Continuations

Daniil Berezun

danya.berezun@gmail.com

2022

Basic Expressions

```
(+ 2 3) ; 5
(- 2 3) ; -1
(* 1 2 3 4 5) ; 120
(+ (* 2 3) (/ 4 2)) ;8
```

Globals

```
(define a 2)
```

Local declarations

```
(let * [(a 1)
        (b 2)
        (c (+ a b))]
  (display "Hello")
  (set! b (+ b 1))
  (display b))
```

Conditions

```
(define a
  (if (zero? q)
      (+ 5 5)
      (* 12 12)))
(define (fun n)
  (cond
  [(zero? n)       1]
  [(equal? #f #t) 2]
  [else (* n q)]))
```

Functions

```
(define square
  (lambda (x) (* x x)))
(define (square x) (* x x))
(define (fact n)
  (if (< x 1)
  1
  (* x (fact (- x 1)))))
```

Basic Expressions

```
'(1 2 3 4) ; list
'()        ; empty list
#t         ; true
#f         ; false
'hello ; Symbol
       ;(immutable string)
"hello"; String
```

Continuations

Racket

```
(+ (* 3 4) 5)
(+ (* _ 4) 5)
(+ (* v 4) 5)
(Let ([v 3]) (+ (* v 4) 5))
((λ (v) (+ (* v 4) 5)) 3)
```

Tells you
"what to do next"

There might be some computations here!!!

Haskell

```
(+) ((*) 3 4) 5
(+) ((*) _ 4) 5
(+) ((*) v 4) 5
let v = 3 in (+) ((*) v 4) 5
(\ v -> (+) ((*) v 4) 5) 3
```

Continuations for sub-expressions of (+ (* 3 4) 5)

Racket		Haskell	
(+ (* 3 4) 5)	(lambda (v) v)	(+) ((*) 3 4) 5	\ v -> v
(* 3 4)	(lambda (v) (+ v 5))	(*) 3 4	\ v -> (+) v 5
3	(lambda (v) (+ (* v 4) 5))	3	\ v -> (+) ((*) v 4) 5
4	(lambda (v) (+ (* 3 v) 5))	4	\ v -> (+) ((*) 3 v) 5
5	(lambda (v) (+ (* 3 4) v))	5	\ v -> (+) ((*) 3 4) v

Consider again (+ (* 3 4) 5)

```
(+ (call/cc (lambda (k)
    (* 3 4)))
  5)
> 17
```

```
(+ (call/cc (lambda (k)
    (k (* 3 4))))
  5)
> 17
```

- call/cc is a procedure of one argument that takes a procedure of one argument (continuation k); ignores it in this case
- k itself represent a continuation to be ”call after”, i.e. (+ _ 5) in this case

But what is k exactly? Let’s do some more fun and check it:

```
(+ (call/cc
  (lambda (k)
    (begin
      (set! *k* k)
      (k (* 3 4)))))
  5)
> 17
> *k*
#<system continuation>
```

So *k* is our continuation!

(i.e. (lambda (v) (+ v 5))):

```
> (*k* (* 3 4))
17
> (*k* 12)
17
> (*k* 20)
25
```

k is exactly the ”rest of work to be done”

call/cc: “early exit”

Division Example

```
(call/cc
  (lambda (k)
    (/ 5 0 )))
>
Exception in /: undefined for 0
```

```
(call/cc
  (lambda (k)
    (/ 5 (k 0))))
> 0
```

Top-level continuation is identity!

Foo Example

```
(call/cc
  (lambda (k)
    (display "foo")))
> foo>   --- print!
```

```
(call/cc
  (lambda (k)
    (display (k "foo"))))
> "foo"  --- String
```

```
(call/cc
  (lambda (k)
    (error (k "foo"))))
> "foo"  --- String
```

Morale: call/cc allows us to implement “early exit” à la **break** in Java or **goto** in C
“early exit” *sim* this continuation never returns back

call/cc: “early exit” — step-by-step

```scheme
(let (
 (my-val (call/cc
  (lambda (the-continuation)
   (display "This will be executed\n")
   (the-continuation 5)
   (display
     "This will not be executed\n")))))
 (display my-val))

;; Output
; This will be executed
; 5
```

