Pràctica CAP Q1 curs 2017-18
Fils cooperatius

Aarón Jiménez García
2017/2018

Índex

Part A

Funcionament de les continuacions a Smalltalk

A Pharo disposem d'un mètode anomenat callcc que ens permet fer ús de les continuacions. Tot i que aquest mètode no es troba dins del package de Kernel podem afegir-ho del package de KernelTests.

Callcc té un argument que és un bloc amb un paràmetre de la forma: [:cc | ...].
La idea és capturar el context actual en una instància de Continuation i passar-la com a paràmetre al bloc quan s'avalüi. D'aquesta manera, si volem aprofitar la continuació més endavant en el programa haurem de assignar el paràmetre a una variable externa del bloc i d'aquesta podrem tornar al context guardat cridant a la continuació de la forma següent: continuacio value: x on x és qualsevol cosa que vulguem que es retorni a la línia on haviem guardat el context.

Un exemple on podem veure clar l'ús de les continuacions a Pharo seria el següent:

```
1: | kont value |
2: value := Continuation callcc: [ :cc | kont := cc. 0 ].
3: Transcript show: value.
4: (value = 5)
5:     ifTrue: [ Transcript show: 'Ha arribat a 5 gracies a la continuacio' ]
6:     ifFalse: [ kont value: value + 1 ]
```

Aquest codi el que fa es sumar un a una variable value fins que arribi a cinc. Això ho pot fer ja que cada vegada que no es compleix que el valor sigui cinc torna a la línia dos on haviem demanat la continuació i value pasa a tenir un valor més. Com que la variable kont continua tenint la continuació podem continuar tornant fins que value tingui valor cinc.

Funcionament de les continuacions a Rhino

A Rhino disposem de les continuacions. Una continuació a Rhino és un objecte JavaScript que representa una captura de l'estat d'execució d'un script, per exemple la seva pila d'execució. L'intèrpret ens permet guardar-les en variables per poder utilitzar-les quan nosaltres vulguem durant el flux del programa i retornar-les. Quan volguem restaurar el context, haurem de cridar la continuació ja que aquesta és una funció. Si li passem un valor, a la línia on tornem del flux del programa s'hi retornarà l'argument, sino, es retornarà un undefined.

Per a capturar una continuació a Rhino hem de cridar a:

```
new Continuation()
```

És necessari fer la crida dins d'una funció ja que si no ho fèsim, quan cridessim a la continuació, al estar al scope global del script faria que acabés l'execució d'aquest mateix. Per exemple, el següent codi faria que l'script acabés al cridar a la continuació:

```
1: var finalitza = new Continuation();
2:
3: function foo(finalitza) {
4:     print("entra a la funcio i cridem a la continuacio");
5:     finalitza();
6:     print("no arribara mai");
7: }
8:
9: foo(finalitza);
```

Per tant, realitzant l'exemple que haviem fet en l'apartat anterior per a Pharo amb l'ús de les continuacions a Rhino seria:

```
1: function current_continuation() {
2:     print("Agafem la continuacio");
3:     return new Continuation();
4: }
5:
6: var value = 0,
7:     kont = current_continuation();
8:
9: print(value);
10:if (value === 5)
11:    print("Ha arribat a 5 gracies a la continuacio");
12:else {
13:    value++;
14:    kont(kont);
15:}
```

En aquest cas, en comptes de pasar com a parametre a la continuació la variable value més 1 pasem la mateixa continuació, d'aquesta forma podem tornar a utilitzar-la. Com que value es global a JavaScript no es captura el seu context lèxic en la continuació com a Smalltalk on es una variable local.

Implementació a Pharo de la continuació equivalent a Rhino

Aprofitant el codi de continuacions que ja té SmallTalk i que el podem veure, la implementació de les continuacions equivalent al JavaScript de Rhino és ràpida.

La nova funcionalitat s'en diu continuation i el seu codi és el següent:

```
1: continuation
2:    ^self current
```

Com veiem l'únic que fa es delegar la responsabilitat al mètode current. El mètode current és similar al currentDo que feiem servir amb callcc on es crida a from context amb el context a guardar, la diferència es que current no utilitza cap bloc i per tant el seu valor retornat no es el resultat del bloc

sinò que serà la continuació directament. Per tornar al context salvat anteriorment amb la continuació farem el mateix que amb callcc.

