Pràctica CAP 2017 - 2018

Albert López Alcàcer Carlos Lázaro Costa

Índex

a.1 - Pharo: Continuation class >> continuation

En JavaScript l'evaluació de `new Continuation` retorna al punt d'execució de la crida a la funció on es va cridar `new Continuation`.

En Pharo tenim la possibilitat d'accedir a la pila amb `thisContext` i l'evaluació de `Continuation current` retorna al punt on es va cridar `new Continuation`, en comptes de la funció que la va cridar.

Per tant, la implementació de `new Continuation()` a Pharo serà similar a la de `current`, però utilitzant la implementació amb una continuació que té el context de la funció que crida a `continuation`, és a dir `thisContext sender sender` sent el primer sender la crida a `continuation` (implementació de `current`) i el segon sender la crida a la funció que crida a `continuation`.

```
continuation
    "JS-like new Continuation()"
    ^self fromContext: thisContext sender sender
```

Per tant, a partir d'aquí podem definir el nostre propi `current` que anomenarem `current_continuation` utilitzant el `continuation`, que serà la mateixa implementació que utilitzarem en JavaScript a l'apartat b. També definida al class-side de Continuation.

```
current_continuation
    "current implementation using JS-like continuation"
    ^self continuation
```

Per provar aquests mètodes tenim el següent fragment de codi:

```
| cc |
cc := Continuation current_continuation.
Transcript show: cc class; cr.
(cc isMemberOf: Continuation)
    ifTrue: [ cc value: 2 ]
    ifFalse: [ Transcript show: cc; cr ].
```

El resultat del Transcript haurà de ser:

```
Continuation
SmallInteger
2
```

Finalment, comprovem que efectivament el nostre `current_continuation` és equivalent a `current` amb el següent fragment de codi:

```
| cc |
cc := Continuation current.
Transcript show: cc; cr.
(cc == 2) ifFalse: [ cc value: 2 ].

cc := Continuation current_continuation.
Transcript show: cc; cr.
(cc == 2) ifFalse: [ cc value: 2 ].
```

El resultat del Transcript haurà de ser:

```
a Continuation
2
a Continuation
2
```

a.2 - JavaScript: callcc(f)

Donat que el `new Continuation()` funciona diferent que el `current` de Pharo la implementació de callcc en JavaScript ha de ser la següent:

```
function callcc(f) {
    return f(new Continuation());
}
```

D'aquesta manera s'evaluarà la funció `f` amb la continuació com a paràmetre que en ser evaluada retornarà allà on s'ha cridat `callcc`, que és exactament el mateix comportament que amb Smalltalk.

També podem definir el `current_continuation` que hem vist a l'apartat anterior:

```
function current_continuation() {
    return new Continuation();
}
```

Per provar aquests mètodes hem definit dos tests:

```
// current_continuation
function testContinuation() {
  var cc = current_continuation();
  if (cc instanceof Continuation) {
    cc(2);
  } else {
    assertEquals(2, cc);
  }
}

// callcc
function testCallcc() {
```

```
  var kont;
  var i = callcc((cc) => {
    kont = cc;
    return 1;
  });

  if (i === 1) {
    kont(2);
  } else {
    assertEquals(2, i);
  }
}
```

Aquests tests es poden trobar al fitxer **continuation.js**.

La prova `testContinuation` és una implementació equivalent a la mateixa prova que hem definit amb Pharo a l'apartat anterior. La prova `testCallcc` comprova que efectivament el comportament és equivalent al `callcc` definit amb Pharo:

```
callcc: aBlock
    ^self currentDo: aBlock
```

El codi equivalent de `testCallcc` amb Smalltalk seria:

```
| kont i |
i := Continuation callcc: [ :cc | kont := cc. 1 ].
(i = 1)
    ifTrue: [ kont value: 2 ]
    ifFalse: [ Transcript show: i; cr ].
```

Això hauria de mostrar `2` al Transcript.

b - JavaScript: Multi-threading cooperatiu

Per fer un sistema multi-fil cooperatiu hem de definir mètodes que ens permetin afegir un thread, canviar el fil d'execució a un altre thread i per inicialitzar i finalitzar els threads.