1. Saves the current stack into the-continuation
2. Reinstates the saved stack state
3. Return value 5 to the continuation’s calling context
4. The return value 5, goes to call/cc’s calling context, i.e. stores in my-val

(+ (* 3 4) 5) via call/cc

Term	Continuation	call/cc syntax
`(+ (* 3 4) 5)`	`(lambda (v) v)`	`(call/cc (lambda (k) (+ (* 3 4) 5)))`
`(* 3 4)`	`(lambda (v) (+ v 5))`	`(+ (call/cc (lambda (k) (k (* 3 4))))) 5)`
`3`	`(lambda (v) (+ (* v 4) 5))`	`(+ (* (call/cc (lambda (k) (k 3))) 4) 5)`
`4`	`(lambda (v) (+ (* 3 v) 5))`	`(+ (* 3 (call/cc (lambda (k) (k 4))))) 5)`
`5`	`(lambda (v) (+ (* 3 4) v))`	`(+ (* 3 4) (call/cc (lambda (k) (k 5))))`

call/cc: ignore function example

Ignore (forgetting) function

```
> (lambda (ignore) "hi")
#<procedure>
```

```
> ((lambda (ignore) "hi") 5)
"hi"
```

Let’s bind x to current continuation!

```
(let ([x (call/cc (lambda (k) k))])
  (x (lambda (ignore) "hi")))
> "hi" --- Why?
(let ([x  (lambda (v)
      (let ([x v])
      (x (lambda (ignore) "hi"))))
    ])
  (x (lambda (ignore) "hi")))
> "hi"
```

```
k: (lambda (v)
     (let ([x v])
     (x (lambda (ignore) "hi"))))
```

The same as:

```
(let ([x  (lambda (ignore) "hi"))])
  (x (lambda (ignore) "hi")))
> "hi"
```

or

```
((λ (ignore) "hi") (λ (ignore) "hi"))
```

NB: x is bound twice!

1. Since (**lambda** (k) k) returns its argument, x is bound to the continuation itself;
2. this continuation is applied to the procedure resulting from the evaluation of (**lambda** (ignore) "hi").
3. This has the effect of binding x (**again!**) to this procedure and applying the procedure to itself.
4. The procedure ignores its argument and returns "hi".

CPS by fact example

Naïve

```
(trace-define (fact n)
 (cond
 [(zero? n) 1]
 [else (* (fact (sub1 n))
          n)]))
> (fact 5)
>(fact 5)
> (fact 4)
> >(fact 3)
> > (fact 2)
> > >(fact 1)
> > > (fact 0)
< < < 1
< < <1
< < 2
< <6
< 24
<120
120
```

APS

```
(trace-define
 (fact-aps n acc)
 (cond
  [(zero? n) acc]
  [else (fact-aps (sub1 n)
                (* acc n))]))
(trace-define factA
  (λ (n) (fact-aps n 1)))
> (factA 5)
>(factA 5)
>(fact-aps 5 1)
>(fact-aps 4 5)
>(fact-aps 3 20)
>(fact-aps 2 60)
>(fact-aps 1 120)
>(fact-aps 0 120)
<120
120
```

CPS

```
(trace-define (fact-cps n k)
 (cond
  [(zero? n) (k 1)]
  [else (fact-cps (sub1 n)
    (λ (v) (k (* v n))))]))
> (factCPS 5)
>(factCPS 5)
>(fact-cps 5 #<procedure:...>)
>(fact-cps 4 #<procedure:...>)
>(fact-cps 3 #<procedure:...>)
>(fact-cps 2 #<procedure:...>)
>(fact-cps 1 #<procedure:...>)
>(fact-cps 0 #<procedure:...>)
<120
120
```

aps vs cps

- ✗ aps stores a number; cps stores the whole procedure: **huge** amount of heap **space**!
- ✓ cps transformation can be done **automatically** with **ANY** function making it tail-recursive!

