**16. Continuation**

**1. Introduction**

This chapter is about continuation which is a characteristic data type of Scheme. As other programming languages does not have this data type, it is difficult to understand it. You don't need fully understanding for now, just getting the feel of it.

I will explain continuation in general and continuation passing style (CPS in short), then explain the continuation of Scheme. I think this way is easier to understand the continuation.

**2. Continuation in General**

Continuation is the calculation what should be performed before returning to the toplevel. Actually, **the continuation exists everywhere during computation**. For instance, in the case of (\* 3 (+ 1 2)),  
the calculation that should be done is multiplying 3 { (\* 3 []) }  
after evaluating (+ 1 2). As most of languages do not treat it explicitly, however, it is unfamiliar ot many programmers.

**3. Continuation Passing Style (CPS)**

**3.1. Simple CPS**

CPS is a programming style in which the successive function(s) that uses the result of current function is given as an argument of the current function. [code 1] shows adding and multiplying written in CPS. In k+ and k\*, **k** is the successive function.

[code 1]

(define (return x)

x)

(define (k+ a b k)

(k (+ a b)))

(define (k\* a b k)

(k (\* a b)))

[example 1] shows how to calculate (\* 3 (+ 1 2)) using the CPS.

[example 1]

(k+ 1 2 (lambda (x) (k\* x 3 return)))

In the ordinary form of Scheme, values that are calculated in parentheses go outside of them. In the CPS, on the contrary, values go inside of other parentheses. In [example 1], k+ passes the value of (+ 1 2) to (lambda (x) (k\* x 3 return)) and **k\*** passes the result of (\* (+ 1 2) 3) to return.

**3.2. Recursive Functions written in CPS**

Recursive functions can be written in CPS as well. [code 2] shows functions that calculate factorial written in the normal style (fact) and CPS (kfact).

[code 2]

;;; normal factorial

(define (fact n)

(if (= n 1)

1

(\* n (fact (- n 1)))))

;;; CPS factorial

(define (kfact n k)

(if (= n 1)

(k 1)

(kfact (- n 1) (lambda (x) (k (\* n x))))))

[example 2] adds 3 to the factorial of 4.

[example 2]

;;; normal

(+ 3 (fact 4))

;;; CPS

(kfact 4 (lambda (x) (k+ x 3 return)))

[code 3] shows functions to calculate the product of list items written in the ordinaly way and CPS. In the CPS function, the next function is stored in a local variable **break**, so that it can quit immediately when 0 is multiplied.

[code 3]

;;; normal

(define (product ls)

(let loop ((ls ls) (acc 1))

(cond

((null? ls) acc)

((zero? (car ls)) 0)

(else (loop (cdr ls) (\* (car ls) acc))))))

;;; CPS

(define (kproduct ls k)

(let ((break k))

(let loop ((ls ls) (k k))

(cond

((null? ls) (k 1))

((zero? (car ls)) (break 0))

(else (loop (cdr ls) (lambda (x) (k (\* (car ls) x)))))))))

[example 3] shows adding 100 to the product of '(2 4 7).

[example 3]

;;; normal

(+ 100 (product '(2 4 7)))

;;; CPS

(kproduct '(2 4 7) (lambda (x) (k+ x 100 return)))

Even CPS is not so profitable in such simple cases, it is useful to write complicated programs such as natural language parsing and logical programming, because the CPS can change the successive process more flexible than ordinary programming style.

Exception handling is a simple example of such cases. [code 4] shows error handling version of kproduct, in that a non-number value is shown and the calculation is terminated when it appears in the input list.

[code 4]

(define (non-number-value-error x)

(display "Value error: ")

(display x)

(display " is not number.")

(newline)

'error)

(define (kproduct ls k k-value-error)

(let ((break k))

(let loop ((ls ls) (k k))

(cond

((null? ls) (k 1))

((not (number? (car ls))) (k-value-error (car ls)))

((zero? (car ls)) (break 0))

(else (loop (cdr ls) (lambda (x) (k (\* (car ls) x)))))))))

[example 4]

;;; valid

(kproduct '(2 4 7)

(lambda (x) (k+ x 100 return))

non-number-value-error)

;Value: 156

;;; invalid

(kproduct '(2 4 7 hoge)

(lambda (x) (k+ x 100 return))

non-number-value-error)

Value error: hoge is not number.

;Value: error

**4. Continuation of Scheme**

You may grasp what is continuation by the above explanation. Continuation

1. exists whole computional processes, and
2. it can be treated explicitly by a functional programming language and CPS.

In addition, examples above show that it is a **chain of closure**.

However, reading and writing programs using CPS is painful and it will be convenient to handle the continuation with conventional programming style.

