Lenguajes de Programación, 2021-2
Nota de clase 8: Continuaciones

Karla Ramírez Pulido Manuel Soto Romero

29 de julio de 2021
Facultad de Ciencias UNAM

8.1. Introducción

Una continuación se basa en el concepto de *materialización* de la pila de ejecución de programas. Esta técnica consiste en considerar la pila de ejecución de un programa como cualquier otro valor de un lenguaje de programación, es decir, se puede devolver o pasarse como argumento a funciones e incluso almacenarse en alguna estructura de datos. Esto permite entre otras cosas, dado un punto de ejecución, volver a un estado anterior o *continuar* con la ejecución del resto del programa [2].

Una *continuación* es una función de primera clase que representa el resto de la ejecución de un programa en un punto específico. Por ejemplo, en el Código 1 [2]:

```
;; potencias: number → number
(define (potencias n)
   (+ (expt n 2) (expt n 3) (expt n 4) (expt n 5)))
```

Código 1: Función potencias

- Si se detiene la ejecución de la función justo antes de calcular **`(expt n 4)`**, el programa habrá calculado **`(+ (expt n 2) (expt n 3))`** hasta ese momento.
- Cuando se haya calculado la operación **`(expt n 4)`**, la función *continuará* obteniendo el valor de **`(expt n 5)`** y lo sumará con el cálculo que ya tenía.
- De esta forma, se dice que la continuación de **`(expt n 4)`** es el resto de la ejecución del programa, es decir, el cómputo pendiente.

Para hacer más claro lo anterior, se puede reescribir la función potencias como se muestra en el Código 2, para hacer explícita la continuación, recordando que es una función [2]:

```
;; potencias: number → number
(define (potencias n)
   (let* ([ant (+ (expt n 2) (expt n 3))]
          [k (λ (v) (+ ant v (expt n 5)))])
      (k (expt n 4))))
```

Código 2: Función potencias con continuación explícita

La función que representa el cómputo pendiente es almacenada en la variable `k`. Se muestran a continuación otros ejemplos:

Ejemplo 8.1. Dado el Código 3.

```
;; foo: number → number
(define (bar x)
   (+ x 3))
```

Código 3: Función potencias con continuación explícita

Si se detiene la ejecución, justo antes de evaluar el número 3, ¿cuál es la continuación asociada en este punto?

Solución. La continuación se forma a partir del cálculo pendiente, en este caso, se debe sumar el cálculo que se llevaba, la suma de **x**. Sin embargo, después de esto no hay ninguna operación pendiente. Con lo cuál la continuación asociada es:

```
(λ (v) (+ x v))
```

□

Ejemplo 8.2. Dado el Código 4.

```
;; foo: number → number
(define (foo x y)
   (+ (* 2 x) (* 3 y) (/ 1 x) (/ 2 y)))
```

Código 4: Función potencias con continuación explícita

Si se detiene la ejecución, justo antes de calcular **`(/ 1 x)`** ¿cuál es la continuación asociada en este punto?

Solución. La continuación se forma a partir del cálculo pendiente, en este caso, se debe sumar el cálculo que se llevaba **`(+ (* 2 x) (* 3 y))`** con el valor **`(/ 2 y)`**. Con lo cual la continuación asociada es:

```
(λ (v) (+ (* 2 x) (* 3 y) v (/ 2 y)))
```

□

8.2. Programando con continuaciones

Lenguajes de programación como Racket, incluyen primitivas para capturar la continuación actual en un punto específico de la ejecución de un programa. En Racket, es posible hacer esto mediante las primitivas **`call/cc`** (*call with current continuation*) y **`let/cc`** (*let the current continuation*).

call/cc La sintaxis de la primitiva `call/cc` es la siguiente [3]:

```
(call/cc
   (λ (k)
      ... k ...))
```

Es decir, `call/cc` toma una función cuyo parámetro es la continuación actual y puede usarla en su cuerpo para obtener algún valor. Por ejemplo, se podría reescribir el Código 1 como en el Código 5 para hacer la continuación de `(expt n 4)` como se aprecia en la línea 6.

```
1 ;; potencias: number → number
2 (define (potencias n)
3    (+
4       (expt n 2)
5       (expt n 3)
6       (call/cc (λ (k) (k (expt n 4))))
7       (expt n 5)))
```

Código 5: Función potencias usando call/cc

let/cc La primitiva `let/cc` funciona como una asignación local cuyo identificador es la continuación actual y en cuyo cuerpo se describe qué hacer con ésta. Su sintaxis es la siguiente [3]:

```
(let/cc k
   ... k ...)
```

Por ejemplo, se puede reescribir la función del Listado de código 1 como en el Listado de código 6 usando `let/cc`.

```
1 ;; potencias: number → number
2 (define (potencias n)
3    (+
4       (expt n 2)
5       (expt n 3)
6       (let/cc k (k (expt n 4)))
7       (expt n 5)))
```

Código 6: Función potencias usando call/cc

8.3. Escapes

¿Cómo deberían evaluarse los Códigos 5 y 6? A continuación se muestra la idea intuitiva de su evaluación.

