Continuaciones en Scheme

Luis Mateu

Definición

- Una continuación es la abstracción de la cadena de llamadas de procedimientos.
- En Scheme se trata de un objecto funcional con un solo parámetro.
- Toda invocación de función recibe un argumento implícito que es su continuación.
- Por ejemplo: f -> g -> h. La continuación de h es una función que ejecuta el retorno a la función g.

En otros lenguajes

- C: *setjmp*, *longjmp*. El destino debe existir, si no, *segmentation fault*.
- Java: *try-catch*, *throw*. Garantiza que el destino existe.
- Pero en Scheme las continuaciones son ***mucho*** más poderosas

Continuaciones de primera clase

- En Scheme las continuaciones pueden ser de primera clase: se pueden almacenar en variables, vectores, pasar como argumentos y ser retornados.
- Esto se hace con:

```
(call-with-current-continuation proc)
```

- En donde proc debe tener la forma:

```
(lambda (kont) ...)
```

- "**`kont`**" es la continuación de 1era. clase.

Uso: Reificación

- Se usan para lograr escapes dinámicos:

```
(call/cc    ; call-with-current-continuation
  (lambda (exit-fun)
    ...
    (lambda (...)
      ...
      (lambda (...)
        ...
        (exit-fun val) ))))
```

- Se usan para lograr corrutinas

Ejemplo: búsqueda exhaustiva

```
(define (full-search sym tree)
  (call/cc
    (lambda (kont)
      (define (search tree)
        (cond
          ((eq? sym tree) (kont #t))
          ((pair? tree)
                   (search (car tree))
                   (search (cdr tree)) )))
      ; cuerpo del lambda
      (search tree)
      #f )))
```

Variante: return

```
(define (full-search sym tree)
  (call/cc
    (lambda (return)
      (define (search tree)
        (cond
          ((eq? sym tree) (return #t))
          ((pair? tree)
                   (search (car tree))
                   (search (cdr tree)) )))
      ; cuerpo del lambda
      (search tree)
      #f )))
```

Variante: paso como parámetro

```
(define (search sym tree kont)
  (cond
    ((eq? sym tree) (kont #t))
    ((pair? tree)
            (search sym (car tree) kont)
            (search sym (cdr tree) kont) )))
(define (full-search sym tree)
  (call/cc
    (lambda (kont)
      (search syn tree kont)
      #f )))
```

Múltiples invocaciones

```
(let
  ( (v 1)
    (kont '()) )
  (set! kont (call/cc (lambda (k) (k k))))
  (display v) (newline)
  (set! v (+ v 1))
  (if (<= v 10)
    (kont kont) ))
```

- No se entiende? Sin embargo la semántica es una sola!
- Abre un sin número de posibilidades para escribir programas ilegibles!

Múltiples invocaciones

```
(let
  ( (v 1)
    (kont (call/cc (lambda (k) (k k)))) )
  (display v) (newline)
  (set! v (+ v 1))
  (if (<= v 10)
    (kont kont) ))

; equivale a:
( (lambda (v kont)
     (display v) (newline)
     (set! v (+ v 1))
     (if (<= v 10)
       (kont kont) ))
  (1 (call/cc (lambda (k) (k k)))) )
```

- Las implementaciones no necesariamente implementan correctamente call/cc.

Historia

- Las primeras versiones de Scheme solo tenían escapes en la forma de catch/throw.
- Idea: como especificar la semántica de Scheme? En particular, la semántica de catch/throw.
- Solución: traducir los programas a una máquina abstracta.
- Todas las implementaciones deben entregar el mismo resultado que la máquina abstracta.
- Lenguaje de la máquina abstracta: lambda cálculo.

Problema: efectos de borde

- En lambda cálculo no hay efectos de borde.
- Solución:
  - Mapa de asociación entre variables y valores
  - El mapa se pasa como argumento a todas las funciones
  - La asignación crea un nuevo mapa a partir del mapa actual.
  - Cómo se retorna?