For this reason, the continuation is impremented as a first class object (which means ordinal data type) in Scheme and is invoked by a function named **call-with-current-continuation** at any timing. As the continuation is an ordinaly data type, it can be re-used as mach as you want.

As the call-with-current-continuation is a long name, an abbreviated name call/cc is often used.

(define call/cc call-with-current-continuation)

The call-with-current-continuation (call/cc) takes one argument. The argument is a function whose argument is the current continuation.   
Followings show examples:

(\* 3 (call/cc (lambda (k) (+ 1 2)))) ⇒ 9 ; [1]

(\* 3 (call/cc (lambda (k) (+ 1 (k 2))))) ⇒ 6 ; [2]

In the case [1], the continuation is not invoked and it behaves same as an ordinal S-expression does. On the other hand, the continuation is invoked in the case [2] and 2 is given as a parameter of the continuation. In such cases, the parameter of the continuation skips the process in call/cc and goes outside of the call/cc. In this case, k is a function with one parameter and equal to   
(lambda (x) (\* 3 x))   
In general, current continuation stores the process to return to the toplevel from the point that call/cc is invoked.

Current continuations can be saved like as other data type and re-used as much as you want.

(define cc)

(\* 3 (call/cc (lambda (k)

(set! cc k)

(+ 1 2))))

As the current continuation is the process to come back to the toplevel, it returns to the toplevel with ignoring surrounding S-expressions.

(+ 100 (cc 3)) ⇒ 9

(+ 100 (cc 10)) ⇒ 30

**4.1. Throwing values using call/cc**

The easiest way of using the current continuation is escape from a process. [code 5] shows a function that search trees (nested lists). If the function finds **obj** in a tree, it returns it otherwise #f. When the **obj** is found the function throws it directly to the outermost.

[code 5]

(define (find-leaf obj tree)

(call/cc

(lambda (cc)

(letrec ((iter

(lambda (tree)

(cond

((null? tree) #f)

((pair? tree)

(iter (car tree))

(iter (cdr tree)))

(else

(if (eqv? obj tree)

(cc obj)))))))

(iter tree)))))

[example 5]

(find-leaf 7 '(1 (2 3) 4 (5 (6 7))))

⇒ 7

(find-leaf 8 '(1 (2 3) 4 (5 (6 7))))

⇒ ()

[code 6] shows a code for block that support throw.

[code 6]

(define-syntax block

(syntax-rules ()

((\_ tag e1 ...)

(call-with-current-continuation

(lambda (tag)

e1 ...)))))

[expample 7] shows how to use it.

[expample 7]

(block break

(map (lambda (x)

(if (positive? x)

(sqrt x)

(break x)))

'(1 2 3)))

⇒ (1 1.4142135623730951 1.7320508075688772)

(block break

(map (lambda (x)

(if (positive? x)

(sqrt x)

(break x)))

'(1 -2 3)))

⇒ -2

**4.2. generator**

I will explain a tree matching generator which uses call/cc. The generator takes a tree as an argument and returns a function which returns the successive leaf every time when it is invoked. You can find the original version of this function at [Section 13.3 of Teach Yourself Scheme in Fixnum Days](http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-15.html#node_sec_13.3). The generator is used like as follows:

(define tr '((1 2) (3 (4 5))))

(define p (leaf-generator tr))

(p) => 1

(p) => 2

(p) => 3

(p) => 4

(p) => 5

(p) => () ; finally it returns '()

The definition of the generator is shown in [code 6]. This is basically the same as the original version but modified slightly.

[code 6]

01: (define (leaf-generator tree)

02: (let ((return '())) ; 1

03: (letrec ((continue ; 2

04: (lambda ()

05: (let loop ((tree tree)) ; 3

06: (cond ; 4

07: ((null? tree) 'skip) ; 5

08: ((pair? tree) (loop (car tree)) (loop (cdr tree))) ; 6

09: (else ; 7

10: (call/cc (lambda (lap-to-go) ; 8

11: (set! continue (lambda () (lap-to-go 'restart))) ; 9

12: (return tree)))))) ;10

13: (return '())))) ;11

14: (lambda () ;12

15: (call/cc (lambda (where-to-go) ;13

16: (set! return where-to-go) ;14

17: (continue)))))))