Evaluación del Código 5 con entrada `7`*:*

```
> (potencias 7)

(+
   (expt 7 2)
   (expt 7 3)
   (call/cc (λ (k) (k (expt 7 4))))
   (expt 7 5))

(+
   49
   343
   ((λ (v) (+ 49 343 v 16807)) 2401)
   16807)

(+
   49
   343
   (+ 49 343 2401 16807)
   16807)

(+
   49
   343
   19600
   16807)

36799
```

Evaluación del Código 6 con entrada **7***:*

```
> (potencias 7)

(+
   (expt 7 2)
   (expt 7 3)
   (let/cc k (k (expt 7 4)))
   (expt 7 5))

(+
   49
   343
   ((λ (v) (+ 49 343 v 16807)) 2401)
   16807)

(+
   49
   343
   (+ 49 343 2401 16807)
   16807)
```

```
(+
   49
   343
   19600
   16807)

36799
```

Sin embargo al ejecutar en DrRacket las funciones de los Códigos 1, 5 y 6 se obtienen los siguientes resultados:

```
> (potencias 7) ;; Código 1
19600
> (potencias 7) ;; Código 5
19600
> (potencias 7) ;; Código 6
19600
```

La ejecución (hecha a mano) muestra resultados distintos con respecto a la ejecución de Racket, pues se considera la continuación asociada como si fuera cualquier otra función del lenguaje. Es decir, al aplicarse sólo devuelve un resultado. Sin embargo, al aplicar una continuación, en Racket, ésta detendrá la ejecución de la expresión actual y devolverá su evaluación como resultado.

Para no causar confusiones suele usarse la notación $\lambda\uparrow$, para distinguir entre continuaciones y el resto de funciones del lenguaje. Estas funciones también reciben el nombre de *escapes* pues al aplicarse, se finaliza la ejecución del programa [3]. De esta forma, al evaluar:

```
(+
   (expt 7 2)
   (expt 7 3)
   (call/cc (λ (k) (k (expt 7 4))))
   (expt 7 5))
```

o

```
(+
   (expt 7 2)
   (expt 7 3)
   (let/cc k (k (expt 7 4)))
   (expt 7 5))
```

la continuación asociada, usando la notación $\lambda\uparrow$ queda como sigue:

```
(λ↑ (v) (+ 49 343 v 16807))
```

Con lo cual al aplicar:

```
(λ↑ (v) (+ 49 343 v 16807)) 2401)
```

se obtiene

19600 y finaliza la ejecución del programa.

8.4. Estilo de Paso de Continuaciones[1]

El Estilo de Paso de Continuaciones (CPS) es una técnica ampliamente usada por algunos lenguajes de programación funcionales que permite, entre otras cosas, optimizar el proceso de compilación de algunos lenguajes de programación. La idea detrás de esta técnica consiste en pasar una continuación como argumento adicional a las funciones con el fin de llevar consigo toda la información necesaria para que la ejecución del programa continúe después de aplicar dicha continuación[1].

En este sentido, la definición de las funciones que siguen este estilo, consiste en modificar la continuación asociada llamada tras llamada y llevar la recursión mediante llamadas de cola. Al igual que en el proceso de transformación a recursión de cola, es necesario modificar la definición original para que use este nuevo parámetro.

En general, se recomiendan seguir los siguientes pasos para obtener una conversión a CPS óptima:

1. Convertir la función recursiva a una versión recursiva de cola.
2. Modificar el caso base para que aplique la continuación recibida como argumento al valor original.
3. Modificar las llamadas recursivas para que construyan una nueva función (continuación) de manera similar a cuando modificábamos el acumulador en las versiones de cola. La acumulación se da en este caso aplicando la continuación actual con la operación pendiente.

A continuación se revisan algunos ejemplos:

Ejemplo 8.3. Suma de los elementos de una lista.

Versión original

```
1 ;; suma-list: (listof number) → number
2 (define (suma-list lst)
3    (match lst
4       ['() 0]
5       [(cons x xs) (+ x (suma-list xs))]))
```

[1]Del inglés *Continuation Passing Style (CPS)*.