D'aquesta manera tenim un mètode que actua igual que les continuacions de Rhino, retornant una instància de Continuation i que quan volem tornar al context salvat li passem al mètode value: allò que vulguem retornar.

Un exemple d'ús d'aquest mètode que hem creat és el següent:

```
1: | kont |
2: kont := Continuation continuation.
3: Transcript show: kont; cr.
4: (kont isMemberOf: SmallInteger) ifFalse: [kont value: 2]
5:                                 ifTrue: [Transcript show: 'Ha fet continuacio'; cr.]
```

Aquest codi el que fa es agafar una instància de continuació i comprobar si la variable kont es de tipus SmallInteger, es clar que en un principi no ho serà ja que es de tipus Continuation, ho podem veure al Transcript. Si es fals recuperaria el context salvat i retornaria un 2 a la línia 2. Ara com es compleix la condició de ser un SmallInteger realitzarà el ifTrue imprimint al Transcript.

Implementació a JavaScript de la funció callcc(f)

Com hem comentat, a Pharo el bloc que passem a callcc cal que tingui un paràmetre que s'utilitzarà per a guardar la continuació. A JavaScript un bloc es una funció, per tant, aquest paràmetre se li pasarà com a paràmetre. Aquest paràmetre el que farem serà assignar-li la continuació i després retornarem el valor d'executar el bloc, es a dir, de la funció.

La implementació de la funció callcc, per tant, quedaria de la següent forma:

```
1: function callcc(foo) {
2:     var k = new Continuation;
3:     return foo(k);
4: }
```

Un exemple d'ús d'aquest mètode seria el següent:

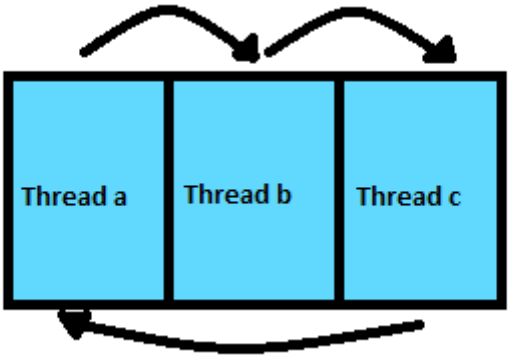
```
1: function callcc(foo) {
2:     var k = new Continuation;
3:     return foo(k);
4: }
5: var cont;
6: var f = function(k) {
7:     cont = k;
8:     return 2;
9: }
10: var result = callcc(f);
11: print(result);
12: if (result === 2) cont(3);
13: else print("Ha fet continuacio");
```

Com podem veure el nostre bloc és ben senzill, se li assigna a la variable cont el paràmetre que li pasen que serà la continuació i el resultat del bloc simplement és retornar 2. Aquest bloc que esta assignat a la variable f es passa com a paràmetre a la funció callcc que simula la funció callcc de Pharo i result acaba obtenint el 2 que retorna el bloc. A partir d'aquí comprobem si la variable result té valor 2, com és cert cridem a la continuació i li passem un 3. D'aquesta forma restaurem el context i tornem a la línia 10 on result ara tindrà el 3 que li havíem passat a la continuació. Ara, al contrari que abans, entrarem al else on s'imprimirà per pantalla.

Part B

Introducció a la pràctica

La pràctica consisteix en la realització de threads d'usuari, en concret, cooperative multithreading, es a dir, cada thread deixarà el processador a altres threads de manera voluntària. Estudiant el codi proporcionat a la pràctica i la sortida esperada, hem vist que el canvi de threads al cedir el control es cíclic com es pot veure a la següent figura.



Gràfic 1: Execució cíclica

De igual forma ho hem tingut present a la pràctica i ho hem implementat d'aquesta manera, de forma que si el thread b finalitza serà el thread c qui s'executarà a continuació i no l'a.

Per últim, comentar que juntament amb aquest document s'adjunta el programa anomenat filsCooperatius.js i els diferents jocs de proves amb nom JocProvaX.txt, on X és el número del joc de prova, que el grup ha realitzat per comprovar que la implementació és correcta.