La funció `make_thread_system` crea un objecte amb aquestes cuatre funcions com a propietats:

```
function make_thread_system() {
    var threads = [];
    var halt;

    function shift() {
        var nt = threads.shift();
        nt();
    }

    return {
        spawn: function(thunk) {
            var c = current_continuation();
            if (c instanceof Continuation) {
                threads.push(c);
            } else {
```

```
                thunk();
            }
        },
        quit: function() {
            if (threads.length > 0) {
                shift();
            } else {
                halt();
            }
        },
        relinquish: function() {
            var c = new Continuation();
            threads.push(c);
            shift();
        },
        start_threads: function() {
            halt = new Continuation();
            shift();
        }
    }
}
```

Em definit una cua de threads disponibles pel sistema de threads, implementats com a continuacions, una funció `shift` encarregada de treure la primera continuació de la cua i evaluar-la i una variable `halt` on guardem una referència a la continuació que en ser evaluada retorna a la crida `start_threads`, finalitzant l'execució del sistema.

A la funció `spawn` afegim un thread a la cua de threads, com una continuació dins d'aquesta funció, de forma que quan s'evaluï s'executarà el codi del thread corresponent, anomenat *thunk*. Aquesta evaluació té lloc a la funció `start_threads`, on una vegada em afegit tots els threads necessaris procedim a executar-los. Per canviar el fil d'execució tenim la funció `relinquish` que afegeix el punt actual com una continuació a la cua de threads i executa el següent, allà on s'hagués quedat anteriorment. Finalment tenim la funció `quit` encarregada de parar el sistema de threads en cas de que no hi hagi cap més thread per executar-se, o delegar l'execució a un altre thread restant, que quan acabi farà la seva crida a `quit`.

Per provar aquest codi s'han definit quatre exemples diferents.

El primer és el codi d'exemple de la pràctica, que prova com diferents threads poden dividir-se el treball de comptar desde 10 fins a 0, intercanviant de thread en cada pas.

```
var counter = 10;
function make_thread_thunk(name, thread_system) {
    function loop() {
        if (counter < 0) {
            thread_system.quit();
        }
        print('in thread', name, '; counter =', counter);
        counter--;
        thread_system.relinquish();
        loop();
    };
    return loop;
}
```

```
var thread_sys = make_thread_system();
thread_sys.spawn(make_thread_thunk('a', thread_sys));
thread_sys.spawn(make_thread_thunk('b', thread_sys));
thread_sys.spawn(make_thread_thunk('c', thread_sys));
thread_sys.start_threads();
```

Aquest exemple ha de mostrar la sortida següent:

```
in thread a ; counter = 10
in thread b ; counter = 9
in thread c ; counter = 8
in thread a ; counter = 7
in thread b ; counter = 6
in thread c ; counter = 5
in thread a ; counter = 4
in thread b ; counter = 3
in thread c ; counter = 2
in thread a ; counter = 1
in thread b ; counter = 0
```

El segon exemple és el del avís del racó com exemple alternatiu de la pràctica, que prova diferents threads realitzant la mateixa feina cadascun de comptar de 0 a 4, intercanviant de thread a cada pas.

```
function make_thread_thunk_2(name, thread_system) {
  function loop() {
    for (let i = 0; i < 5; i++) {
      print('in thread', name, '; i =', i);
      thread_system.relinquish();
    }
    thread_system.quit();
  };
  return loop;
}

thread_sys = make_thread_system();
thread_sys.spawn(make_thread_thunk_2('a', thread_sys));
thread_sys.spawn(make_thread_thunk_2('b', thread_sys));
thread_sys.spawn(make_thread_thunk_2('c', thread_sys));
thread_sys.start_threads();
```

Aquest exemple ha de mostrar la sortida següent:

```
in thread a ; i = 0
in thread b ; i = 0
in thread c ; i = 0
in thread a ; i = 1
in thread b ; i = 1
in thread c ; i = 1
in thread a ; i = 2
in thread b ; i = 2
in thread c ; i = 2
in thread a ; i = 3
in thread b ; i = 3
in thread c ; i = 3
```

```
in thread a ; i = 4
in thread b ; i = 4
in thread c ; i = 4
```

El tercer exemple és un exemple propi que calcula el màxim d'una llista no buida utilitzant diferents threads que es divideixen el treball, intercanviant de thread a cada pas:

```
var list = [1, 3, 2, 6, 9, 3, 0]; // not empty
print('list', list);

var i = 0;
var max = list[i];

function make_thread_thunk_3(name, thread_system) {
  function list_max() {
    var count = 0;
    while (i < list.length) {
      print('in thread', name, '; list[', i, '] = ', list[i]);
      if (list[i] > max) {
        print('update max = ', list[i]);
        max = list[i];
      }
      i++;
      thread_system.relinquish();
    }
    thread_system.quit();
  };
  return list_max;
}

thread_sys = make_thread_system();
thread_sys.spawn(make_thread_thunk_3('a', thread_sys));
thread_sys.spawn(make_thread_thunk_3('b', thread_sys));
thread_sys.spawn(make_thread_thunk_3('c', thread_sys));
thread_sys.start_threads();
print('max = ', max);
```