Problema: escapes

- Lambda cálculo no tiene escapes catch/throw.
- Solución: estilo de paso de continuaciones (CPS)
  - Los procedimientos nunca retornan
  - Reciben un parámetro adicional: la continuación
  - En vez de retornar, se invoca la continuación

Ejemplo: fib

```
(define (fib n) ; estilo normal
  (if (<= n 2)
    1
    (+ (fib (- n 1) (fib (- n 2))) ))

(define (kfib n k) ; estilo cps
  (if (<= n 2)
    (k 1)
    (kfib (- n 1)
      (lambda (res1)
        (kfib (- n 2)
          (lambda (res2)
            (k (+ res1 res2) ))))))
```

- No se entiende nada?

Ejemplo: búsqueda exhaustiva (2)

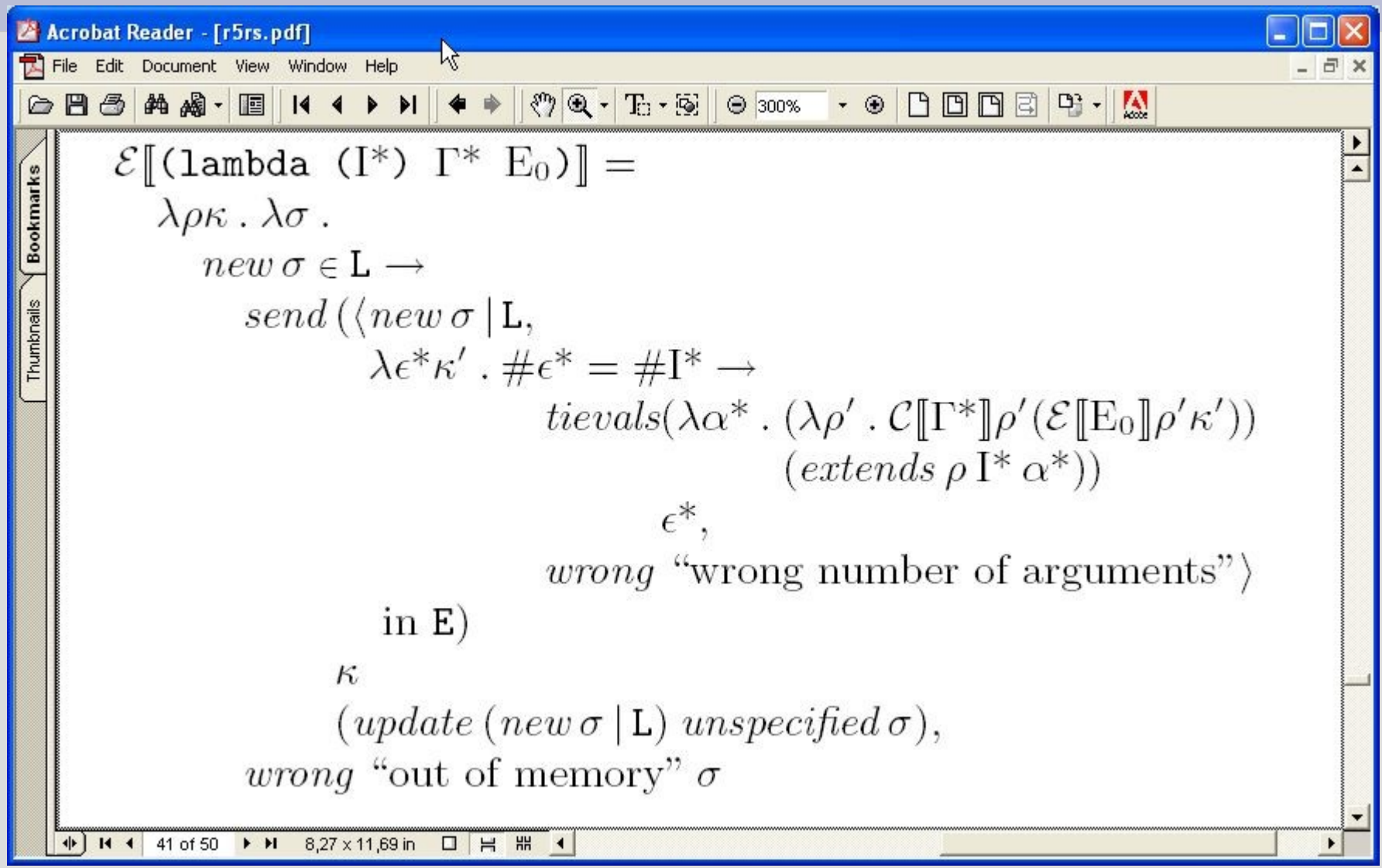
```
(define (full-search sym tree k1)
  (define (search tree k2)
    (cond
      ((eq? sym tree) (k1 #t))
      ((pair? tree)
        (search (car tree)
          (lambda (dummy)
            (search (cdr tree) k2) )))
      (else (k2 'void)) ))
  ; cuerpo de full-search
  (search tree (lambda (val) (k1 #f))) )
```

