

Continuations

Motivação

Programas para *web* costumam apresentar vários *bugs*, como este:

- Fulano procura um hotel em um serviço
- Servidor propõe hotéis **A** e **B**
- Fulano abre **A** em uma janela separada
- Fulano volta para a primeira janela e abre **B**
- Fulano usa a janela de **A**, ainda aberta, para a reserva
- Servidor reserva **B**!

Erro do programador ou ambiente?

Pode-se colocar um preceito:

“O usuário deve sempre ter a resposta de acordo com a página em uso”

Mas em uma loja virtual? Não se quer perder o que já se colocou no carrinho!!! Não é apenas o botão clicado que manda.

Os *browsers* não avisam a aplicação de todos os passos seguidos pelo usuário (criação de *bookmarks*, avanços e retrocessos na página, etc). É tudo baseado apenas nos eventos enviados.

A arquitetura *Web* faz com que cada passo concluído termine o programa. Cada *click* dispara uma nova aplicação. É um protocolo sem estado (*stateless*).

Protocolos com e sem estado

Exemplos de protocolo COM estado:

- ftp
- smtp
- ssh

São mais fáceis de programar, pois o *contexto* é mantido no servidor.

Para protocolos com estado, a carga no servidor é maior.

Protocolos sem estado são leves, mas o servidor precisa reconstruir o contexto quando necessário.

Pense em um jogo de xadrez pelo correio.

Exemplo simples, mas complicado

Pedir 2 números e somá-los, via *web*. Supondo que existam funções prontas:

```
(web-display  
  (+ (web-read "First number: ")  
     (web-read "Second number: ")))
```

São 3 programas!!!!

- primeira leitura
- segunda leitura
- soma e apresentação

O estado deve ser transmitido. Normalmente com "*hidden fields*". Mas se for via *weblets*, os valores podem ser guardados em um banco de dados ou em um objeto de sessão. Pode haver conflito se várias janelas forem abertas.

Este programa exemplo nem tão simples, não funciona, pois cada subprograma termina assim que sua parte é feita.

Após a primeira leitura, por exemplo, o que falta ser feito (*pending computations*) é descartado junto com o valor lido.

Deve-se guardar o contexto ou retransmiti-lo.

O QUE FALTA SER FEITO

```
(web-display
  (+ já feito
    (web-read "Second number: ")))
```

Isso pode ser escrito de outra forma

```
(λ (•)
  (web-display
    (+ •
      (web-read "Second number: "))))
```

O segundo passo fica:

```
(λ (•2)
  (web-display
    (+ •1 •2)))
```

O segundo passo pode ser:

```
(λ (•2)  
  (web-display  
    (+ •1  
      •2 )))
```

”•₁” está no fechamento deste lambda.

UMA PRIMITIVA MELHOR: _____

`web-read` recebe um *prompt* e lê um valor, via `http`. `web-read/k` recebe o `prompt` e a continuação:

1. gera uma chave `C` e guarda em um hash o par `(C, continuação)`
2. gera a página com o `prompt`, a entrada e um `submit` com a chave `C` embutida
3. envia a chave e termina.

A aplicação chamada pelo *submit* pega a *closure* na tabela de hash e passa o valor lido.

```
(web-display
  ( + (web-read/k "First number:"
    (lambda (•1) •1))
    (web-read "Second number:"))))
```


Mas ainda termina antes da hora! Podemos jogar o web-display para o final:

```
(web-read/k "First number:"  
  (lambda (•1)  
    (web-read "Second number:"  
      (lambda (•2)  
        (web-display  
          (+ •1 •2))))))
```

EVITANDO AS CONTINUAÇÕES

Isto é, sem usar o `web-read/k` Usa-se o `web-read/r` -> que passa o nome do procedimento! `(web-read/r "First number:f1")`

`(define (f1 •1) (web-read/r "Second number:f2"))`

`(define (f2 •2) (web-display (+ •1 •2)))`

Quem é •₁ em f2????????

Não dá para passar mais de um argumento, mas pode-se passar a página com campos escondidos.

Testando

Para rodar dentro do Scheme, fingindo estar enviando e recebendo:

Provoca um erro, mas guarda a descrição da continuação em um *box* referenciado no *closure*.