Aquest exemple ha de mostrar la sortida següent:

```
list 1,3,2,6,9,3,0
in thread a ; list[ 0 ] =  1
in thread b ; list[ 1 ] =  3
update max =  3
in thread c ; list[ 2 ] =  2
in thread a ; list[ 3 ] =  6
update max =  6
in thread b ; list[ 4 ] =  9
update max =  9
in thread c ; list[ 5 ] =  3
in thread a ; list[ 6 ] =  0
max =  9
```

El quart exemple és molt similar a l'anterior, a diferència de que l'ordre d'execució dels threads és diferent, en aquest exemple cada thread fa més d'una iteració consecutivament, de forma que al primer

thread li corresponen els primers elements de la llista, al segon una altra part, i així per tots els threads. Si amb aquestes iteracions no s'ha acabat la llista (és a dir, la divisió de la mida de la llista entre el nombre de threads no és entera) llavors els elements restants els procesa el primer thread:

```
var list = [1, 3, 2, 6, 9, 3, 0]; // not empty
var n_threads = 3;
var thread_count = parseInt(list.length / n_threads);

print('list', list);
print('n_threads', n_threads);
print('thread count', thread_count);

var i = 0;
var max = list[i];

function make_thread_thunk_4(name, thread_system) {
  function list_max() {
    var count = 0;
    while (i < list.length) {
      print('in thread', name, '; list[', i, '] = ', list[i]);
      if (list[i] > max) {
        print('update max = ', list[i]);
        max = list[i];
      }
      i++;
      count++;
      if (count === thread_count) {
        count = 0;
        thread_system.relinquish();
      }
    }
    thread_system.quit();
  };
  return list_max;
}

thread_sys = make_thread_system();

for (let t = 1; t <= n_threads; t++) {
  thread_sys.spawn(make_thread_thunk_4(String.fromCharCode(96 + t), thread_sys));
}

thread_sys.start_threads();
print('max = ', max);
```

Aquest exemple ha de mostrar la sortida següent:

```
list 1,3,2,6,9,3,0
n_threads 3
thread count 2
in thread a ; list[ 0 ] =  1
in thread a ; list[ 1 ] =  3
update max =  3
```

```
in thread b ; list[ 2 ] =  2
in thread b ; list[ 3 ] =  6
update max =  6
in thread c ; list[ 4 ] =  9
update max =  9
in thread c ; list[ 5 ] =  3
in thread a ; list[ 6 ] =  0
max =  9
```

L'execució amb **`rhino -opt 2 fils-cooperatius.js`** executa els tests de l'apartat **a.2** i els exemples del `make_thread_system`.

La sortida està guardada en el fitxer **fils-cooperatius-sortida.txt**:

```
Test testContinuation success
Test testCallcc success
Test make_thread_thunk (global counter)
in thread a ; counter = 10
in thread b ; counter = 9
in thread c ; counter = 8
in thread a ; counter = 7
in thread b ; counter = 6
in thread c ; counter = 5
in thread a ; counter = 4
in thread b ; counter = 3
in thread c ; counter = 2
in thread a ; counter = 1
in thread b ; counter = 0
Test make_thread_thunk_2 (per-thread counter)
in thread a ; i = 0
in thread b ; i = 0
in thread c ; i = 0
in thread a ; i = 1
in thread b ; i = 1
in thread c ; i = 1
in thread a ; i = 2
in thread b ; i = 2
in thread c ; i = 2
in thread a ; i = 3
in thread b ; i = 3
in thread c ; i = 3
in thread a ; i = 4
in thread b ; i = 4
in thread c ; i = 4
Test make_thread_thunk_3 (maximum of a list)
list 1,3,2,6,9,3,0
in thread a ; list[ 0 ] =  1
in thread b ; list[ 1 ] =  3
update max =  3
in thread c ; list[ 2 ] =  2
in thread a ; list[ 3 ] =  6
update max =  6
in thread b ; list[ 4 ] =  9
```

```
update max =  9
in thread c ; list[ 5 ] =  3
in thread a ; list[ 6 ] =  0
max =  9
Test make_thread_thunk_4 (maximum of a list defining thread granularity)
list 1,3,2,6,9,3,0
n_threads 3
thread count 2
in thread a ; list[ 0 ] =  1
in thread a ; list[ 1 ] =  3
update max =  3
in thread b ; list[ 2 ] =  2
in thread b ; list[ 3 ] =  6
update max =  6
in thread c ; list[ 4 ] =  9
update max =  9
in thread c ; list[ 5 ] =  3
in thread a ; list[ 6 ] =  0
max =  9
```