Universitat Politècnica de Catalunya

Conceptes Avançats de Programació

Pràctica: JS Continuations

Fèlix Arribas Pardo
Carlota Catot Bragós

Quadrimestre Tardor 2017-2018

1 Continuation()

1.1 Smalltalk

Mètode

Presentem un mètode en Smalltalk que realitza una continuació com *callcc* però amb *Continuation class*, de la mateixa manera que ho fa JavaScript.

```
Continuation class >> continuation
    ^ self new initializeFromContext: thisContext sender.
```

Per arribar a aquesta conclusió hem fet un estudi a fons del que fa callcc per tal d'unificar totes les subcrides que realitza directament en una única crida anomenada *continuation*.

Provés per tal de comprovar la bona execució del mètode

Per tal de comprovar el correcte funcionament del mètode anterior, hem implementat en Smalltalk l'exemple que teníem a les diapositives que era un codi en JavaScript usant el mètode **Continuation**.

Codi en JavaScript:

```
function someFunction() {
    var kont = new Continuation();
    print('captured: ' + kont);
    return kont;
}

var k = someFunction();

if (k instanceof Continuation) {
    print('k is a continuation');
    k(200);
} else {
    print('k is now a ' + typeof(k));
}

print(k);
```

Sortida:

```
captured: [object Continuation]
k is a continuation
k is now a number
200
```

Codi en Smalltalk:

```
| someFunction k |

someFunction := [
    | kont |
    kont := Continuation continuation.
    Transcript show: 'captured: '.
    Transcript show: kont; cr.
    kont ].

k := someFunction value.

( k class = Continuation )
    ifTrue: [
        Transcript show: 'k is a continuation'; cr.
        k value: 200. ]
    ifFalse: [
        Transcript show: 'k is a '.
        Transcript show: k class; cr.
         ].

Transcript cr.
Transcript show: k; cr.
```

Sortida:

```
captured: a Continuation
k is a continuation
k is now a number
200
```

1.2 JavaScript

Mètode

Presentem un mètode que realitza la funció *callcc* de Pharo en JavaScript, usant la funció de Continuation() de JavaScript.

```
Continuation.callcc = function(aBlock) {
    return aBlock(new Continuation());
}
```

Per arribar a aquesta conclusió hem analitzat bé el que ens proporciona la funció Continuation() de JavaScript i per tal de fer-ho igual que el *callcc* de Pharo ens ha anat bé el realitzat a l'apartat anterior.

Provés per tal de comprovar la bona execució del mètode

Per tal de comprovar el correcte funcionament del mètode hem realitzat 3 codis de prova:

1. **Codi bàsic**: Aquesta prova ens ha servit per veure el funcionament del mètode amb un codi bàsic, pensant el que hauria de treure abans d'executar-lo i comprovant que, efectivament, és correcte. El codi consisteix en una variable que inicialment és una continuació i, a posteriori, s'avalua perquè passi a ser la lletra 'f'.

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

var x = someFunction();

if (!(x instanceof Continuation)) {
    print(x);
    print("és la lletra f");
} else {
    print(x);
    x('f');
}
```

Sortida:

```
[object Continuation]
f
és la lletra f
```

2. **callcc**: a partir del següent codi en Smalltalk, que utilitza el callcc:

```
| cont x |
x := Continuation callcc: [ :cc |
        cont := cc.
        cont value: 1 ].
(x = 1)
    ifTrue: [
            Transcript show: 'x = '.
            Transcript show: x; cr.
            cont value: 2 ]

    ifFalse: [
            Transcript show: 'x = '.
            Transcript show: x; cr. ].
```

Hem realitzat el mateix codi en js per comprovar que efectivament treu el mateix resultat i per tant que el nostre mètode callcc a JavaScript usant la funció Continuation(), és correcta en aquest cas:

```
Continuation.callcc = function(aBlock) {
    return aBlock(new Continuation());
}

var cont, x;

x = Continuation.callcc(function(cc) {
    cont = cc;
    return cc(1);
});

if (x == 1) {
    print("x = " + x);
    cont(2);
} else {
    print("x = " + x);
}
```

Sortida:

```
x = 1
x = 2
```

3. **WhileTrue**: Per últim, a partir del següent codi en Smalltalk, que implementa el whileTrue usant callcc:

```
| cont |
cont := Continuation callcc: [ :cc | cc ].
self value
    ifTrue:  [ aBlock value.
               cont value: cont ]
    ifFalse: [ ^ nil].
```

Hem realitzat el mateix codi en js per comprovar que efectivament treu el mateix resultat i per tant que el nostre mètode callcc a JavaScript usant la funció Continuation(), és correcta en aquest cas:

```
Continuation.callcc = function(aBlock) {
    return aBlock(new Continuation());
}
x = 0;
var lessThan6 = function() {
    return x < 6;
}
var printAndIncrement = function() {
    print("x = " + x);
    return ++x;
}
var whileTrue = function(booleanBlock, executionBlock) {
    var cont = Continuation.callcc(function(cc) {
        return cc;
    });
    if (booleanBlock()) {
        executionBlock();
        cont(cont);
    } else {
        return null;
    }
}
whileTrue(lessThan6, printAndIncrement);
```

Sortida:

```
x = 0
x = 1
x = 2
x = 3
x = 4
x = 5
```

2 Thread System

Per dur a terme la pràctica, el que vam començar fent va ser escriure i entendre bé que feia cada una de les funcions que es necessitaven:

Spawn

Aquesta funció, inicialment sembla senzilla i així va ser fins a gairebé al final. Simplement calia afegir un nou thread (amb un thunk) a la llista de threads del sistema ThreadSystem.

Start threads

Aquí només s'inicia l'execució de tots el threads.

Relinquish

Des d'un principi sabíem que hauríem d'utilitzar continuacions: Havíem d'"aturar" l'execució del Thread actual i continuar la del següent.

Quit

Finalment, en aquesta funció d'alguna manera (pista: Halt) havia d'acabar amb l'execució del thread i treure'l de la llista de threads.

Esquelet inicial

Vam veure clar que calia una cua d'execució. Hem utilitzat un simple array. Inicialitzant-lo buit this.threads = [], fent this.threads.push(thread) per afegir un nou thread i this.threads.shift() per eliminar el primer (pop).

While True

Després d'un parell de dies donant-li voltes, l'exemple de while true amb continuacions ens va fer veure la solució. En el while true, creem una continuació amb callcc que retorna el paràmetre que li passes:

```
cont := Continuation callcc: [ :cc | cc ].
```

```
var cont = Continuation.callcc(function(cc) {
    return cc;
});
```

A aquesta continuació, l'avaluàvem amb ella mateixa (una continuació), i per tant tornàvem on l'havíem inicialitzat però valent el mateix.

Així doncs, si controlem on es crea la continuació i on l'avaluàvem, podem controlar quin codi o thunk s'executa.

El problema és que després de la creació de la continuació s'executa el mateix codi... Ho podem evitar amb un booleà que ens indiqui si el thread està actiu o no.

Així doncs, un thread contarà de tres elements bàsics:

- **thunk**: Codi que executarà
- **isActive**: Booleà que ens indica si és el thread actiu o no
- **cont**: Continuació que ens permetrà restaurar l'execució

2.1 Primera versió

Spawn

```
var thread = {};
thread.thunk = thunk;
thread.isActive = false;
this.threads.push(thread);
```

En afegir un nou thread, aquest no està actiu, tampoc inicialitzem la continuació fins que no cridem *ThreadSystem.start_threads*. No volem que comenci l'execució.

Start threads

A l'hora de començar tots els threads activem tots amb un **forEach**, si no està activat, no fa res, en canvi, si està activat, comença l'execució del *thunk*. Clarament al principi de tot cap dels threads començarà el thunk, però a mesura que els anem activant en el **relinquish** aniran entrant al *thunk*.

També inicialitzem la continuació **halt**, única per tot el sistema, per sortir quan ja no hi ha més threads per executar. S'utilitza al **quit**.

