Pràctica CAP: Continuacions en Smalltalk

Albert Bertran Serrano
Gerard Queralt Ferré

Scheduler

Abans de començar amb la implementació de la pràctica, hem hagut de comprendre bé quina és la funcionalitat de cada un dels mètodes que s'especifica, i que els descrivim breument a continuació:

- spawn: aThunk:
  Aquesta primera funció s'encarrega d'afegir un nou thread a la cua de l'Scheduler.
- quit:
  Aquesta funció s'encarrega de parar el thread que s'està executant i treure'l de la cua.
- relinquish:
  Aquesta funció, una mica més complexa, s'encarrega de pausar el thread actual i cedir el control al següent thread de la cua.
- startThreads:
  Per últim, aquesta funció és l'encarregada de començar a executar els threads que hi ha a la cua.

Addicionalment, a les funcions comentades anteriorment també necessitarem crear un bloc d'escapament "stop" que aturarà totalment l'execució.
També hem creat una classe Queue a partir d'OrderedCollection, amb els mètodes que defineixen una cua, "addLast" i "removeFirst". Aquesta classe, en realitat, és redundant, perquè només crida als mètodes ja implementats a OrderedCollection, però, tot i això, l'hem definit per diferenciar que la nostra estructura és una cua.

Primera versió

Per començar vam veure que necessitaríem una cua per gestionar els diferents threads, primer vam pensar de fer-ne dues, una pels diferents thunks i una altra per les continuacions. Però vam veure que la cua de thunks no feia falta realment i que l'únic que feia era complicar la implementació de la solució.
Així doncs, vam decidir fer una única cua de continuacions.

En aquesta primera versió vam implementar la classe Queue i vam definir els atributs de la classe Scheduler, així com la seva inicialització:

- Scheduler class >> initialize [
  super initialize.
  self threadQueue: Queue new
  ]

A continuació detallarem la implementació de les funcions principals de la pràctica.
En primer lloc “Quit”:

- Scheduler class >> quit [
  self threadQueue isNotEmpty ifTrue: [self threadQueue removeFirst value]
  ifFalse: [self stop value]
  ]

  Aquest missatge atura l'execució del thread actual i el treu de la cua. Així doncs, primer comprovem si la cua de continuacions és buida; si no ho està, continuem l'execució del següent thread; si ho està aturem l'execució avaluant el bloc d'escapament.

- Scheduler class >> spawn: aThunk [
  | cc |
  cc := Continuation current.
  cc isNotNil
  ifTrue: [self threadQueue addLast: cc]
  ifFalse: [aThunk value. self quit]
  ]

En aquest mètode afegim una continuació a la cua de threads que quan s'avaluï comenci a executar el thunk. Per fer-ho, després de la declaració de la continuació comprovem que l'acabem de crear amb un "isNotNil" (perquè quan l'avaluem li passem "nil" com a valor), i de ser així l'afegim a la cua. Si l'estem avaluant volem que s'executi el thunk.

- Scheduler class >> relinquish [
  | cc |
  cc := Continuation current.
  (cc isNotNil and: [self threadQueue notEmpty])
  ifTrue: [self threadQueue addLast: cc.
  self threadQueue removeFirst value]

  Aquesta funció pausa el thread actual i executa el següent. La manera de fer-ho és creant una continuació i fent una comprovació com la de l'spawn, amb l'afegit que hem d'assegurar-nos que hi ha un altre thread que executar. Si és així, afegim la nova continuació al final de la cua i reprenem l'execució de la primera.

- Scheduler class >> startThreads [
  self stop: [^nil].
  self threadQueue notEmpty ifTrue: [self threadQueue removeFirst value]
  ]

  En aquesta funcionalitat declarem el bloc d'escapament i, en cas que la cua de threads no sigui buida, n'executem el primer.

Aquesta versió, però, no funcionava, i tot i que teniem clar perquè, ens va costar adonar-nos-en de la solució. Com bé sabem, un bloc només pot retornar si el context en que s'ha creat existeix encara a la pila d'execució. El problema era que totes les continuacions es creaven a l'spawn, i com que el context de startThreads no existia en aquell punt de l'execució que restauravem amb la continuació, l'execució sempre acabava amb un BlockCannotReturn.