Transformações

As mudanças para o formato “invertido” pode ser automatizada:

tally recebe uma lista, pede os preços de cada item e calcula o total.

```
(define (tally/k item-list k)  
  (if (empty? item-list)  
      0  
      (+ (web-read (generate-item-cost-prompt (first item-list)))  
         (tally (rest item-list))))))
```

Primeiro passo: mudar a função

Segundo passo: mudar wread

```
(define (tally/k item-list k)
  (if (empty? item-list)
      (k 0)
      (+ (web-read/k (generate-item-cost-prompt (first item-list)))
          (lambda (primeiro)
              (tally (rest item-list)
                      (lambda (resto)
                          (k (+ primeiro resto))))))))))
```

Note que **k** deve ser aplicado ao resultado da chamada.

Chamando mais de uma função

Pede o preço e depois o custo de envio (S&H)

```
(define (total+s&h item-list)
  (local ([define total (tally item-list)])
    (+ (web-read (generate-s&h-prompt total))
       total)))
```

Colocando na forma de pendência:

```
(define (total+s&h/k item-list k)
  (local ([define total (tally/k item-list ???)])
    (+ (web-read (generate-s&h-prompt total))
       total)))
```

transformando

Primeiro chama tally

```
(define (total+s&h/k item-list k)
  (tally/k item-list
    (lambda (tally-of-items)
      ??? )))
```

Calcula total

```
(define (total+s&h/k item-list k)
  (tally/k item-list
    (lambda (tally-of-items)
      (local ([define total tally-of-items])
        ??? ))))
```

```
(define (total+s&h/k item-list k)
  (tally/k item-list
    (lambda (tally-of-items)
      (local ([define total tally-of-items])
        (web-read/k (generate-s&h-prompt total)
          (lambda (s&h-amount)
            (k (+ s&h-amount
                  total))))))))))
```

Note a chamada de k no nível mais interno, é k quem trata o valor final!

Estado

```
(define account
  (local ([define balance 0])
    (lambda ()
      (begin
        (set! balance (+ balance
                           (web-read
                            (format "Balance: ~a; Change" balance))))
        (account)))))
```

Transformação

Para transformar, como sempre, identifica-se o resto da computação:

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (set! balance (+ balance
                          (web-read
                           (format "Balance:  ~a; Change" balance))))
        (account/k ??? ))))))
```

Agora, a ordem das computações

A leitura é a primeira coisa _____

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (web-read/k (format "Balance:  ~a; Change" balance))
        (lambda (v)
          (begin
            (set! balance (+ balance v))
            (account/k ??? ))))))))
```

O que falta ser feito?

Apenas o que faltava antes da chamada _____

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (web-read/k (format "Balance:  ~a; Change" balance)
          (lambda (v)
            (begin
              (set! balance (+ balance v))
              (account/k k))))))))))
```

Nota: na *closure* está guardada a localização, o valor fica na memória do servidor

A transformação em si _____

Basicamente, o que acontece é o seguinte:

Dado que existe:

$$(f \ v_1 \ v_2 \ \dots \ v_m) \rightarrow X$$

A forma

$$(f \ v_1 \ v_2 \ \dots \ v_m \ k)$$

aplica k a X

Funções de ordem superior

Para esta aplicação:

```
(define (get-one-temp c)
  (web-read (format "Temperature in city ~a" c)))
(web-display
 (average
  (map get-one-temp
       (list "Bangalore" "Budapest" "Houston" "Providence"))))
```

(average está pré-definida e não interage com a web)

get-one-temp é trivial:

```
(define (get-one-temp/k c k)
  (web-read/k (format "Temperature in city ~a" c)
              k))
```

Chamando a versão modificada _____

```
(web-display
  (average
    (map get-one-temp/k
      (list "Bangalore" "Budapest" "Houston" "Providence")))))
```

Não funciona! Cadê o k no map?

Para fazer um map/k, vamos escrever um map:

```
(define (map f l)
  (if (empty? l)
      empty
      (cons (f (first l))
            (map f (rest l)))))
```

Tentativa para a continuação

```
(define (map f l)
  (if (empty? l)
      empty
      (cons (f (first l) (lambda (x) x))
            (map f (rest l))))))
```

Não resolve, porque a identidade termina imediatamente!

Podemos percorrer a lista na continuação:

```
(define (map f/k l)
  (if (empty? l) empty
      (f/k (first l) (lambda (v)
                      (cons v (map f (rest l)))))))
```

Mas isso para na segunda chamada (por quê?)

Com um map/k de verdade _____