Ejecución original

```
(suma-list '(1 2 3)) =
(+ 1 (suma-list '(2 3))) =
(+ 1 (+ 2 (suma-list '(3))) =
(+ 1 (+ 2 (+ 3 (suma-list empty)))) =
(+ 1 (+ 2 (+ 3 0))) =
6
```

Versión de cola

```
1 ;; suma-list: (listof number) → number
2 (define (suma-list lst)
3    (suma-list-cola lst 0))
4
5 ;; suma-list: (listof number) number → number
6 (define (suma-list-cola lst acc)
7    (match lst
8        ['() acc]
9        [(cons x xs) (suma-list-cola xs (+ x acc))]))
```

Código 7: Suma de los elementos de una lista

Versión usando CPS

```
1 ;; suma-list: (listof number) → number
2 (define (suma-list lst)
3    (suma-list-cps lst (λ (x) x)))
4
5 ;; suma-list: (listof number) procedure → number
6 (define (suma-list-cps lst k)
7    (match lst
8       ['() (k 0)]
9       [(cons x xs) (suma-list-cps xs (λ (v) (k (+ x v))))]))
```

Código 8: Suma de los elementos de una lista

Ejecución usando CPS

#	Llamada	Continuación
1	(suma-list '(1 2 3))	
2	(suma-list-cps '(1 2 3) k1)	k1 = (λ (x) x)
3	(suma-list-cps '(2 3) k2)	k2 = (λ (v) (k1 (+ 1 v)))
4	(suma-list-cps '(3) k3)	k3 = (λ (v) (k2 (+ 2 v)))
5	(suma-list-cps '() k4)	k4 = (λ (v) (k3 (+ 3 v)))
6	(k4 0)	
7	(k3 (+ 3 0))	
8	(k2 (+ 2 (+ 3 0)))	
9	(k1 (+ 1 (+ 2 (+ 3 0))))	
10	(+ 1 (+ 2 (+ 3 0))))	
11	6	

Algunas observaciones sobre el Código 8 y su ejecución:

- La línea 3 muestra la llamada a la función optimizada mediante CPS. Se usa en este caso la función identidad `(λ (x) x)` a manera de ejemplo, sin embargo, cuando se usa esta técnica para efectos de optimización u en otros procesos suele tomarse la continuación actual, por ejemplo, capturada con las primitivas `let/cc` y `cal/cc` de Racket.
- El caso base de la función, se vuelve una aplicación de función. Esto quiere decir que cuando se llegue a un caso base, comienza una aplicación de funciones para obtener el valor a partir del estado del programa que se fue cargando llamada tras llamada.
- Si se cae en el caso recursivo, se hace una llamada de cola (sin llamadas pendientes), y se construye una nueva continuación que conoce el estado anterior del programa (la función `k` recibida como parámetro) y el cálculo pendiente (sumar la cabeza de la lista).
- La ejecución muestra la construcción de las funciones llamada a llamada y la aplicación de funciones encadenada al caer en un caso base. □

Ejemplo 8.4. Función que filtra los números positivos de una lista.

Versión original

```
1 ;; filtra-pos: (listof number) → (listof number)
2 (define (filtra-pos lst)
3    (match lst
4       ['() lst]
5       [(cons x xs)
6           (if (> x 0)
7               (cons x (filtra-pos xs))
8               (filtra-pos xs))]))
```

Ejecución original

```
(filtra-pos '(-1 4 -2 3)) =
(filtra-pos '(4 -2 3)) =
(cons 4 (filtra-pos '(-2 3))) =
(cons 4 (filtra-pos '(3))) =
(cons 4 (cons 3 (filtra-pos empty))) =
(cons 4 (cons 3 empty)) =
'(4 3)
```

Versión de cola

```
1 ;; filtra-pos: (listof number) → (listof number)
2 (define (filtra-pos lst)
3    (filtra-pos-cola lst '()))
4
5 ;; filtra-pos-cola: (listof number) (listof number) → (listof number)
```

```
6  (define (filtra-pos-cola lst acc)
7     (match lst
8        ['() acc]
9        [(cons x xs)
10           (if (> x 0)
11               (filtra-pos-cola xs (append acc (list x)))
12               (filtra-pos-cola xs acc))]))
```