CPS by fact example

cps fact again

```
(trace-define (fact-cps n k)
  (cond
   [(zero? n) (k 1)]
   [else (fact-cps (sub1 n)
     (λ (v) (k (* v n))))]))
> (factCPS 5)
>(factCPS 5)
>(fact-cps 5 #<procedure:...>)
>(fact-cps 4 #<procedure:...>)
>(fact-cps 3 #<procedure:...>)
>(fact-cps 2 #<procedure:...>)
>(fact-cps 1 #<procedure:...>)
>(fact-cps 0 #<procedure:...>)
<120
120
```

What if continuation is not identity

```
(trace-define factCPS2
   (λ (n) (fact-cps n (λ (x) (* x 3)))))
> (factCPS2 5)
???360
> (fact-cps 5 (λ (x) (+ (* x 4) 5)))
???485
```

Exercises:

- One can try to do that with Fibonacci to see the beauty
- Compare memory usage with aps, cps and usual fib by running (fib -1) which goes to infinite loop

What is the type of `call/cc`?

```
(+ 3 4)
(+ 3 (call/cc (λ (k) 4))) ; same with call/cc
; call/cc : (? -> ?) -> ? ; call/cc is a function that taks a function
; call/cc : (? -> ?) -> Number ; in (+ 3 _) "_" obviously should be a Number

(+ 3 (call/cc (λ (k) (k 4))))  ; inner function takes a continuation
; call/cc : ((? -> ?) -> ?) -> Number ; , i.e. another 1-arg function
; call/cc : ((Number -> ?) -> ?) -> Number ; k ≡ (define (k x) (+ 3 x))

(+ 3 (call/cc (λ (k) (k 4) 5)))
; call/cc : ((Number -> ?) -> Number) -> Number ; λ can return a Number

(+ 3 (call/cc (λ (k) (zero? (k 4)) 5))) ; k can return anything
; call/cc : ((Number -> Boolean) -> Number) -> Number
(+ 3 (call/cc (λ (k) ((string-length (k 4)) 5)))
; call/cc : ((Number -> String) -> Number) -> Number
; call/cc : ((Number -> β) -> Number) -> Number

(string-append "Hello " (call/cc (λ (k) (string-length (k "World")) "NOT")))
; call/cc : ((α -> β) -> α) -> α
```

What is $((\alpha \to \beta) \to \alpha) \to \alpha$? **Pierce law**!!

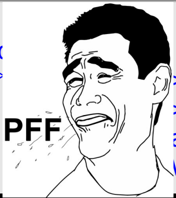
Intuitionistic logic + Pierce law $\Rightarrow$ Classical logic!

But Racket type system is not by Hindley–Milner, it is different!

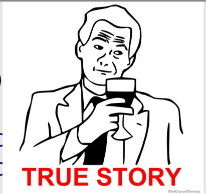
```
> (if #t 23 "Hello") ; is ok
23
> (if #f 23 "Hello") ; is ok
"Hello"

; call/cc : ((α -> β) -> α) -> α

(string-append "Hello " (call/cc (λ (k) (zero? (k "World")) #f))) ; ok
(string-append "Hello " (call/cc (λ (k) #f)))                     ; ok
(string-append "Hello " (call/cc (λ (k) (k "World"))))            ; ok

; call/cc : ((α -> β) -> γ) -> α ∪ γ ; something like that
```

Actually . . . in Typed Racket (statically typed): type of call/cc is

```
[call/cc
 (-polyd
   (cl->
                > (Un)) (-values            ues null)
                >... (list a)            ))
                ake-Va            ist (-result b)) c 'c
                Values            list (-result (Un a b))) c
```



As easy as pie!!!



What else can we do with continuations?

Via call/cc can be expressed

- Early exit (break)
- goto
- Conditionals
- Exceptions (Try/Catch/Throw)
- Cooperative Multithreading
- ...Actually... **ANY CONTROL** flow operators and manipulations

Cost

- ✗ Slow
- ✗ Huge heap usage
- ✗ Too complicated

Too complicated ...

