera um semáforo
function V(sem)
  sem.s = 0
  local queue = sem.blocked          -- fila de espera
  if table.getn(queue) > 0 then
    local c = table.remove(queue,1) -- remove tarefa da fila
    table.insert(tasks,c)           -- e a reinsere na lista
  end                               -- de tarefas vivas
end
```

Figura A.3: Implementação de um semáforo binário

ou mesmo situações de *starvation*, se quando um semáforo é liberado, a co-rotina removida da fila de espera for colocada no final da lista de tarefas vivas. Soluções que tentem evitar esse tipo de problema podem ser desenvolvidas, por exemplo, colocando-se a co-rotina no início da lista de tarefas vivas, e/ou suspendendo a co-rotina que libera um semáforo, colocando-a no final da lista de tarefas. Entretanto, a necessidade de mecanismos de sincronização em aplicações que utilizam um modelo de gerência cooperativa é significativamente menor do que em ambientes que envolvem preempção. Para a maioria das aplicações, o cuidado em evitar situações como a descrita é desnecessário.

Semáforos contadores são tipicamente utilizados quando é necessário limitar o número de tarefas que compartilham, simultaneamente, um dado recurso. A implementação de um semáforo contador é bastante semelhante à de um semáforo binário. As modificações necessárias são a atribuição de um valor inicial ao semáforo (correspondente ao número máximo de tarefas que podem obtê-lo), o decremento desse valor cada vez que uma tarefa obtém o semáforo, o bloqueio de tarefas quando esse valor é 0, e o incremento do valor de um semáforo quando uma tarefa o libera.

Mecanismos de sincronização por condições também têm uma implementação trivial, apresentada na Figura A.4. Para representar uma condição, é suficiente uma tabela que representa a fila de tarefas bloqueadas. A espera por uma condição pode ser implementada por uma função que

```
-- cria uma condição
function create_cond() return {blocked = {}} end

-- aguarda uma condição
function wait(cond)
  coroutine.yield("blocked", cond)
end

-- sinalização da condição (libera uma tarefa apenas)
function signal(cond)
  local queue = cond.blocked
  if table.getn(queue) > 0 then
    local c = table.remove(queue,1)
    table.insert(tasks,c)
  end
end

-- sinalização da condição (libera todas as tarefas)
function signal_all(cond)
  local queue = cond.blocked
  for i = 1, table.getn(queue) do
    table.insert(tasks, queue[i])
  end
  cond.blocked = {}
end
```

Figura A.4: Implementação de sincronização por condições

suspende a co-rotina ativa, retornando ao escalonador um valor que indica que a co-rotina deve ser inserida na fila da condição especificada, e removida da lista de tarefas (ou seja, o mesmo procedimento implementado para aguardar um semáforo). As operações de sinalização removem uma, ou todas, as co-rotinas bloqueadas da fila da condição correspondente, reinserindo-a(s) na lista de tarefas. Na ausência de preempção, mecanismos de exclusão mútua, utilizados em implementações tradicionais de monitores em ambientes de *multithreading*, não são necessários.

A Figura A.5 apresenta a versão final do escalonador do nosso ambiente de gerência cooperativa de tarefas. Essa versão incorpora o suporte a operações de entrada e saída não bloqueantes, o uso de um *pool* de co-rotinas e o bloqueio de tarefas à espera de uma condição ou de um semáforo.

```
function dispatcher()
  while true do
    local n = table.getn(tasks)
    if n == 0 then break end
    resources = {}                -- tabela de recursos
    for i = 1, n do
      task = tasks[i]
      local res,s = task()        -- reativa tarefa
      if res == "task ended" then
        table.remove(tasks, i)    -- tarefa terminou
        break
      elseif res == "blocked" then
        table.remove(tasks, i)    -- tarefa bloqueada
        table.insert(s.blocked, task)
        break
      elseif res ~= nil then      -- tarefa aguarda E/S
        table.insert(resources, res)
      end
    end

    -- todas as tarefas vivas aguardam E/S ?
    if table.getn(resources) == table.getn(tasks) then
      select(resources)
    end
  end
end
```

Figura A.5: Gerência cooperativa de tarefas com sincronização