Código 9: Suma de los elementos de una lista

Versión usando CPS

```
1  ;; filtra-pos: (listof number) → (listof number)
2  (define (filtra-pos lst)
3     (filtra-pos-cps lst (λ (x) x)))
4
5  ;; filtra-pos-cps: (listof number) procedure → (listof number)
6  (define (filtra-pos-cps lst k)
7     (match lst
8        ['() (k '())]
9        [(cons x xs)
10           (if (> x 0)
11               (filtra-pos-cps xs (λ (v) (k (cons x v))))
12               (filtra-pos-cps xs k))]))
```

Código 10: Filtra los valores positivos en una lista

Ejecución usando CPS

#	Llamada	Continuación
1	`(filtra-pos '(-1 4 -2 3))`	
2	`(filtra-pos-cps '(-1 4 -2 3) k1)`	`k1 = (λ (x) x)`
3	`(filtra-pos-cps '(4 -2 3) k1)`	`k2 = (λ (v) (k1 (cons 4 v)))`
4	`(filtra-pos-cps '(-2 3) k2)`	`k3 = (λ (v) (k2 (cons 3 v)))`
5	`(filtra-pos-cps '(3) k2)`	
6	`(filtra-pos-cps '() k3)`	
7	`(k3 '())`	
8	`(k2 (cons 3 '())`	
9	`(k1 (cons 4 (cons 3 '())))`	
10	`(cons 4 (cons 3 '()))`	
11	`'(4 3)`	

□

Ejemplo 8.5. Función que aplica una función a los elementos de una lista (`map`).

Versión original

```
1 ;; mapea: procedure (listof any) → (listof any)
2 (define (mapea f lst)
3    (match lst
4       ['() lst]
5       [(cons x xs) (cons (f x) (map f xs))]))
```

Ejecución original

```
(mapea add1 '(1 7 2 9)) =
(cons (add1 1) (mapea add1 '(7 2 9))) =
(cons 2 (cons (add1 7) (mapea add1 '(2 9)))) =
(cons 2 (cons 8 (cons (add1 2) (mapea add1 '(9))))) =
(cons 2 (cons 8 (cons 3 (cons (add1 9) (mapea add1 '()))))) =
(cons 2 (cons 8 (cons 3 (cons 10 '())))) =
'(2 8 3 10)
```

Versión usando CPS

```
1 ;; mapea: procedure (listof any)  → (listof any)
2 (define (mapea f lst)
3    (mapea-cps f lst (λ (x) x)))
4
5 ;; mapea-cps: procedure (listof any) procedure → (listof any)
6 (define (mapea-cps f lst k)
7    (match lst
8       ['() (k lst)]
9       [(cons x xs) (mapea-cps f xs (λ (v) (k (cons (f x) v))))]))
```

Código 11: Aplica una función a los elementos de una lista

Ejecución usando CPS

#	Llamada	Continuación
1	`(mapea add1 '(1 7 2 9))`	
2	`(mapea-cps add1 '(1 7 2 9) k1)`	`k1 = (λ (x) x)`
3	`(mapea-cps add1 '(7 2 9) k2)`	`k2 = (λ (v) (k1 (cons 2 v)))`
4	`(mapea-cps add1 '(2 9) k3)`	`k3 = (λ (v) (k2 (cons 8 v)))`
5	`(mapea-cps add1 '(9) k4)`	`k4 = (λ (v) (k3 (cons 3 v)))`
6	`(mapea-cps add1 '() k5)`	`k5 = (λ (v) (k4 (cons 10 v)))`
7	`(k5 '())`	
8	`(k4 (cons 10 '()))`	
9	`(k3 (cons 3 (cons 10 '())))`	
10	`(k2 (cons 8 (cons 3 (cons 10 '()))))`	
11	`(k1 (cons 2 (cons 8 (cons 3 (cons 10 '())))))`	
12	`(cons 2 (cons 8 (cons 3 (cons 10 '()))))`	
13	`'(2 8 3 10)`	

□

Esta forma de definir funciones permite por ejemplo, que éstas verifiquen la información de un programa que está cargando la función y tomen acciones de mitigación o de recuperación. Algunos ejemplos de uso de este estilo de programación es el manejo de excepciones en lenguajes de programación o la implementación de transacciones en Bases de Datos.

Referencias

[1] Favio E. Miranda, Lourdes del C. González, *Lenguajes de Programación*, Notas de clase, Facultad de Ciencias UNAM, Revisión 2019-1.

[2] Manuel Soto Romero, *Manual de Prácticas para la Asignatura de Programación Declarativa*, Proyecto de Apoyo a la Docencia, Facultad de Ciencias UNAM, 2019.

[3] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, 2007. [En línea: `https://plai.org/`].