Continuations, how to have made a different sandwich

Michal Atlas

January 31, 2023

What is a continuation?

A basic calculation

Expression cont.

```
(+ 2 (* 2 4))
```

```
(* 2 4) => (+ 2 _)
```

Statement cont.

```
(display 3)
(display (+ 2 2))
(display 5)
```

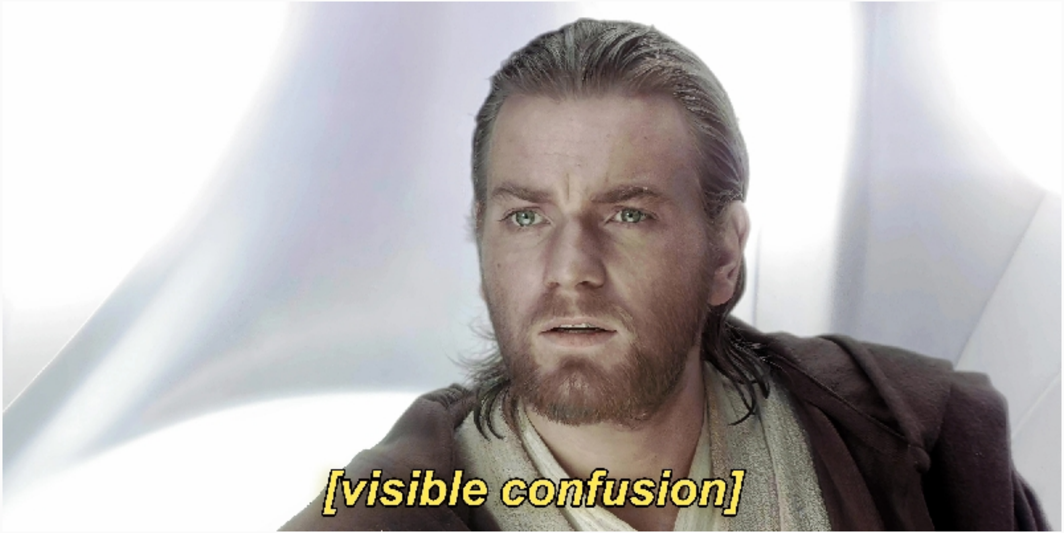
Wth?

```
(define k #f)
(define sandwich (call/cc (lambda (q) (set! k q) 'blt)))
sandwich
(k 'tomato)
sandwich
```

REPL ’blt & ’tomato

Program Loops indefinitely

What is going on?



Λ_{F}-Calculus[1]

$$\mathcal{F}_{\mathcal{R}} : V(\mathcal{F}M) \rightarrow \mathcal{F}(\lambda k.M(\lambda m.k(Vm)))$$

$$\mathcal{F}M \triangleright M(\lambda x.x)$$

[1] The Theory and Practice of First-Class Prompts - Matthias Felleisen

$\Lambda_{\mathbf{F}\#}$-Calculus

$$\mathcal{F}_{\mathcal{R}} : V(\mathcal{F}M) \to \mathcal{F}(\lambda k.M(\lambda m.k(Vm)))$$

$$\cancel{\mathcal{F}M \rhd M(\lambda x.x)}$$

$$\#(\mathcal{F}M) \to \#M(\lambda x.x)$$

Operators

call/cc[2]**(F)** undelimited[3] non-composable

shift/reset (F/#) delimited composable

control/prompt similar to shift/reset

[2] R^{5}RS and that's about it

[3] ignores prompts

Let’s see them in action

Early Return in functional languages

```
(if <some-cond>
    20
    (if <another-cond>
        3
        <actual-body>))
       ;^ can't put them in the middle here nicely
```

call/cc - Early Return in functional languages

```
(call/cc (lambda (return)
           (if <some-cond> (return 20))
           (if <some-cond> (return 3))

           <actual-body>
           ...
           (if <some-cond> (return 0))
           ...))
```

let/ec - So useful we have a macro

```
(let/ec return
  (if <some-cond> (return 20))
  (if <some-cond> (return 3))

  <actual-body>
  ...
  (if <some-cond> (return 0))
  ...)
```

escape continuation cheap but not general

call/cc - Cooler than it seems

```
(define (bar l)
  (let/ec return1
    ...
    (let/ec return2
      ...
      (foo return1 l))))

(define (foo k l)
  ...
  (k 20)
  ...)
```

shift/reset - Take[4]

```
(define (take n ls) (reset (%take n ls)))
(define (%take n ls)
  (if (zero? n)
      (shift k (cons (car ls) (k (cdr ls))))
      (cons (car ls) (%take (1- n) (cdr ls)))))

(take 2 '(1 2 3 4 5)) ;=> (3 1 2 4 5)
```

[4] Delimited Continuations for Everyone by Kenichi Asai

Comprehension check

```
(define (take n ls) (reset (%take n ls)))
(define (%take n ls)
  (if (zero? n)
      (call/cc (lambda (k) (cons (car ls) (k (cdr ls)))))
      (cons (car ls) (%take (1- n) (cdr ls)))))

(take 2 '(1 2 3 4 5)) ;=> ?
```

Comprehension check

```
(define (take n ls) (reset (%take n ls)))
(define (%take n ls)
  (if (zero? n)
      (call/cc (lambda (k) (cons (car ls) (k (cdr ls)))))
      (cons (car ls) (%take (1- n) (cdr ls)))))

(take 2 '(1 2 3 4 5)) ;=> (1 2 4 5)
```

