PRÀCTICA CAP:

Corutines en Rhino

Q1 2022-23

Integrants:
Alfonso Cano Pérez
Pol Pérez Castillo

Data d'entrega: 20-01-2023

Implementació de *make_coroutine*

En aquesta primera part de la pràctica se'ns demana investigar i implementar una estructura de control anomenada **Corutina**, però amb la diferència o la gràcia de fer-ho guardant la pila d'execució fent ús de les continuacions amb Rhino..

Per poder assolir els conceptes demanats ens hem ajudat del capítol 17 del llibre *Scheme and The Art Of Programming* proporcionat com a referència en l'enunciat de la pràctica, exactament en el capítol 17 apartat 6 : *Coroutines: Continuations in Action*.

El primer que hem de tenir clar és: **Què és una Corutina?**
En un programa, una Corutina és una estructura de control que, a diferència de les subrutines, mentres el retorn d'una subrutina és el final d'aquesta, el retorn d'una corrutina és el resultat de la suspensió del seu tractament fins que se l'indiqui reprendre la seva execució.

Aleshores per dur a terme la implementació d'aquesta estructura de control el primer que hem fet ha estat definir el mateix *callcc* que hem vist a classe. *Callcc* és una funció que té com a argument una altra funció *f*. El que fa primer és crear una instància de *Continuation*, que serveix per guardar l'estat actual del programa i permet tornar a aquest estat més endavant. Després es retorna l'execució de la funció *f* amb aquesta instància de *Continuation* com a argument.

Ara sí, la funció *make_coroutine* l'utilitzem per crear una Corutina. La idea principal és rebre una funció com a paràmetre, que anomenarem *coroutine* i tindrà el codi de la pròpia Corutina, i retornar una nova funció que podrà ser cridada per reprendre la Corutina dada.
Començant pel principi veiem la primera variable anomenada *savedContinuation* que és on es guardarà l'estat de la pròpia Corutina, que òbviament al principi de l'execució encara no hem guardat res i així ho indiquem. Després tenim una variable anomenada *updatedContinuation*, que ens permetrà actualitzar la continuació passada com a paràmetre. Després tenim una funció anomenada *resumer*, que s'encarrega de reprendre la Corutina (aquesta funció s'explica més detalladament més endavant). La última variable que declarem és un booleà anomenat *firstTime* que, com el propi nom diu, indica si és la primera vegada que es crida a la funció que es crea.
La funció que retorna *make_coroutine* és una funció amb només un paràmetre *value*. La primera vegada que es crida a aquesta funció, s'executa la funció *coroutine* (capturada perquè estem tractant amb closures) amb dos paràmetres com a arguments, el *resumer* i el *value*, tal i com ens diu l'enunciat de la pràctica que la Corutina ha de tenir aquests dos paràmetres. Després de la primera vegada, la Corutina no començarà de nou, si no que es reprendrà des d'on es va aturar anteriorment i se li passa només el *value* a la continuació.

La funció *make_resume* l'utilitzem per actualitzar la Corutina guardada i continuar amb la següent Corutina que s'hagi cridat. És una funció que rep com a argument la funció *updatedContinuation*, reanomenada ara com *updateCoroutine*, i retorna una nova funció.
Aquesta nova funció és la funció *resumer* mencionada anteriorment. Quan es crida a aquesta funció amb arguments *nextCoroutine* (la següent Corutina que es vol iniciar o reprendre) i *value* (valor que es retorna), primer cridem a *callcc* amb una funció *receiver* com a argument. Aquesta funció *receiver* actualizta la Corutina guardada utilitzant *updateCoroutine* i després ja es pot iniciar o reprendre la *nextCoroutine* pasant el valor com a paràmetre.

Tests fets

- **exemple_enunciat.js**

Com el seu propi nom indica és l'exemple que se'ns dona a l'enunciat de la pràctica. No hi ha res a afegir que no sigui que passa satisfactòriament el test.

- **test1.js**

En aquest test definim 4 Corutines, cadascuna de les quals imprimeix un número i després salta a la següent Corutina. Aquest test l'hem utilitzat per tornar a comprovar el bon funcionament de la nostra implementació aquest cop observant que una vegada ha arribat al final de la Corutina que acaba primer , tot i que hi hagi més codi en les altres, l'execució acaba igualment.