- No se entiende, pero no hay ambigüedad sobre como implementarlo!

Implementación de referencia

- La implementación de referencia traduce los programas a CPS (*continuation passing style*).
- Implementa los efectos laterales de manera funcional por medio de una mapa de asociaciones (*look up table*).
- Una implementación real nunca lo hace así, pero debe entregar el mismo resultado.
- La implementación de referencia define la *semántica* del lenguaje.
- En Scheme, esta implementación de referencia se llama semántica denotacional.

Semántica denotacional



Implementación de referencia de call/cc

- Tan simple como:

```
(define (call/cc proc kont)
  (proc kont kont) )
```

- La idea era sólo especificar catch/throw, pero esta definición también sirve para dar un significado a continuaciones con:
  - invocaciones hacia arriba
  - múltiples invocaciones!

Corrutinas

- Procesos con paso explícito del control (non preemptive)
- **`(define c (start proc))`**: crea una corrutina.
- **`(resume c)`**: transfiere el control a la corrutina c.
- **`(current)`**: entrega el identificador de la corrutina en ejecución
- Ejemplo:
  - `(same-fringe '(a (b c)) '((a b) c)) => true`

Implementación

```
(define *eot* '(eot))   ; fin del árbol
(define (same-fringe tree-a tree-b)
  ; Crea una corrutina para cada árbol
  (let ((t-a (make-walker tree-a))
        (t-b (make-walker tree-b)))
    ; Obtiene cíclicamente una hoja para cada árbol
    (let loop ((leaf-a (resume t-a 'any))
               (leaf-b (resume t-b 'any)))
      (cond
        ((eq? leaf-a *eot*)
           ; no more leaves on tree-a
           (eq? leaf-b *eot*))
        ((eq? leaf-a leaf-b)
           ; get a new leaf for each
           ; tree and loop
           (loop (resume t-a 'any) (resume t-b 'any)))
        (else
           ; Encontró dos hojas diferentes
           #f) ))))
```

Implementación (cont)

```
(define (make-walker tree)
  (start
    (lambda (t-comp)
      ; t-comp es el identificador del padre
      (define (walk-tree tree)
        ; Recorre el árbol
        (if (atom? tree)
          (resume t-comp tree)
          ; else
          (begin
            (walk-tree (car tree))
            (walk-tree (cdr tree)) )))
      ; pass my thread-id
      (resume t-comp (current-thread))
      ; now traverse the tree
      (walk-tree tree)
      ; Finally, pass the end-token
      *eot* )))
```

Como implementar call/cc

- Directamente: transformar a CPS
  - No hay pila: todas las llamadas son recursivas por la cola.
  - Las variables están en el heap.
  - Ineficiente.
- Duplicar las pilas
  - Para respetar la semántica de call/cc se requiere que toda variable asignada con set! quede en el heap.
  - No tan eficiente.
  - La mayoría de las implementaciones lo hace, pero algunas no.

Conclusiones

- Call/cc otorga una expresividad inigualable a Scheme: escapes + corrutinas < call/cc
- Pero es difícil implementar eficientemente las invocaciones múltiples de una continuación.
- Piquer y yo propusimos una semántica de corrutinas para Scheme en donde:

  escapes + corrutinas = call/cc

- Es compatible con call/cc, excepto en las invocaciones múltiples.

Bibliografía

- Draft paper “Continuations: Multiple Invocations Considered Harmful”, Mateu y Piquer.
- Contiene un survey de artículos acerca de continuaciones.