A més a més, al fitxer filsCooperatius.js ve inclòs el programa exemple que es proporcionava a la pràctica i conté comentaris per a entendre com funciona el codi. Per provar els jocs de proves només cal comentar el programa exemple i enganxar el codi del joc de prova. A més per a cada joc de prova s'inclou en el mateix fitxer que el joc la sortida esperada.

Per compilar la pràctica cal tenir Rhino instal·lat i executar la següent comanda:

```
java -cp pathJar/x.jar org.mozilla.javascript.tools.shell.Main -opt -2
pathPractica/filsCooperatius.js
```

On pathJar és el path on es troba el jar per a la compilació de JavaScript, x el nom del jar i pathPractica el path on es troba la pràctica.

Explicació del codi

El codi consisteix en un objecte anomenat Scheduler que conté al seu prototipus quatre funcions, spawn, quit, relinquish i start_threads on s'interactuarà amb les propietats que té l'objecte. A més a

més hi han dues funcions una anomenada current_context que ens permetra obtenir el context del thread per guardarlo i utilitzarlo posteriorment i la funció make_thread_system que ens retornarà un objecte Scheduler.

A continuació explicarem pas per pas com funciona tot:

Funció make_thread_system

```
1: function make_thread_system() {
2:     return new Scheduler();
3: }
```

Aquesta funció ens retornarà un objecte de tipus Scheduler que veurem més endavant i que ens servirà per utilitzar l'API de threads que hem dissenyat.

Funció current_context

```
1: function current_context() {
2:     return new Continuation;
3: }
```

La funció current_context és molt important ja que ens retornarà un objecte de tipus Continuation que es guardaran i posteriorment ens seran útils per restaurar els contextos dels threads i que puguin continuar la seva execució.

Objecte Scheduler

L'objecte Scheduler és el nucli de tota la pràctica. Com es pot imaginar aquest nom està posat a consciència ja que un scheduler s'encarrega de controlar quin proces s'executa. En el nostre cas, s'encarregarà de controlar quin dels threads d'usuari serà el que s'executarà.

```
1: function Scheduler() {
2:     this.thread_queue = [];
3:     this.actual_thread = 0;
4:     this.num_threads = 0;
5:     this.main_context;
6: }
```

Com podem veure la funció Scheduler s'utilitzarà per a la creació d'objectes ja que el seu nom comença per majúscula (convenció de JavaScript). Aquest objecte contindrà quatre propietats que s'utilitzaran per controlar els threads.

- thread_queue: Serà la cua que representarà els threads. Contindrà en un primer moment els blocs que han d'executar els threads i posteriorment, aprofitant que les variables a JavaScript no necessiten de tipus, contindrà els contextos dels threads per poder continuar l'execució on l'havien deixat.
- actual_thread: Indicarà la posició del thread en la cua, que s'està executant o haurà d'executar-se.

- num_threads: Indicarà el nombre de threads que estan actius, es a dir, que encara han de acabar la seva execució.
- main_context: Aquesta variable contindrà el context del thread principal per tal d'acabar l'execució.

Funció spawn

```
1: Scheduler.prototype.spawn = function(thunk) {
2:     this.num_threads++;
3:     this.thread_queue.push(thunk);
4: }
```

La funció spawn cada vegada que es cridi incrementarà el nombre de threads que hi han actius i afegirà a la cua el bloc (thunk) que ha d'executar el thread.

Funció start_threads

```
1: Scheduler.prototype.start_threads = function() {
2:     if ((this.main_context = current_context())) {
3:      this.thread_queue[this.actual_thread]();
4:     }
5: }
```

La funció start_threads el que fa és molt senzill però a la vegada molt important. Com es pot veure a la línia 2 al if es produeix una assignació, NO una comparació. D'aquesta manera s'esta assignant a la propietat main_context el context actual, es a dir, tindrem una continuació. Aquesta propietat indica el context del thread principal d'execució, aquell del programa principal, que ens serà útil més endavant per poder acabar l'execució. A partir d'aquí com es cumplirà la condició del if cridarà al thread que ha d'executar-se el qual sempre serà el primer.

Quan arribi el cas de que es cridi a la continuació de main_context per restaurar el context, fixem-nos que, com hem dit a la part A del document, aprofitem que la crida a la continuació retornarà undefined de forma predeterminada i per tant no entrarà dins del if.