What does this code?

```
(let* ((yin ((λ (foo) (newline) foo)
             (call/cc (λ (bar) bar))))
       (yang ((λ (foo) (write-char #\*) foo)
              (call/cc (λ (bar) bar)))))
  (yin yang))
```

Output:

```
*
**
***
****
*****
******
*******
...
```

Continuations and callCC in Haskell

Please take a look at these sources:

- wikibooks: Monad transformers
- The Cont monad
- The ContT monad transformer
- Nice into; the following slides are borrowed from it

Continuations

```haskell
ret value = \ f -> f value
twoC    = ret 2
helloC = ret "Hello"
```

Chaining Continuations

```haskell
-- takes a continuation and a function which is provided the value of it
-- and returns a new continuation as a result
inC `bind` f = \out -> inC (\inCVal -> (f inCVal) out)
-- double the value
fourC = twoC `bind` \two -> ret ((*) two 2)
-- fourC id == 4
-- glueing two continuations:
twoHelloC = twoC `bind` \two ->
              helloC `bind` \hello ->
                ret $ (show two) ++ hello
-- twoHelloC id == "2hello"
```

The Cont Monad

Actually, monad transformers but for the sake of briefly:

```haskell
newtype Cont r a = Cont { runCont :: (a -> r) -> r }

instance Functor (Cont r) where
    fmap f (Cont c) = Cont $ \out -> c (out . f)

instance Applicative (Cont r) where
    pure val = Cont $ \out -> out val
    (Cont f) <*> (Cont c) = Cont $ \out -> f $ \fn -> c (out . fn)

instance Monad (Cont r) where
    return = pure
    -- like your `bind` but wrapping/unwrapping Cont:
    (Cont c) >>= f = Cont $ \out -> c (\a -> (runCont (f a)) out)

-- callCC
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC fn = Cont $ \out -> runCont (fn (\a -> Cont $ \_ -> out a)) out

-- in the transformer version it hides away the identity monad
cont :: ((a -> r) -> r) -> Cont r a
cont = Cont
```

The Cont Monad: Example — 1

```
twoC'    = return 2
helloC' = return "hello"

twoHelloC' = do
    two <- twoC'
    hello <- helloC'
    return $ (show two)++hello

twoHelloC'' = twoC' >>= \two ->
                helloC' >>= \hello ->
                  return $ (show two)++hello

> (runCont twoHelloC') id -- "2hello"
```

Branching and Generation

```
boom1C = do
    n <- cont $ \out -> "boom! "
    l <- cont $ \out -> out "a" ++ out "b"
    x <- cont $ \out -> out "X" ++ out "Y"
    return $ n ++ l ++ x ++ " "
> runCont boom1C id -- "boom! "

boom2C = do
    n <- cont $ \out -> out "1" ++ out "2"
    l <- cont $ \out -> "boom! "
    x <- cont $ \out -> out "X" ++ out "Y"
    return $ n ++ l ++ x ++ " "
> runCont boom2C id -- "boom! boom! "

boom3C = do
    n <- cont $ \out -> out "1" ++ out "2"
    l <- cont $ \out -> out "a" ++ out "b"
    x <- cont $ \out -> "boom! "
    return $ n ++ l ++ x ++ " "
> boom3C id -- "boom! boom! boom! boom! "
```

callCC example

```haskell
import Control.Monad.Trans.Class
import Control.Monad.Trans.Cont

main = flip runContT return $ do
    lift $ putStrLn "alpha"
    (k, num) <- callCC $ \k -> let f x = k (f, x)
                               in return (f, 0)
    lift $ putStrLn "beta"
    if num < 5
        then k (num + 1) >> return ()
        else lift $ print num
```

```
ghci> main
alpha
beta
beta
beta
beta
beta
beta
5
```

Goto

```haskell
{-# LANGUAGE ScopedTypeVariables #-}

import qualified Control.Monad.Trans.Cont  as C
import           Control.Monad.Trans.Class (lift)
import           System.Random             as R

--simple goto
goto = C.callCC $ \out -> let fn = out fn
                          in return fn

-- we either go back to 1, 2, or finish
gotoEx = flip C.runContT return $ do

    marker1 <- goto
    lift $ putStrLn "one"

    marker2 <- goto
    lift $ putStrLn "two"

    (num :: Int) <- lift $ R.randomRIO (0,2)

    if      num < 1 then marker1
    else if num < 2 then marker2
    else lift $ putStrLn "done"
```

```
ghci> gotoEx
one
two
two
done
ghci> gotoEx
one
two
done
ghci> gotoEx
one
two
one
two
one
two
one
two
one
two
two
done
ghci>
```