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*.