FACULTAT D'INFORMÀTICA DE BARCELONA
COMPUTER SCIENCE DEPARTMENT
CONCEPTES AVANÇATS DE PROGRAMACIÓ

Pràctica CAP 2017-18

Continuacions en JavaScript i Smalltalk
Fils cooperatius

Alumnes:
Ruiz Lombarte, Albert
Gonzàlez Montiel, Pau

Data:
31/01/2018

Curs:
QT 2017-2018

Índex de continguts

a.1.- Smalltalk: Implementació del mètode `Continuation class >> continuation` **2**

a.1.1.- Proves realitzades per comprovar `Continuation class >> continuation` **2**

a.2.- JavaScript: Implementació del mètode `callcc(f)` **3**

a.2.1.- Proves realitzades per comprovar `callcc(f)` **4**

b.- Fils coopeartius: Implementació de la funció `make_thread_system()` **6**

b.1.- Proves realitzades per comprovar `make_thread_system()` **8**

a.1.- Smalltalk: Implementació del mètode `Continuation class >> continuation`

A continuació, us mostrem el mètode ***continuation*** en *Smalltalk* que funciona exactament igual que el mètode **Continuation()** de *Javascript:*

```
continuation
    ^Continuation fromContext: thisContext sender sender.
```

Tal i com diu l'enunciat, hem definit un mètode anomenat **continuation** que retorna una instància de **Continuation.** En *Smalltalk,* les instàncies de la classe **Continuation** serveixen per guardar la pila d'execució en un moment donat. El que estem fent en la línia de codi anterior és retornar el context actual en una instància de **Continuation.** Utilitzem el *sender*, que referencia un altre Context, aquell que es activat per *thisContext*, que fa referència a la pila de temps d'execució actual com un objecte de tipus *MethodContext.* Per tant, *thisContext* fa referència a una instància de *MethodContext* que descriu el marc de la pila actual.

a.1.1.- Proves realitzades per tal de comprovar el bon funcionament de `Continuation class >> continuation`

Per comprovar el bon ús de la funció, el que hem fet ha sigut comparar el resultat d'executar un programa amb javascript utilitzant **Continuation()** i un programa amb *Smalltalk* utilitzant el codi anterior. Hem agafat un codi semblant que hi ha a les transparències de l'assignatura(*CAP_Tema3_Proto.pdf, pag. 69)* .
Exemple en *javascript:*

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

var k = continuacio(); //línia 5
if (k instanceof Continuation) {
    print('k és una continuació');
    k(1); //línia 8
}
else {
    print('k ara és un ' + typeof(k));
}
print(k);
```

El programa consisteix en la variable ***k*** que primer és una continuació, però un cop ***k*** és invocada a la línia 8, el programa torna a la crida de la funció ***continuacio()*** de la línia 5, que aquest cop la funció retorna el valor '1', que ha sigut passat a la línia 8. Per tant la sortida és la següent:

```
k és una continuació
k ara és un number
1
```

En *Smalltalk:*

```
| continuacio k |
continuacio := [
  | cont |
  cont := Continuation continuation.
  cont ].

k := continuacio value.
( k class = Continuation )
     ifTrue: [
          Transcript show: 'k és una continuació'; cr.
          k value: 1. ]
     ifFalse: [
          Transcript show: 'k ara és un '; cr.
          Transcript show: k class; cr.
     ].
Transcript cr.
Transcript show: k; cr.
```

Sortida:

```
k és una continuació
k ara és un SmallInteger
1
```

a.2.- JavaScript: Implementació del mètode `callcc(f)`

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

Per implementar el mètode la primera pista que teniem és que era necessari una funció `f` com a únic argument. Aleshores necessitavem una instància del context de control actual del programa com a objecte que això és el que ens proporicona la crida `new Continuation()` de Rhino, després aplicariem `f` a aquest objecte. D'aquesta manera quan s'aplica un objecte continuació a un argument, la continuació existent s'elimina i la continuació aplicada es restaura en el seu lloc, de manera que el flux del programa continuarà en el punt en què es va capturar la continuació i l'argument d'aquesta es converteix en el valor retornat de la invocació de *callcc*.

a.2.1.- Proves realitzades per tal de comprovar el bon funcionament de `callcc(f)`

Les proves que hem fet per tal de comprovar que el mètode funciona correctament són les següents:

Codi senzill de l'exemple de les transparències (*CAP_Tema2_Reflexio_Smalltalk.pdf, pàg. 120)*

Codi en *Smalltalk*:

```
| x continuation |
x := Continuation callcc: [ :cc | continuation := cc. false ].
x ifFalse: [ continuation value: true ].
x
```

Sortida:
`true`

Codi en *JavaScript*:

```
var x;
var continuation;
x = callcc(function(cc){
    continuation = cc;
    return false;
});
if (!x) {
    continuation(true);
}
print(x);
```

Sortida:
`true`

Codi del **whileTrue** (*CAP_Tema2_Reflexio_Smalltalk.pdf, pàg. 124*) que utilitza el callcc i que hem implementat amb JavaScript fent un petit programa que es manté dins del bucle mentre és cert que un valor generat aleatoriament entre 0 i 100 és igual o major que 10.

Codi en Smalltalk del whileTrue:

```
whileTrue: aBlock
"whileTrue code version implemented with callcc"
| continuation |
continuation := Continuation callcc: [ :cc | cc ].
self value ifTrue:[   aBlock value.
                      continuation value: continuation]
           ifFalse:[  ^ nil].
```

Codi en JavaScript de la prova:

```
var continuation = callcc(function(cc) {
        return cc;
    });
var randomNumber;
var returnsBooleanFunction = function() {
    randomNumber = Math.floor(Math.random()*100);
    return (randomNumber > 10);
};
var whileTrue = function(returnsBooleanFunction) {
    if (returnsBooleanFunction()) {
        print('The random number is '+randomNumber+' not less than 10...');
        continuation(continuation);
    }
    else {
        print('LOOP EXITED: The number is '+randomNumber+' less than 10.');
        return null;
    }
};
whileTrue(returnsBooleanFunction);
```

Possible sortida:

```
The random number is 29 not less than 10...
The random number is 46 not less than 10...
The random number is 37 not less than 10...
The random number is 73 not less than 10...
The random number is 18 not less than 10...
LOOP EXITED: The number is 4 less than 10.
```

b.- Fils coopeartius: Implementació de la funció `make_thread_system()`

Abans de començar a picar codi vam extreure una serie de d'idees de l'enunciat que configurarien els atributs de la funció **ThreadSystem**.

En primer lloc, hem necessitat una cua de threads on emmagatzemar els fils que s'han d'executar. Això ho hem representat en codi com un Array, a més amb l'operació *push()* que ens ha servit per afegir elements al final de la cua i amb l'operació *shift()* per tal d'extreure el primer.

També hem necessitat un objecte que representés el Thread que té el control i que s'està executant en un moment donat. Finalment hem necessitat una continuació que ens servis per recuperar el context en el que s'inicia la tasca del següent Thread de manera que quan un thread acaba la seva execució o cedeix el pas és pugui executar la tasca del següent. Així doncs, el codi resultant d'aquest raonament ha estat el següent:

```
function ThreadSystem() {
    this.threads_queue = [];
    this.current_running_thread = {};
    this.continuation;
}
```

Un cop fet això, hem implementat les quatre funcions que representen les quatre propietats de l'objecte a retornar i que conformen les utilitats del sistema multi-fil cooperatiu. Cadascuna d'aquestes propietats les hem afegit al prototype de la funció ThreadSystem per tal de disposar d'elles en totes les instàncies d'aquesta.

`spawn(thunk)`: Posa un thread nou amb la funció que anomenem thunk a la cua de threads. Que en termes de codi bàsicament ha consistit en afegir un nou element amb la funció *push()* al nostre Array que representa la cua de threads.