```
(define (map/k f/k l k)
  (if (empty? l)
      empty
      (f/k (first l)
            (lambda (v)
              (cons v (map/k f/k (rest l) ??? ))))))))
```

Vai ter uma chamada a `web-read/k` no meio, parando o programa e perdendo o contexto.

O `cons` deve ser feito depois.

Completando

```
(define (map/k f/k l k)
  (if (empty? l)
      empty
      (f/k (first l)
            (lambda (v)
              (map/k f/k (rest l)
                    (lambda (v-rest) (cons v v-rest))))))))
```

Programa principal:

```
(map/k get-one-temp/k
      (list "Bangalore" "Budapest" "Houston" "Providence")
      (lambda (v) (web-display (average v))))
```

Resulta na lista vazia!!!!!!

k deve ser aplicado a todos os resultados! _____

```
(define (map/k f/k l k)
  (if (empty? l)
      (k empty)
      (f/k (first l)
            (lambda (v)
              (map/k f/k (rest l)
                    (lambda (v-rest) (k (cons v v-rest))))))))))
```

Procedimento

Os passos são os seguintes:

- Gerar continuações para as computações pendentes.
- Passar os valores para as continuações ao invés de retorná-los.
- Em alguns casos, a continuação é apenas passada adiante (`get-one-temp`)

Implicações:

- Decisões de ordem de cálculo.
- Transformações de funções podem se propagar a outras.
- O programa passa a ter uma forma procedural e não funcional!

Continuações × Direto ---

As continuações tornam explícitas certas coisas:

- Retorno de procedimentos → Chamada de continuação
- Valores intermediários → Recebem nomes (são argumentos da continuação)
- Ordem de avaliação de argumentos → Fica explícita (bem definida)
- Recursão de cauda → Repasse da continuação

Transformação automática: um compilador *Web*

A chave está no *receiver*. O que é ele?

Para cada expressão e , o *receiver* é um procedimento que diz como a computação deve prosseguir quando e for concluída.

Fato curioso: não há a necessidade de retornar valores, pois eles sempre são propagados “para a frente”, isto é, para o *receiver*.

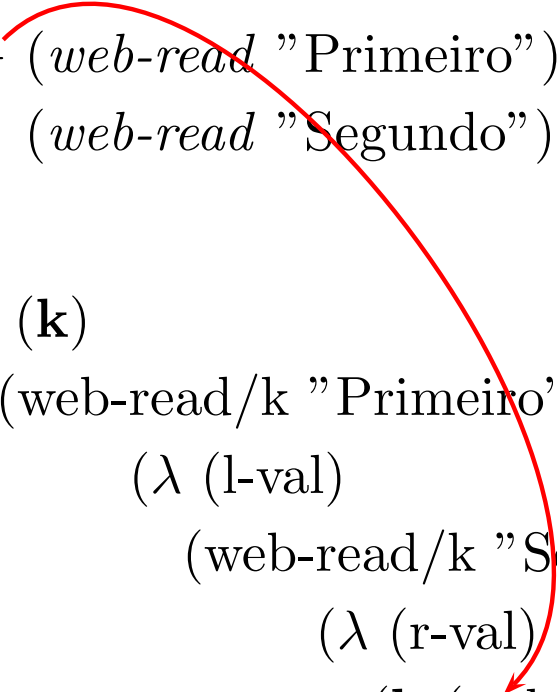
O *receiver* é uma representação procedural da pilha de execução. Programas escritos nesta forma são ditos estar no **estilo de passagem de continuação** (*Continuation Passing Style* — *CPS*).

Transformação informal

Toda expressão deve ter uma continuação.

$(+ \text{ (web-read "Primeiro")}$
 $\text{ (web-read "Segundo")})$

$(\lambda (\mathbf{k})$
 $\text{ (web-read/k "Primeiro"$
 $(\lambda (\text{l-val})$
 $\text{ (web-read/k "Segundo"$
 $(\lambda (\text{r-val})$
 $\text{ (}\mathbf{k} \text{ (+ l-val r-val))} \text{))))))$)



A chamada mais externa é passada para a identidade: $(\lambda (\mathbf{x}) \mathbf{x})$

Procedimentos

Cada expressão recebe uma continuação.

Mas, e a aplicação de um procedimento?

$$((proc\ arg)\ (func\ x))$$

Temos *proc* e *func*. As transformações ficam:

$$((proc/k\ arg\ (\lambda\ (f-val)\ \boxed{???})))$$

e

$$((func/k\ x\ (\lambda\ (a-val)\ \boxed{???})))$$

Mas o retorno também deve ser aplicado a uma continuação *depois* de calculado.

$((proc\ arg)\ (func\ x))$

$(\lambda\ (k)$
 $(proc/k\ arg$
 $(\lambda\ (f-val)$
 $(func/k\ x$
 $(\lambda\ (a-val)$
 $(k\ (f-val\ a-val))))))$

Constantes e variáveis

Constantes simples são passadas diretamente para a continuação:

$$42 \longrightarrow (\lambda (k) (k\ 42))$$

E expressões *lambda*?

$$(\lambda (x) x) \longrightarrow ???$$

$$(\lambda (k) (k\ (\lambda (x) x)))?????$$

O *lambda* interno não tem continuação!

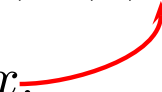
Ela é definida no tempo da **aplicação** e não da **definição**.

..continuação

A continuação em tempo de aplicação reflete a pilha de **execução**, que deve ser mantida. A associação *binding* não está no fechamento.

$$\begin{aligned} &(\lambda (k) \\ &\quad (k (\lambda (x \text{ dynk}) \\ &\quad\quad ((\lambda (k) (k x)) \text{ dynk})))) \end{aligned}$$

Este é o x .



O λ mais interno pode ser eliminado diretamente:

$$\begin{aligned} &(\lambda (k) \\ &\quad (k (\lambda (x \text{ dynk}) (\text{dynk } x)))) \end{aligned}$$

Transformação formal

É uma transformação de fonte em fonte (texto), na mesma linguagem, desde que ela tenha recursos adequados.

Em *Scheme* existe a definição de macros, mais ou menos como em *C*.

(**cps** *e*) passará *e* para o estilo com passagem de continuações.

Cabeçalho:

```
(define-syntax cps  
  (syntax-rules (+ lambda web-read))
```

Regras: soma

```
[(cps (+ e1 e2))  
  (lambda (k)  
    ((cps e1) (lambda (l-val)  
      ((cps e2) (lambda (r-val)  
        (k (+ l-val r-val)))))))]
```

Regras: aplicação de função

```
[(cps (f a))  
  (lambda (k)  
    ((cps f) (lambda (f-val)  
      ((cps a) (lambda (a-val)  
        (f-val a-val k)))))))]
```

Regras: definição de função

$$[(\mathbf{cps} \ (a) \ \mathit{corpo}) \\ \quad (\mathbf{lambda} \ (k) \\ \quad \quad (k \ (\mathbf{lambda} \ (a \ \mathit{dyn-k}) \\ \quad \quad \quad ((\mathbf{cps} \ \mathit{corpo}) \ \mathit{dyn-k})))))]$$

Para uma chamada com símbolo pré-definido é simples:

$$[(\mathbf{cps} \ (\mathbf{web-read} \ \mathit{prompt})) \\ \quad (\mathbf{lambda} \ (k) \\ \quad \quad (\mathit{web-read/k} \ \mathit{prompt} \ k))]$$

Contante:

$$[(\mathbf{cps} \ v) \ (\mathbf{lambda} \ (k) \ (k \ v))]$$

```

(define-syntax define-cps
  (syntax-rules ()
    [(define-cps (f arg) body)
     (define-cps f (lambda (arg) body))])
    [(define-cps v val)
     (define v ((cps val) (lambda (x) x))))])
(define-syntax cps
  (syntax-rules (+ lambda web-read)
    [(cps (+ e1 e2))
     (lambda (k)
       ((cps e1) (lambda (l-val)
                    ((cps e2) (lambda (r-val)
                                (k (+ l-val r-val))))))))])

    [(cps (lambda (a) body))
     (lambda (k)
       (k (lambda (a dyn-k)
            ((cps body) dyn-k))))])

    [(cps (web-read prompt))
     (lambda (k)

```



```

        (web-read/k prompt k))]
[(cps (f a))
 (lambda (k)
   ((cps f ) (lambda (f-val)
                ((cps a) (lambda (a-val)
                            (f-val a-val k)))))))]