Funció relinquish

```
1: Scheduler.prototype.relinquish = function() {
2:      if (this.thread_queue[this.actual_thread] = current_context()) {
3:      this.actual_thread = (this.actual_thread + 1)%this.num_threads;
4:      this.thread_queue[this.actual_thread]();
5:     }
6: }
```

En quant a la funció relinquish, aquesta s'encarrega de cedir el control del thread actual per el següent.

Com pasava amb la funció start_threads al if de la línia 2 es produeix una assignació en la que guardem la continuació del thread que ha de cedir el processador i que podrem fer servir més

endavant per restaurar el se context i continuar l'execució del codi. Com abans, entrarem al if menys quan restaurem el context ja que es retornarà un undefined.
A la línia 3 el que estem fent és determinar la posició del thread al qual li toca el processador i a la 4 restaurem el context d'aquest o la funció que ha d'executar depenent si havia començat la seva execució o no.

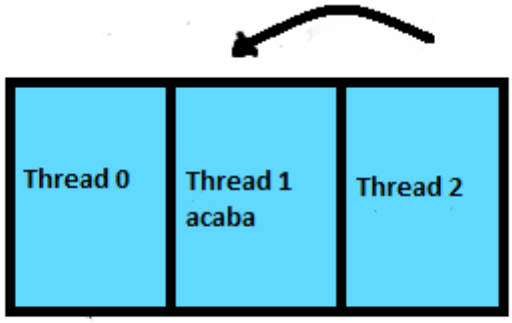
Funció quit

```
1: Scheduler.prototype.quit = function() {
2:     this.num_threads--;
3:     if (this.actual_thread !== this.num_threads)
4:      this.thread_queue[this.actual_thread] = this.thread_queue[this.num_threads];
5:     if (this.actual_thread !== this.num_threads - 1)
6:      this.actual_thread = (this.actual_thread + 1)%(this.num_threads + 1);
7:     if (this.num_threads > 0) this.thread_queue[this.actual_thread]();
8:     else this.main_context();
9: }
```

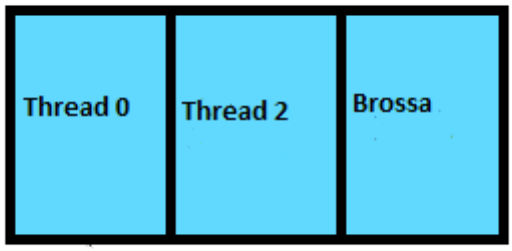
La funció quit ha de treure els threads de la cua per a que no s'executin més. Hem implementat aquesta funció de la següent forma:

Primer de tot a la línia 2 reduïm el nombre de threads que hi han en execució ja que el que es troba ara ha de acabar. Després a la línia 3 comprobem si el thread que acaba no és l'últim thread de la cua per tal de posar l'últim thread de la cua a la posició del que s'en va. Amb això aconseguim que a la part final de l'array que controla la cua de threads sempre estiguin els threads que acaben i com que la variable num_threads controla el nombre de threads que hi han actius mai arribarem a aquelles posicions de la cua les quals considerem com a brossa, a més mantindrem l'ordre dels threads. Als següents gràfics podem veure una representació del que simulem al codi.

Gràfic 2: El thread 1 acaba.



Gràfic 3: El thread 1 ha de moure's al final del array i mantenim l'ordre.



Gràfic 4: Disminuïm el nombre de threads de 3 a 2.

Després com un thread acaba hem de pasar al següent això ho fem a les línies 6 i 7. Al contrari que a la funció relinquish que sempre determinavem el següent thread, ara només ho fem si no és el penúltim ja que si ho fos, al reduïr el nombre de threads pasariem a executar el thread equivocat.

Per últim comprobem si hi han threads a la línia 7, si és així, restaurarem el context del thread al qui li tocaria executar i retornariem a la funció relinquish on aquest cop no entariem al if, sino fos així, sabem que hauriem acabat i per tant hauriem de tornar al thread principal, per tant, restauraríem el context del thread principal, on es retornaria a la funció start_threads i aquest cop no entrariem al if i acabariem l'execució.

