Revisitando Co-rotinas

Roberto Ierusalimschy, PUC-Rio

Os Primórdios

- 1963 — Trabalho pioneiro de Conway propõe co-rotinas para estruturar uma compilador multi-fases
  - segundo Knuth, o trabalho original é de 1958
  - arquitetura produtor–consumidor
- 1967 — Simula 67 incorpora mecanismos de co-rotinas
  - uso dirigido para concorrência colaborativa (área de simulação)
  - mecanismo bastante complexo
- 1968 — TAOCP (Knuth) descreve co-rotinas
  - descrição vaga
  - usos mais variados: produtor–consumidor, simulação, iteradores

DÉCADA DE 70

- Expansão de co-rotinas

- Vários artigos propondo novos usos
  - backtracking, iteradores, concorrência, etc.

- Algumas linguagens incorporam co-rotinas
  - Modula-2: co-rotinas para implementar multithreading
  - Primeiras versões de Icon: *generators*
  - CLU: *iterators*
  - BCPL: segue modelo (confuso) de Simula

O Ápice

- 1980 — *Coroutines: A Programming Methodology, a Language Design and an Implementation*
  - tese de doutorado de Christopher Marlin
  - trabalho “definitivo” sobre co-rotinas (até hoje)
- Proposta muito complicada
  - segue modelo de Simula
- Definição fraca para co-rotinas
  - “the values of data local to a coroutine persist between successive calls”
  - “the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage”
- Diversos mecanismos com diferentes poderes expressivos convivem sob o mesmo rótulo

O QUE FALTOU DIZER

- Se co-rotinas são valores de primeira classe
- Se co-rotinas são *stackful*
- Se co-rotinas são simétricas ou assimétricas

Co-rotinas como Valores de Primeira Classe

- Linguagem permite livre manipulação de co-rotinas?
- Co-rotinas podem ser invocadas em qualquer ordem? De qualquer lugar?
- *Generators* em Icon e *iterators* em CLU não são valores de primeira classe
- *Co-expressions* em Icon, *coroutines* em Modula-2 e *generators* em Python são valores de primeira classe

CO-ROTINAS *Stackful*

- Uma co-rotina pode suspender sua execução dentro de um nível arbitrário de chamadas de função?

- A implementação de co-rotina dada em TAOCP e *generators* em Python não são *stackful*

- *Iterators* em CLU, *generators* em Icon e *coroutines* em Modula-2 são *stackful*

CO-ROTINAS SIMÉTRICAS × ASSIMÉTRICAS

- Co-rotinas simétricas possuem um único comando (*transfer*) para passar o controle entre co-rotinas
  - Por exemplo, co-rotinas em Modula 2 são simétricas
- Co-rotinas assimétricas possuem dois comandos (*resume* e *yield*) para passar o controle entre co-rotinas
  - *resume* re-inicia uma co-rotina suspensa, retornando o controle para o ponto onde ela foi suspensa (uma chamada a *yield*)
  - *yield* suspende uma co-rotina em execução, retornando o controle para o *resume* correspondente
- Essa distinção entre co-rotinas é normalmente reconhecida, mas suas consequências são erroneamente entendidas
  - O termo “semi-corotinas”, as vezes usado para co-rotinas assimétricas, contribui para a confusão

Comparação entre Co-rotinas

- Co-rotinas de primeira classe são mais expressivas
- Co-rotinas *stackful* são mais expressivas
- Entre co-rotinas *stackful* de primeira classe, as simétricas e as assimétricas têm a mesma expressividade!
  - dada uma, podemos implementar a outra como uma pequena biblioteca
  - como são comuns implementações de co-rotinas assimétricas que não são *stackful* e/ou de primeira classe, muitas pessoas erroneamente atribuem a fraqueza a todas co-rotinas assimétricas
- Chamamos co-rotinas *stackful* de primeira classe de *co-rotinas completas*

DÉCADA DE 80

- Surgimento de linguagens com *continuações de primeira classe*
- Conceito muito elegante e muito expressivo
  - co-rotinas são frequentemente vistas como menos expressivas do que realmente são (e.g., backtracking)
- Conceito “simples”
  - para os teóricos, pelo menos
  - co-rotinas frequentemente vistas como muito complexas (devido ao modelo Simula)
- Co-rotinas viram caso particular de continuações

DÉCADA DE 90

- Disseminação de Multithreading
  - Java e *pthreads*
- Conceito “simples”
  - para os práticos, pelo menos
- Co-rotinas viram caso particular de multithreading

DÉCADA DE 00

- Algumas linguagens começam a buscar alternativas para continuações
  - Por exemplo, Python descarta continuações (e co-rotinas!) e implementa *generators*
- Vários sistemas começam a buscar alternativas para multithreading
  - *event-driven programming*
  - *fibers*, *lightweight threads*
- Linguagens com continuações (p.e., Scheme) enfrentam dificuldades em compatibilizar continuações com multithreading
- Co-rotinas podem substituir tanto continuações quanto multithreading com um conceito único e simples

Co-rotinas × Continuações

- Co-rotinas são tão expressivas quanto continuações *one-shot*!
  - em nossa bibliografia, não encontramos nenhuma aplicação real de continuações que não fosse trivialmente reescrita como *one-shot*
- Co-rotinas tem uma implementação muito mais simples que continuações
  - ou muito mais eficiente
- Co-rotinas podem ter uma formalização tão simples quanto continuações

CO-ROTINAS × MULTITHREADING

- Co-rotinas são muito mais eficientes que threads
  - poucas máquinas aguentam mil threads, mas é fácil aguentar centenas de milhares de co-rotinas

- Co-rotinas são não preemptivas
  - “fairness” fica mais difícil
  - mas corretude em geral fica muito mais fácil (não há *race conditions*)

CONCLUSÕES

- Co-rotinas completas são tão poderosas quanto continuações *one-shot*
- Co-rotinas completas podem substituir multithreading
- Co-rotinas podem ser simples de descrever formal e informalmente
- Co-rotinas têm uma implementação simples e eficiente
- Lua 5.0 oferece co-rotinas completas assimétricas