```
ThreadSystem.prototype.spawn = function(thunk) {
    this.threads_queue.push(thunk);
};
```

`quit()`: Que té com a objectiu aturar el thread que s'estava executant i el treu de la cua de threads. Si no hi ha cap thread restant a la cua es torna a inicialitzar l'objecte que representa el thread que s'està executant actualment. Si hi ha algun thread a la cua, és continua amb el següent thread.

```
ThreadSystem.prototype.quit = function() {
    if (this.threads_queue.length > 0) {
     this.current_running_thread = this.threads_queue.shift();
     this.current_running_thread();
    }
    else {
     this.current_running_thread = {};
     this.continuation();
    }
};
```

`relinquish()`: La tasca de la qual consisteix en cedir el control del thread que s'està executant actualment a un altre thread. Al fer un canvi de control d'execució a un altre thread, en la cua es s'emmagatzema l'estat del thread que acaba de deixar-se d'executar per tal de recuperar-lo posteriorment.

```
ThreadSystem.prototype.relinquish = function() {
    this.current_running_thread = this.threads_queue.shift();
    this.threads_queue.push(new Continuation());
    this.current_running_thread();
};
```

`start_threads()`: Té com a objectiu començar a executar els threads de la cua de threads.

```
ThreadSystem.prototype.start_threads = function() {
    this.continuation = new Continuation();
    this.current_running_thread = this.threads_queue.shift();
    this.current_running_thread();
};
```

Finalment, per tal d'utilitzar el sistema multi-fil cooperatiu fem us del mètode *make_thread_system()* per tal de retornar una instància de **ThreadSystem**:

```
function make_thread_system() {
    return new ThreadSystem();
}
```

b.1.- Proves realitzades per tal de comprovar el bon funcionament de `make_thread_system()`

El primer pas per tal de comprovar que el codi anterior començava a funcionar de manera correcta, abans de executar les proves donades com a exemple a l'enunciat, vam decidir executar el següent codi que senzillament fa que un thread executi una tasca i quan ha acabat deixa a un segon thread el control perquè executi la mateixa tasca.

```
function make_thread_thunk(thread_name, thread_system) {
    function task() {
        for (let i=1; i < 100; i++) {
            if (i%50 === 0) {
                print('... Computing '+thread_name+' task ...');
            }
        }
        print(thread_name+' has finished');
        thread_system.quit();
        task();
    };
    return task;
};


var simple_thread_sys = make_thread_system();
simple_thread_sys.spawn(make_thread_thunk('Thread 1',simple_thread_sys));
simple_thread_sys.spawn(make_thread_thunk('Thread 2',simple_thread_sys));
simple_thread_sys.start_threads();
```

La sortida d'aquest codi és la següent:

```
... Computing Thread 1 task ...
Thread 1 has finished
... Computing Thread 2 task ...
Thread 2 has finished
```

Per acabar, per tal de comprovar que el codi funciona de manera completament correcta hem implementat una prova que mostrem a continuació i que consisteix en un exemple que hi ha a les transparències de la funció factorial utilitzant el sistema multi-fil cooperatiu.

```
var fact = [];
function make_fact_thunk(n, thread_system) {
  function factorial() {
    if (isFinite(n) && n>0 && n==Math.round(n)) {
    if (!(n in factorial))
    thread_system.spawn(make_fact_thunk(n-1, thread_system));
    while(fact[n-1] == undefined) {
      thread_system.relinquish();
    }
    fact[n] = n*fact[n-1];
    thread_system.quit();
    }
    else {
    fact[0] = 1;
    thread_system.quit();
    }
  };
  return factorial;
}

var fact_threads = make_thread_system();
fact_threads.spawn(make_fact_thunk(5, fact_threads));
fact_threads.start_threads();
print("fact ", fact[5]);
```

La sortida d'aquesta prova és doncs el factorial de 5 que és igual a 120.

```
fact 120
```