all about delimited continuations revised1

by /zt'2016/9/2

我来写一篇分享使用 continuation 的经验文章，对 continuation 进行更加深入的挖掘。
由于本人水平不算太高，仅仅希望分享一下经验，如有错误，还请指出。

1.INTRODUCTION TO SHIFT/RESET

有人一谈到 continuation，就会想到 callcc，其实 continuation 可不只 callcc 这一个。
callcc 属于 full continuation，而其他的一般属于 delimited continuation，那么有什么区别呢？

我们先来介绍一个 scheme 里面的 delimited continuation，shift/reset
在 racket 里面加入：

(require racket/control)

测试下面的代码
(reset 1)
(reset (+ 1 2))
结果分别是 1 和 3，因此当只有 reset 的时候，**(reset E)=>E**

shift 的形式：**(shift k E)**,而且 shift 只能在 reset 内使用
(reset (+ (shift k 2) 3)) =>2
因此我们得到 **(reset (... (shift k E) ...)) => E**

当然我们也可以用 callcc 来实现同样的功能：

```
(call/cc (lambda (exit)
           (+ (exit 2) 3)))
```

继续来看 shift，还有一个问题没有解决，那就是，k 是什么？
(reset (+ (shift k (k 1)) 2))
结果为 3，根据我们上面的原理，结果应该为(k 1),那么 k 是什么呢
k = (+ [] 2) ,k 就是这个 continuation

好了理解了这个，我们把上面的程序变一下
(reset (+ (shift k (k (k 1))) 2))
返回 **(k (k 1)) where k = (+ [] 2)**
所以应该返回 5

再变一下
(reset (+ (shift k k) 2))
这回是直接把这个 continuation 返回了，所以就等价于**(lambda (v) (+ v 2))**
((reset (+ (shift k k) 2)) 1) => 3

(reset (+ (shift k k) (shift k k)))
这么复杂，应该如何理解呢
为了便于理解，我变换一下程序
(reset (+ (shift k1 k1) (shift k2 k2)))
那么首先返回的肯定是这个 k1 了，**k1 = (+ [] (shift k2 k2))**,对吗
有一个问题，shift 必须要在 reset 里面使用，而这个 shift 外面没有 reset 啊
所以 **k1= (reset (+ [] (shift k2 k2)))**
现在我们传一个值给 k1，比如说 1 吧
这样就成了 **(reset (+ 1 (shift k2 k2)))** 我们会得到 **k2 =(reset (+ 1 []))**
再给 k2 传一个值，比如 2，就会得到最终结果 3

所以 **(reset (+ (shift k k) (shift k k)))** 等价于 **(lambda (v1) (lambda (v2) (+ v1 v2)))**

我们可以得到 shift reset 的精确定义(E 是不含 reset 和 shift 的表达式）
(reset E) => E
(reset (... (shift k E) ...) => E [k = (reset (... [] ...))]

2.FULL CONTINUATION VS DELIMITED CONTINUATION
通过 shift 获取的所有 continuation 都会被严格限定在 reset 范围之内。
这是什么意思呢

```
(define c (void))
(+ 10 (+ 2 (call/cc (lambda (k)
                (set! c k) 1))))
(c 2)
```

这里的 callcc 捕获的 continuation 是**(+ 10 (+ 2 []))** 因此**(c 2)**会返回 14

```
(require racket/control)
(define c (void))
(+ 10 (+ 2 (reset (shift k
                (set! c k) 1))))
(c 2)
```

但是这里的 shift 只捕获了 reset 内的 continuation，**c=(lambda (v) v)**

```
(require racket/control)
(define c (void))
(+ 10 (reset (+ 2 (shift k
                (set! c k) 1))))
(c 2)
```

这里的 shift 只捕获了一部分的 continuation，**c=(+ 2 [])**

```
(require racket/control)
(define c (void))
(reset (+ 10 (+ 2 (shift k
                (set! c k) 1))))
```

(c 2)
把 reset 放到最顶层，捕获整个表达式的 continuation，**c=(+ 10 (+ 2 []))**
所以 callcc 只能捕获整个 continuation，而 reset shift 可以限定范围