Choose[5]

```
(define (choose . options) (shift k (map k options)))

(reset
 (let ([x (choose 1 2)]
       [y (choose 3 4 5)])
   (+ x (* 5 y))))

;=> ((16 21 26) (17 22 27))
```

[5] Quantum Continuations

Introducing some ambiguity

Amb - Backtracking[6]

```
(let ((a (amb 1 2 3 4 5 6 7))
      (b (amb 1 2 3 4 5 6 7))
      (c (amb 1 2 3 4 5 6 7)))

  ; We're looking for dimensions of a legal right
  ; triangle using the Pythagorean theorem:
  (assert (= (* c c) (+ (* a a) (* b b))))
  ; And, we want the second side to be the shorter one:
  (assert (< b a))

  (list a b c)) ;=> (4 3 5)
```

[6] Continuations by example: Exceptions, time-traveling search, generators, threads, and coroutines

Amb - Function itself

```
(define fail-stack '())

(define (amb . choices)
  (let ((cc (call/cc identity)))
    (cond
      [(null? choices) (fail)]
      [(pair? choices)
       (list-push! fail-stack cc)
       (list-pop! choices)])))
```

Amb - Assert & Fail

```
(define (assert condition)
  (if (not condition)
      (fail)
      #t))

(define (fail)
  (if (null? fail-stack)
      (error "back-tracking stack exhausted!")
      (begin
        (let ((back-track-point (list-pop! fail-stack)))
          (back-track-point back-track-point)))))
```

Amb - Why this works

```
(let ((a (amb 1 2 3 4 5 6 7))
      (b (amb 1 2 3 4 5 6 7))
      (c (amb 1 2 3 4 5 6 7)))

  ; We're looking for dimensions of a legal right
  ; triangle using the Pythagorean theorem:
  (assert (= (* c c) (+ (* a a) (* b b))))
  ; And, we want the second side to be the shorter one:
  (assert (< b a))

  (list a b c)) ;=> (4 3 5)
```

Exceptions

Exceptions

Exceptions

```scheme
(define (handler ctx)
  (format #t "Handled: ~a~%" ctx))

(+ 2 (shift k (handler 'context)))
; Prints "Handled: context"
```

Prompt/abort

```
(reset (shift k <handler-with-k-in-scope>)) =>

(% (... (abort ...) ...) <handler-with-k-as-arg>) =>

(with-exception-handler <thunk> <handler>)
```

Nicer syntax

```
(define (handler k ctx)
  (format #t "Handled: ~a~%" ctx))

(% (+ 2 (abort 'ctx)) handler)
;=> whatever the handler returns
```

Recovery

```
(define (handler k ctx)
  (format #t "Handled: ~a~%" ctx)
  (k 4))

(% (+ 2 (abort 'ctx)) handler) ;=> 6
```

Tags

```
(call-with-prompt <tag> <thunk> <handler>)

(abort-to-prompt <tag> <args> ...)
```

Coroutines

SRFI-121 - Generators

```scheme
(define (make-coroutine-generator proc)
  (define return #f)
  (define resume #f)
  (define yield
    (lambda (v) (call/cc (lambda (r) (set! resume r) (return v)))))
  (lambda () (call/cc (lambda (cc)
                        (set! return cc)
                        ...
                        (proc yield)))))
```

SRFI-121 that's it

```
(define (make-coroutine-generator proc)
  (define return #f)
  (define resume #f)
  (define yield (lambda (v) (call/cc (lambda (r) (set! resume r) (return v)))))
  (lambda () (call/cc (lambda (cc) (set! return cc)
                        (if resume
                          (resume (if #f #f))  ; void? or yield again?
                          (begin (proc yield)
                                 (set! resume (lambda (v) (return (eof-object))))
                                 (return (eof-object))))))))
```

Epilogue

For sanity and security

Dynamic Wind

```
(dynamic-wind
  <in-guard>
  <thunk>
  <out-guard>)
```

Continuation Barriers

```
(with-continuation-barrier
 <thunk>)
```

CPS - ~~Poor man's~~ Compiler's continuations

```scheme
(define (len k l) (if (null? l) 1 (k (sum 1+ (cdr l)))))
(len identity '(2 2 2 2 2)) ;=> 5

(define (/& x y ok err)
  (=& y 0.0 (lambda (b)
              (if b
                  (err (list "div by zero!" x y))
                  (ok (/ x y))))))
```

[7], [8]

[7] CPS Wiki

[8] Representing Control

Other uses

- Cooperative multitasking
- Webdev[9]
- Compiler optimizations[10, 11]
- Purely functional `set!`[12]
- Comefrom

[9] Web Applications in Racket

[10] Guile Reference Manual/CPS

[11] Chicken Scheme Wiki

[12] An Introduction to Algebraic Effects and Handlers

Other Languages I found implementations for:

- Haskell
- OCaml
- Racket
- Prolog[13]
- Standard ML
- C++[13] - Through Boost
- R[13]
- Unlambda
- Ruby

[13] Sorta

The Yin Yang Puzzle

Thank you

```
(let* ((yin
        ((lambda (cc) (display #\@) cc)
         (call-with-current-continuation (lambda (c) c))))
       (yang
        ((lambda (cc) (display #\*) cc)
         (call-with-current-continuation (lambda (c) c)))))
  (yin yang))
```

Appendix

```scheme
;; Imports shift/reset & %/abort in Guile
(use-modules (ice-9 control))

(define-syntax-rule (list-push! l v) (set! l (cons v l)))
(define-syntax-rule (list-pop! l) (let ([v (car l)]) (set! l (cdr l)) v))
```