Jocs de prova

Número 1

El primer joc de prova que hem realitzat és similar al programa base que es proporciona a la pràctica i el podem trobar al fitxer JocProva1.txt, només cal copiar-ho al fitxer de la pràctica i comentar el programa base. El que fa és imprimir per terminal les taules de multiplicar del zero fins al deu. Hem afegit un thread més, es a dir, quatre threads que aniran escribint les taules. Quan un thread acaba d'escriure una taula es cedeix el processador al següent thread que escriurà la següent taula fins acabar. Com passava amb el programa base, això ve controlat per una variable global anomenada table que indica quina taula és la que toca.

Amb aquest joc de prova voliem provar que el programa funcionés correctament amb un exemple senzill com el del programa base però afegint incrementant el nombre de threads. Si veiem la sortida, podem veure que els threads es van cedint el processador de forma cíclica tal i com s'esperava i que a l'hora d'acabar també acaben de forma correcta.

Número 2

En el segon joc de prova provem que podem forçar a un thread a acabar la seva execució i continuar amb la resta. El podem trobar a JocProva2.txt.

En aquest cas creem dos threads que executen una funció que realitza multiplicacions. Quan arribem a la condició del for on la variable i és 5 i coincideix que el thread té el nom d'a forcem la seva finalització.

Si observem la sortida veiem que el thread a acaba correctament, quan al for arriba a 5 i més endavant veiem que l'execució la continua el thread b sempre de manera que el thread a ha acabat correctament i no ha afectat a l'execució del b.

Número 3

En el tercer joc de prova utilitzem la mateixa funció de multiplicacions que al joc dos sense la condició que feia la finalització forçosa del thread.

En aquest cas creem només un thread ja que voliem comprobar que no es n'hi hagués cap error en l'execució quan només hi participa un thread.

Com podem observar a la sortida veiem que tot s'executa correctament i finalitza d'igual forma.

Número 4

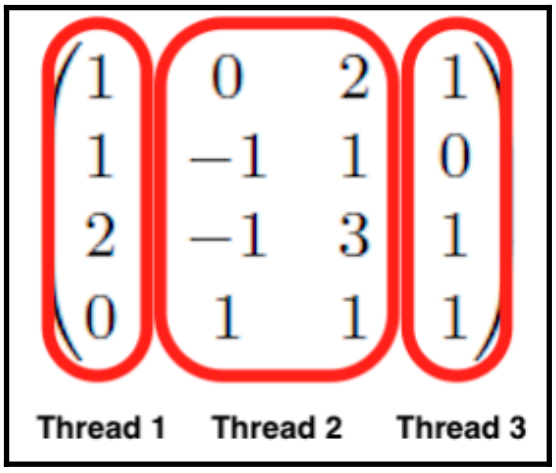
En aquest joc de prova creem tres threads que hauran de executar tres funcions diferents cadascun, una que realitza la seqüencia de fibonacci, una altre que realitza multiplicacions i l'última que tan sols escriu per pantalla. El podem trobar a JocProva4.txt.

En totes les funcions el thread hi haurà un moment en que haurà de cedir el processador a un altre thread, en la de fibonacci serà cada vegada que escrigui un nombre per la terminal, en la de multiplicacions serà similar a la de fibonacci però l'útima en la funció restant només cedirà als altres threads quan sigui parell la variable del for que el fa poder escriure per pantalla.

Amb aquest joc de prova voliem simular una execució paral·lela dels threads en diverses funcions i veure que realment, l'execució era correcta i es cedia bé el processador i acabaven tal i com s'esperava. Si veiem la sortida, podem veure que els threads van canviant-se de forma correcta com passava a la resta de jocs de prova i que acaben de la mateixa forma.

Número 5

En aquest joc de prova creem 3 threads per tal de realitzar una lectura d'una matriu. Aquesta lectura, però, la volem fer selectiva per a cada thread. Volem que un thread llegeixi només la primera columna de l'esquerra de la matriu, un altre thread que llegeixi la columna de la dreta, i l'últim thread que llegeixi la part central.



Exemple de la lectura de la matriu amb els threads

Amb aquest joc de prova voliem comprovar que podiem canviar l'execució dels threads al nostre gust aprofitant que aquests es cedeixen de forma cíclica. Aquest és un exemple però es podria fer de moltes altres maneres. Hem comprovat que cedeixen correctament els processadors de forma cíclica i que per tant, la sortida és tal i com l'esperàvem.