```
this.threads.forEach(function(thread) {
    // init thread.cont
});

this.halt = Continuation.callcc(function(cc) { return cc });

// execute first thread, if exists
if (this.threads.length > 0) {
    this.threads[0].isActive = true;
    this.threads[0].cont(this.threads[0].cont);
}
```

Relinquish

Primer de tot desactivem el thread, perquè no hi hagi problemes no bucles infinits. Tot seguit assignem la nova continuació i, finalment, posem el thread al final de la cua i "sortim".

```
this.threads[0].isActive = false;
this.threads[0].cont = Continuation.callcc(function(cc) { return cc; });
if (!this.threads[0].isActive) {
    // push and quit threads[0]
}
```

El **if (!isActive)** sembla inútil perquè dues línies amunt l'hem fet **false**, però en realitat és molt important: Quan avaluem un altre cop la continuació, **isActive** serà cert i no entrarà al **if**, sortirà del 'relinquish' i continuarà l'execució del *thunk*.

Quit

Llegint el codi podem entendre fàcilment que fa: Elimina el primer thread (el que ha cridat **quit**) i, si hi ha algun thread a la cua, l'activem i avaluem la seva continuació, si no, anem a la continuació **halt** que havíem creat a l'inici de tot.

```
this.threads.shift();
if (this.threads.length > 0) {
    this.threads[0].isActive = true;
    this.threads[0].cont(this.threads[0].cont);
} else {
    this.halt();
}
```

2.2 Segona versió

Amb la versió inicial l'exemple del comptador funciona, però codi ens semblava massa lleig i vam decidir crear una segona versió amb l'objecte **Thread** i el mètode **Thread.prototype.activate** entre altres petites millores.

```
Continuation.current = function() {
    return Continuation.callcc((cc) => cc);
}

function Thread(thunk) {
    this.thunk = thunk;
    this.isActive = false;
}

Thread.prototype.activate = function() {
    this.isActive = true;
    this.cont(this.cont);
}
```

Exemple fibonacci

L'exemple de Fibonacci és una barreja entre el memorizer i l'algoritme del n-èssim Fibonacci recursiu:

```
var fibs = [];
function make_fib_thunk(n, thread_system) {
    function nFib() {
        if (n <= 1) {
            fibs[0] = 0;
            fibs[1] = 1;
            thread_system.quit();
        } else {
            thread_system.spawn(make_fib_thunk(n - 1, thread_system));
            while (fibs[n - 1] === undefined || fibs[n - 2] === undefined) {
                thread_system.relinquish();
            }
            fibs[n] = fibs[n - 1] + fibs[n - 2];
            thread_system.quit();
        }
    };
    return nFib;
}

var fib_thread_sys = make_thread_system();
fib_thread_sys.spawn(make_fib_thunk(9, fib_thread_sys));
fib_thread_sys.start_threads();
```

Aquest codi crea un nou thread per calcular el número de Fibonacci anterior si no és un cas base. Així doncs, es crida **spawn** un cop havent fet **start_threads**. Quin problema hi ha?

Spawn només afegeix el thread a la cua, però aquests inicialitzen la seva continuació en el **forAll** de **start_threads**.

2.3 Tercera versió

En aquesta versió hem mogut el codi d'inicialització de la continuació i l'execució inicial del *thunk* del **forAll** al mètode **spawn**:

```
var thread = new Thread(thunk)
this.threads.push(thread);
thread.cont = Continuation.current();

if (thread.isActive) {
    thread.thunk();
}
```

Ara ja podem executar l'exemple de Fibonacci sense cap bucle infinit ni cap error en intentar avaluar **cont** a un thread que no l'havia inicialitzat.