[(cps v)
 (lambda (k) (k v)))]))
(define-syntax run
  (syntax-rules ()
    [(run e) ((cps e)
               (lambda (x)
                 (error "terminating with value" x))))])

```

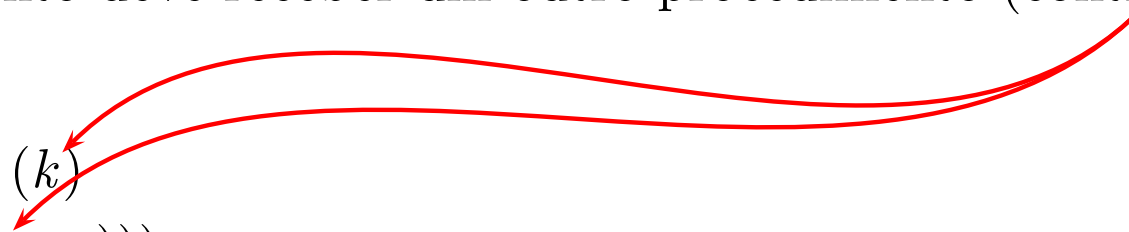
Programando com continuações _____

Scheme tem duas primitivas que permitem construir uma continuação: `call/cc` e `let/cc`.

`call/cc`, ou *call with current continuation*, recebe um procedimento e chama este procedimento com uma continuação.

O procedimento deve receber um outro procedimento (continuação) como argumento:

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



Quando o procedimento tiver um valor pronto, pode aplicar k a este valor. Emula um “return”.

let/cc

let/cc faz uma associação, ao invés de receber um procedimento. O argumento é um símbolo associado à continuação.

let/cc e call/cc são equivalentes.

```
(display (call/cc (lambda (x) (x (+ 2 3) ))))  
(display (let/cc x (x (+ 2 3))))
```

Exemplos

Qual o resultado?

- `(let/cc k (k 3))`
`k é (lambda (x) x) → 3`

- $$\bullet \left(+ 1 \boxed{(\text{let/cc } k \ (k \ 3))} \right) \\ k \ \acute{e} \ (\text{lambda } (\bullet) \ (+ \ 1 \ \bullet)) \rightarrow 4$$

Maaaaas, substituindo:

$(+ 1 ((+ (\text{land}(\text{the}(\text{cat}(\text{red}(\text{dog})) \text{dog})) \text{dog})) \text{dog}))$ Qual o problema?

Quando a continuação é aplicada, ela termina o processamento.

Portanto a segunda aplicação não acontece.

Notação *do livro*: λ^\uparrow

Corresponde a uma *closure* que termina o processamento assim que chamada (*escaper*).

Exemplo de uso: Tratamento de exceções! _____

Divisão por zero:

```
(define (f n) (+ 10 (* 5 (let/cc k (/ 1 n))))  
(+ 3 (f 0))
```

A continuação é

```
(lambda↑ (•)  
  (+ 3 (+ 10 (* 5 •))))
```

Para usar o tratamento, incluimos a verificação:

```
(let/cc esc  
  (/ 1 (if (zero? n) (esc 1) (n))))
```

Não deveríamos ter chamado a continuação para *n*???

Implementação (resumida) _____

Para simplificar, usaremos um meta-interpretador, com *closures* do Scheme.

O interpretador passa a receber uma continuação e a aplica a seu resultado.

Em alguns casos é trivial:

```
[num (n) (k (numV n))]
```

```
[id (v) (k (lookup v env))]
```

Chamando o próprio *interp* ---

Quando o interpretador for chamado recursivamente, devemos passar ou construir uma continuação:

```
[add (l r) (interp l env
            (lambda (lv)
              (interp r env
                    (lambda (rv)
                      (k (num+ lv rv)))))))]
```

Para os outros casos, é similar


Conditional

```
[if0 (test truth falsity)  
  (interp test env  
    (lambda (tv)  
      (if (num-zero? tv)  
        (interp truth env k)  
        (interp falsity env k)))))]
```


Funções

Definição:

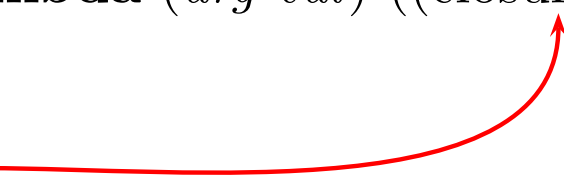
[fun (*param body*)
 (*k* (closureV (**lambda** (*arg-val dyn-k*)
 (*interp body* (aSub *param arg-val env*) *dyn-k*)))))]



Note a continuação dinâmica.

Aplicação:

[app (*fun-expr arg-expr*)
 (*interp fun-expr env* (**lambda** (*fun-val*)
 (*interp arg-expr env*
 (**lambda** (*arg-val*) ((closureV-p *fun-val*) *arg-val k*)))))))]



Representação procedural do *environment*.