3.YIELD RETURN --CALLCC 带来的问题
我相信基本上每个人都用 callcc 实现过 yield return（generator）
这是我从网上找到的一个用 callcc 实现的代码

```
(require racket/control)
(define (generate-one-element-at-a-time lst)

(define (control-state return)
      (for-each (lambda (element)
                        (call/cc (lambda (resume-here)
                                          (set! control-state resume-here)
                                          (return element))))
                      lst)
(return 'end))

(define (generator)       (call/cc control-state))

   generator)
(define generate-digit    (generate-one-element-at-a-time '(0 1 2)))

(generate-digit)
(generate-digit)
(generate-digit)
```

这么简单，不过就是每次返回的时候记下 continuation，然后下次在从 continuation 处继续执行吧。
但是，这样实现的 yield return，存在致命的缺陷。

测试一下下面的代码

```
`(,(generate-digit))
(generate-digit)
(generate-digit)
```

返回值令我们大吃一惊

```
'(0)              right
'(1)              expected to be 1
'(2)              expected to be 2
```

很明显，最后两个应该是 1 和 2 啊

在试试 **(cons (generate-digit) (generate-digit))** 你会发现并不会返回'(0 . 1),而是陷入了死循环。

这个问题可以被修复，但是比较麻烦，根源就在于 callcc 把会捕获整个 continuation(见上文)，而 shift reset 比较守规矩，reset 外的 continuation 是不会被捕获的。

为了解决问题，我们来用 shift 和 reset 写一个 generator 吧。

```
(require racket/control)
(define (make-generator lst)
  (define (control)
    (reset (for-each (lambda (x)
                       (shift k (set! control (lambda () (k (void))))
                         x)) lst)))
  (lambda () (control)))
(define a (make-generator `(0 1 2)))
`(,(a))
(a)
(a)
```

相比刚才的代码更加短小，而且没有 bug 了哦
所以 shift reset 的实际用处其实比 callcc 多哦。

4.有关 shift reset 的算法
cps 变换是一种经典的算法，在编译领域十分有用，详见 andrew appel 的大作 compiling with continuations,还可以用来实现 callcc。
通常的 cps 算法都是把函数写成 cps 的样式，不过我们可以直接同过 shift 和 reset 进行 cps 变换。

```
#lang racket
(require racket/control)
(define (cps exp)
  (match exp
    [(? symbol? x) x]
    [`(lambda (,x) ,expr) `(lambda (,x k)
                             ,(reset `(k ,(cps expr))))]
    [`(,rator ,rand) (shift k
                       `(,(cps rator) ,(cps rand) (lambda (value) ,(k `value))))]))

(reset (cps '(lambda (x) x)))
(reset (cps '(lambda (x) (lambda (y) (y x)))))
(reset (cps '((lambda (x) x) (lambda (y) y))))
```

```
Welcome to DrRacket, version 6.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.
'(lambda (x k) (k x))
'(lambda (x k) (k (lambda (y k) (y x (lambda (value) (k value))))))
'((lambda (x k) (k x)) (lambda (y k) (k y)) (lambda (value) value))
>
```

```
(require racket/control)
(define (cps exp)
  (match exp
    [(? symbol? x) x]
    [`(lambda (,x) ,expr) `(lambda (,x k)
                             ,(reset `(k ,(cps expr))))]
    [`(,rator ,rand) (shift k
                       `(,(cps rator) ,(cps rand) (lambda (value) ,(k `value))))]))

(reset (cps '(lambda (x) x)))
(reset (cps '(lambda (x) (lambda (y) (y x)))))
(reset (cps '((lambda (x) x) (lambda (y) y))))
```

为了更容易看懂，这是一个非常简单的 cps 变换

最后，由于水平和篇幅有限，不可能把所有关于 delimited continuation 的内容都列举。

参考文献

1.https://en.wikipedia.org/wiki/Delimited_continuation
2.http://community.schemewiki.org/?composable-continuations-tutorial
3.oleg:Introduction to programming with shift and reset [http://okmij.org/ftp/continuations/]
4.callcc 实现 generator 的代码来源于 http://blog.sina.com.cn/s/blog_4dff871201018wtz.html