Comments:

| **No.** | **Explanation** |
| --- | --- |
| 1. | declaring a local variable return. |
| 2. | defining continue using letrec. The continue returns current leaf in front, assigns the current continuation to the next continue, and halts. |
| 3. | defining rec using named let. |
| 4. | branching using cond |
| 5. | if empty list, does nothing. |
| 6. | if pair, applies the rec recursively to its car and cdr. |
| 7. | if leaf, |
| 8. | invokes call/cc to get the current state (lap-to-go) |
| 9. | and set it to the next continue. The lap-to-go includes the current state in addition to the original continue. In short, it can be expressed by [ ]in the following S-expression.  (lambda ()  (let rec ((tree tree0))  (cond  ((null? tree) '())  ((pair? tree) (rec (car tree)) (rec (cdr tree)))  (else  [ ]  (return '()))))  As invoking lap-to-go means that (car tree) is a leaf and the process is finished, (rec (cdr tree)) starts at the next function is called. As the process starts after finishing the part of [ ], The argument of the continuation does not matter. |
| 10. | Then the function return the found leaf to where the function is called. (return tree) should be inside of call/cc to restart the process. |
| 11. | Returning an empty list after searching all leaves |
| 12. | It is a generator that leaf-generator returns. |
| 13. | First invoking call/cc |
| 14. | and assign the plase to return values to return. |
| 15. | Then calls continue. |

The behaviour of the function generated by the leaf-generator can be estimated by the behavior of a conventional traversing function (tree-traverse). The process halts at '\*' of the trace and remained process is stored in the continue.  
A conventional traverse functionF

(define tree-traverse

(lambda (tree)

(cond

((null? tree) '\_)

((pair? tree) (tree-traverse (car tree)) (tree-traverse (cdr tree)))

(else

(write tree)))))

Trace of the tree-traverse when tree '((1 2) 3) is given.

> (tree-traverse '((1 2) 3))

|(tree-traverse ((1 2) 3))

| (tree-traverse (1 2))

| |(tree-traverse 1)

1| |#< void> ; \*

| (tree-traverse (2))

| |(tree-traverse 2)

2| |< void> ; \*

| (tree-traverse '())

| \_

|(tree-traverse (3))

| (tree-traverse 3)

3| #< void> ; \*

|(tree-traverse '())

|\_

\_

**4.3. Coroutine**

As continuation remember the successive process, [coroutines](http://en.wikipedia.org/wiki/Coroutines), in which several tasks are executed simultaneously, can be impremented using the continuation.

[code 7] shows a program that print numbers and alphabetic characters alternately. Lines 5 — 22 are imprementation of queue. (enqueue! queue obj) adds **obj** at the end of **queue**. (dequeue! queue) returns the first item of the queue with removing it.

Lines 26 — 38 are imprementation of a coroutine.

**process-queue**

The queue of processes.

**(coroutine thunk)**

adding thunk at the end of the process-queue.

**(start)**

picking up the first process of the process-queue and executing it.

**(pause)**

adding the current continuation at the end of the process-queue and execute the first process of the queue. This function hand over the control to the other routine.

Lines 42 — 61 show how to use it. The routine showing numbers and that showing alphabetic characters call each other and a result shown in [example 7] is obtained.

[code 7]

01: ;;; abbreviation

02: (define call/cc call-with-current-continuation)

03:

04: ;;; queue

05: (define (make-queue)

06: (cons '() '()))

07:

08: (define (enqueue! queue obj)

09: (let ((lobj (list obj)))

10: (if (null? (car queue))

11: (begin

12: (set-car! queue lobj)

13: (set-cdr! queue lobj))

14: (begin

15: (set-cdr! (cdr queue) lobj)

16: (set-cdr! queue lobj)))

17: (car queue)))

18:

19: (define (dequeue! queue)

20: (let ((obj (car (car queue))))

21: (set-car! queue (cdr (car queue)))

22: obj))

23:

24:

25: ;;; coroutine

26: (define process-queue (make-queue))

27:

28: (define (coroutine thunk)

29: (enqueue! process-queue thunk))

30:

31: (define (start)

32: ((dequeue! process-queue)))

33:

34: (define (pause)

35: (call/cc

36: (lambda (k)

37: (coroutine (lambda () (k #f)))

38: (start))))

39:

40:

41: ;;; example

42: (coroutine (lambda ()

43: (let loop ((i 0))

44: (if (< i 10)

45: (begin

46: (display (1+ i))

47: (display " ")

48: (pause)

49: (loop (1+ i)))))))

50:

51: (coroutine (lambda ()

52: (let loop ((i 0))

53: (if (< i 10)

54: (begin

55: (display (integer->char (+ i 97)))

56: (display " ")

57: (pause)

58: (loop (1+ i)))))))

59:

60: (newline)

61: (start)

[example 7]

(load "cor2.scm")

;Loading "cor2.scm"

1 a 2 b 3 c 4 d 5 e 6 f 7 g 8 h 9 i 10 j -- done

;Unspecified return value

**5. Summary**

In this chapter, I explained about continuation.

It may be difficult to understand the idea. But don't worry. You will get it some day.

I will explain about lazy evaluation in the next chapter.