Segona versió

En aquesta segona versió vam trobar la manera de solucionar el problema esmentat: vam veure que havíem de canviar el missatge "spawn", i així ho vam fer. Les altres funcionalitats les vam conservar respecte de la primera versió.
A continuació es mostra el codi corresponent a la funció "spawn" d'aquesta segona versió:

- Scheduler class >> spawn: aThunk [
  self threadQueue addLast: aThunk
  ]

  Com es pot veure hem eliminat la creació de les continuacions del mètode i senzillament afegim el thunk directament a la cua de threads. Podem permetre'ns fer això perquè tant el thunk com la continuació són, senzillament, blocs sense paràmetres, i, per tant, no hi ha diferència en com els avaluem: ho podem fer amb el missatge "value". D'aquesta manera el context on s'ha declarat el bloc d'escapament segueix a la pila quan s'avalua, i, com a resultat, acaba correctament l'execució.

Aquest seria el principal aprenentatge que creiem que ens endúiem de la pràctica: perdre la por a les continuacions i saber tractar-les també com a blocs quan cal. La resta de conceptes de continuacions, com el valor que prenen en ser declarades i en avaluar-se, així com les particularitats i complicacions dels blocs d'escapament, ja ens havien quedat clars a classe.

Explicació exemples

- Factorial

```
Scheduler class >> factorial: aNumber [
	| fact factThunk |
	fact := Array new: aNumber.
	factThunk := [ :n |
		[ n <= 1
			ifTrue: [ (n asString, '! = 1') traceCr.
				fact at: 1 put: 1.
				Scheduler quit. ]
			ifFalse: [ Scheduler spawn: (factThunk value: n - 1).
				[ (fact at: (n-1)) isNil ] whileTrue: [ Scheduler relinquish].
				fact at: n put: (fact at: (n-1)) * n.
				(n asString, '! = ', (fact at: n) asString) traceCr.
				Scheduler quit. ]
		] ].
	Scheduler initialitze.
	Scheduler spawn: (factThunk value: aNumber).
	Scheduler startThreads.
]
```

Aquest exemple, com es desprèn pel nom, calcula el factorial d'un determinat nombre, d'una manera similar a l'exemple dels n-éssims nombres de Fibonacci. Senzillament, definim un cas base i un cas recursiu que cedeix l'execució mentre no s'hagi calculat el nombre anterior; un cop s'ha fet, guarda el resultat que calcula en una array (que alhora serveix per avisar que ha acabat el seu càlcul) i imprimeix el resultat al Transcript.

- Reverse

```
Scheduler class >> reverse: aCollection [
	| reverseThunk |
	reverseThunk := [ :s | [
		s == 0
			ifFalse: [
				((aCollection at: s) asString) trace.
				Scheduler spawn: (reverseThunk value: s - 1).
				Scheduler quit
				]
		] ].
	Scheduler initialize.
	Scheduler spawn: (reverseThunk value: (aCollection size)).
	Scheduler startThreads.
]
```

Aquest exemple inverteix una col·lecció que rep com a paràmetre, com hom es pot imaginar a partir del nom. Senzillament, mostra el valor al Transcript, crea un thread nou i acaba l'execució.

- Map

```
Scheduler class >> map: aBlock to: aCollection [
        | size result mapThunk |
        size := aCollection size.
        result := Array new: size.
        mapThunk := [ :pos |
                [size < pos
                        ifFalse: [
                                | valAtPos |
                                valAtPos := aCollection at: pos.
                                result at: pos put: (aBlock value: valAtPos).
                                (pos asString, '=', (result at: pos) asString) traceCr.
                                Scheduler spawn: (mapThunk value: pos + 1).
                                Scheduler quit.
                        ]
                ] ].

        Scheduler initialize.
        Scheduler spawn: (mapThunk value: 1).
        Scheduler startThreads.
        ^ result
]
```

Aquest exemple també té un nom força explicatiu: és una implementació de map. Per tant, donat un bloc amb un paràmetre i una llista, calcula la llista resultant d'aplicar el bloc sobre cada element.