- **test2.js**

Aquest test és similar a l'anterior però té una petita diferència en l'assignació de les funcions. La funció que abans s'anomenava *b* ara és *c* i a l'inrevés. Aquest canvi l'hem fet per comprovar que era, satisfactòriament no hi ha cap Corutina que cridi a la nova *c*, i per tant, tots els prints que abans si es feien, ara ja no.

- **test3.js**

En aquest test comprovem una altra vegada que la crida a *resume* continua funcionant bé, cridant moltes vegades seguides a aquesta funció entre dues Corutines i, per últim, cridant-ne a una altra de diferent que només fa un print per comprovar que funciona.

Implementació de *same_fringe*

A la segona part de la pràctica se'ns demana d'implementar una funció anomenada **same_fringe**, que donats dos arbres binaris, digui si tenen la mateixa frontera, utilitzant les **corutines** implementades en la primera part.

Abans de fer res definirem el fet que donats dos arbres binaris, aquest tindran la mateixa frontera si tenen exactament les mateixes fulles, llegides d'esquerra a dreta.

Per enfocar aquest problema normalment l'estratègia més simple a seguir és recórrer les fulles dels dos arbres per separat i en ordre, generant dues llistes amb les fulles, i comparar aquestes llistes posteriorment. Aquesta solució requereix llegir els dos arbres sencers is emmagatzemar les dues llistes amb les fulles per a després comparar-les.

El que farem és explorar una solució que mitjançant les corutines pugi tant estalviar generar aquestes llistes, com terminar l'execució del programa quan es troba una fulla diferent, guanyant així tant en memòria com en temps d'execució.

Per la nostra solució hem optat per generar una corutina principal bastant simple, amb un bucle que anirà cridant a dues corutines més que s'encarregaran de llegir els arbres. Aquest bucle continuarà fins que es trobi la primera diferencia entre dues fulles, o fins que arribi al final (que serà el cas on tinguin les mateixes fulles).

Les dues corutines que llegiran els arbres tindran una estructura recursiva, que recorrerà els nodes dels arbres fins a arribar a una fulla (el cas base de la recursió) i en aquest punt tornarà a la corutina principal, retornant el valor de la fulla. Quan la corutina principal (de control) les torni a cridar, continuaran llegint el seu arbre on ho havien deixat. Cadascuna d'aquestes corutines, quan terminin de llegir tot l'arbre, posaran a true una variable global de la funció same_fringe que tenen associada. Aquestes variables només es posaran true a la vegada si els dos arbres tenen exactament les mateixes fulles, ja que en qualsevol altre cas, l'execució s'interromp i tot el que quedava per executar-se desapareix. Per tant, podem fer una comparació booleana && d'aquestes dues variables una vegada terminat el codi per saber si els dos arbres tenen les mateixes fulles.

El codi és el següent:

```
function same_fringe(tree1, tree2) {
    let tree1Ends = false;
    let tree2Ends = false;
    let c1 = make_coroutine( function(resume, value) {
        while(resume(c2,tree1) == resume(c3,tree2)){}
        print(tree1Ends && tree2Ends);
    });
    let c2 = make_coroutine( function(resume, value) {
        function recorrerArbol(arbol) {
            for (let i = 0; i < arbol.length; i++) {
                if (Array.isArray(arbol[i])) {
                    recorrerArbol(arbol[i]);
                } else {
                    resume(c1,arbol[i]);
                }
            }
        }
        recorrerArbol(tree1);
        tree1Ends = true;
    });
    let c3 = make_coroutine( function(resume, value) {
        function recorrerA(arbol) {
            for (let i = 0; i < arbol.length; i++) {
                if (Array.isArray(arbol[i])) {
                    recorrerA(arbol[i]);
                } else {
                    resume(c1,arbol[i]);
                }
            }
        }
        recorrerA(tree2);
        tree2Ends = true;
    });
    if (typeof(c1) === 'function') {
        c1('*')
    }
}
```

Tests fets

Pel que fa als tests d'aquesta part, tots estan definits testEj2_1.js, i els hem executat afegint una línia al codi a cada corutina que recorre els arbres, que imprimeix per consola el valor de cada fulla visitada i a quin arbre pertany, per poder comprovar fàcilment característiques com el fet que el programa pari l'execució en trobar la primera diferencia i no s'executi el que queda per recórrer. Hem definit els tests amb els següents criteris:

Els dos primers tests són dos exemples de casos en què la funció retornaria true, un quan es compara un arbre amb si mateix, i el segon, on es comparen dos arbres diferents, amb estructures de nodes diferents, però amb les mateixes fulles d'esquerra a dreta (mateixa frontera).

- **Test 1: Tree 1 == Tree 2 ⇒ True**
- **Test 2: Tree 1 != Tree 2, but same fringe ⇒ True**

```
Test 1: Tree 1 == Tree 2 -> (true)
tree1: 1
tree2: 1
tree1: 2
tree2: 2
tree1: 3
tree2: 3
tree1: 4
tree2: 4
tree1: 5
tree2: 5
tree1: 6
tree2: 6
tree1: 7
tree2: 7
true
```

```
Test 2: Tree 1 != Tree 2, but same fringe -> (true)
tree1: 1
tree2: 1
tree1: 2
tree2: 2
tree1: 3
tree2: 3
tree1: 4
tree2: 4
tree1: 5
tree2: 5
tree1: 6
tree2: 6
tree1: 7
tree2: 7
true
```

La resta son test que haurien de donar False, i a més a més, haurien de parar l'execució del programa independentment de quan es troba la diferència, i no executar el codi restant de les corutines. Per això hem definit test amb arbres amb diferent nombre de fulles, amb el mateix nombre de fulles, però trobant dues fulles diferents abans de temps (test 5), i un que just falla amb les últimes dues fulles, per veure tot funciona com deuria.

- **Test 3: Tree 1 with fewer leaves than Tree 2 ⇒ False**

```
Test 3: Tree 1 with fewer leaves than Tree 2 -> (false)
tree1: 1
tree2: 1
tree1: 2
tree2: 2
tree1: 3
tree2: 3
tree1: 4
tree2: 4
tree1: 5
tree2: 5
tree1: 6
tree2: 6
tree1: 7
tree2: 7
tree1: 8
false
```

- **Test 4: Tree 1 with more leaves than Tree 2 ⇒ False**

```
Test 4: Tree 1 with more leaves than Tree 2 -> (false)
tree1: 1
tree2: 1
tree1: 2
tree2: 2
tree1: 3
tree2: 3
tree1: 4
tree2: 4
tree1: 5
tree2: 5
tree1: 6
tree2: 6
tree1: 7
tree2: 7
tree2: 8
false
```

- **Test 5: Early difference ⇒ False**

```
Test 5: Early difference -> (false)
tree1: 1
tree2: 1
tree1: 2
tree2: 7
false
```

- **Test 6: Difference in last iteration ⇒ False**

```
Test 6: Difference in last iteration -> (false)
tree1: 1
tree2: 1
tree1: 2
tree2: 2
tree1: 3
tree2: 3
tree1: 4
tree2: 4
tree1: 5
tree2: 5
tree1: 6
tree2: 6
tree1: 7
tree2: 8
false
```

Els arbres utilitzats pels test son els següents:

```
let a1 = [ [ [ [1], [] ], [ [2], [ [3], [4] ] ] ], [ [ [], [5] ], [ [6], [7] ] ] ];
let a2 = [ [ [ [1], [2] ], [ [3], [4] ] ], [ [ [5], [6] ], [ [7], [] ] ] ];
let a3 = [ [ [ [1], [2] ], [ [3], [4] ] ], [ [ [5], [9] ], [ [7], [] ] ] ];
let a4 = [ [ [ [1], [2] ], [ [3], [4] ] ], [ [ [5], [6] ], [ [7], [8] ] ] ];
let a5 = [ [ [ [1], [] ], [ [2], [ [3], [4] ] ] ], [ [ [], [5] ], [ [6], [8] ] ] ];
let a6 = [ [ [ [1], [] ], [ [7], [ [3], [4] ] ] ], [ [ [], [5] ], [ [6], [7] ] ] ];
```