Continuations & Continuation Passing Style

Ramakrishnan Muthukrishnan
Functional Conference 2014, Bangalore
@vu3rdd

Outline of the talk

- The idea of continuations.
- A few examples.
- call-with-current-continuations or call/cc.
- continuation passing style and Tail Call Optimisation.
- Write an interpreter for a toy Scheme in CPS and implement call/cc in that interpreter.
- Q & A

A little bit about me

- Electrical Engineer.
- Self taught programmer.
- Working for Concur Technologies.
- Free Software (GNU project, Debian)

Scheme

- call-by-value.
- S-expressions.
- lexical scoping.
- very few builtin constructs.
- functions as first class.
- macros.
- call/cc.

"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."

Continuations

Let us say that we want to add two numbers taken from the user input:

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

(define (read-number [prompt : string]) : number
  (begin
    (display prompt)
    (let ([v (read)])
      (if (s-exp-number? v)
        (s-exp->number v)
   (read-number prompt)))))
```

What remains to be done after the first number is read?

```
(lambda (v1)
   (+ v1
      (read-number “Second”)))
```

Continuations..

Let us take this expression:

```
(+ (* 3 4) 5)
```

```
(+ (* ? 4) 5)
```

```
(+ (* v 4) 5)
```

```
(let ([v 3])
  (+ (* v 4) 5))
```

Now, **`let`** is just a syntactic sugar on top of **`lambda`**. So, we can write:

```
(let ([v 3])
  (+ (* v 4) 5))

=>  ((lambda (v) (+ (* v 4) 5)) 3)
```

Let us look only at the procedure:

```
(lambda (v) (+ (* v 4) 5))
```

This procedure is the representation of the statement: “Given the value, v, what do we need to do, in order to evaluate the expression `(+ (* v 4) 5)`.

“The term Continuation implies the rest of work that has to happen to finish up the evaluation of the program.”

We represent the continuation as a function that takes a single value.

Continuation at every reduction step

Let us take another look at this expression: `(+ (* 3 4) 5) => 17`

```
(+ (* 3 4) 5)          k: (lambda (v) v)
(* 3 4)                k: (lambda (v) (+ v 5))
3                      k: (lambda (v) (+ (* v 4) 5))
4                      k: (lambda (v) (+ (* 3 v) 5))
5                      k: (lambda (v) (+ (* 3 4) v))
```

The point is that there are multiple continuation points implicit at every step of evaluation in any program.

Scheme self-evaluates primitive types like numbers and strings. When we type them into the REPL, what is their continuation?

```
> 5
5
```

It is the identity function: **`(lambda (x) x)`**

Continuations and Stack

More examples

What is the *context* of the expression `(* 3 4)` in

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ (* 3 4) 5) 2))
```

Step #1: Replace the expression of interest with ?.

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ ? 5) 2))
```

Step #2: Wrap the expression in a lambda, simplify if possible.
`(zero? 5) => #f`, so:

```
(lambda(?)
  (* (+ ? 5) 2))
```

More Examples

What is the context of the expression `(* 3 4)` in:

```
(let ([n 1])
  (if (zero? n)
    (display (+ 3 (* 4 (+ 5 6))))
    (display (* (+ (* 3 4) 5) 2)))
  n)
```

Step #1:

```
(begin
  (display (* (+ ? 5) 2))
  n)
```

Step #2:

```
(lambda (?)
  (begin
    (display (* (+ ? 5) 2))
    n))
```

with `n` bound to `1`.

What is the point?

- makes control flow explicit.
- semantics for describing any control flow operation (loops, exceptions, generators, co-routines, ..).
- Continuations are present in every program in every programming language. Some programming languages provide first class support for continuations, which means, we can capture them in an binding, pass it to other functions, call them later ..
- That brings us to **`call/cc`**.

call with current continuation

Back to our original example: `(+ (* 3 4) 5)`

How can we “capture” the context of (* 3 4) into a variable binding so that we can return to the same point (for whatever reasons) later?

Scheme provides a construct called **`call-with-current-continuation`** or **`call/cc`** that does exactly this.

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

Notice two things in the above expression:

- We are not using k.
- k represents the continuation: **`(lambda (v) (+ v 5))`**

What if we use k?

We can apply the k:

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

Let us capture k in a global variable:

```
(define *k* (lambda(x) x))

(+ (call/cc
     (lambda(k)
       (begin
         (set! *k* k)
         (k (* 3 4)))))
   5)
```

Now, *k* is a continuation represented by the procedure: `(lambda (v) (+ v 5))`

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

i.e. *k* represents the rest of computation with respect to the expression `(* 3 4)`

non-local exit

Let us say we are multiplying a bunch of numbers and if one of them is zero, we don't want to continue and instead exit immediately.

```
(define (product ls)
  (call/cc
    (lambda (break)
      (let f ([ls ls])
        (cond
          [(null? ls) 1]
          [(= (car ls) 0) (break 0)]
          [else (* (car ls) (f (cdr ls)))])))))

(product '(1 2 3 4 5)) => 120
(product '(2 5 3 8 0 2 7 8) => 0
```

Another complicated example

```
(let ([x (call/cc (lambda (k) k))])
  (x (lambda (ignore) "hi")))
=> "hi"
```

what is continuation k?

```
(lambda(v)
  (let ([x v])
    (x (lambda(ignore) “hi”))))
```

The call/cc captures k but returns that value which is bound to x.
We apply this continuation to the parameter, `(lambda(ignore) “hi”)`

Now,

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

Here, x gets bound to `(lambda(ignore) “hi").`

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

Continuation Passing Style

Let us return to the factorial example:

```
(define (fact n)
  (cond [(zero? n) 1]
        [else (* n (fact (- n 1)))]))

> (fact 5)
120
```

If you trace the execution of `(fact 5)`:

```
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120
```

TCO

Accumulator Passing Style:

```
(define (fact-acc n acc)
  (cond [(zero? n) acc]
        [else (fact-acc (- n 1) (* n acc))]))
(define (fact n) (fact-acc n 1))

(fact 5)
(fact-acc 5 1)
(fact-acc 4 5*1)
(fact-acc 3 4*5*1)
(fact-acc 2 3*4*5*1)
(fact-acc 1 2*3*4*5*1)
(fact-acc 0 1*2*3*4*5*1)
> 120
```

There is another way to achieve TCO.

Continuation Passing Style or CPS

```
(define (fact/k n k)
   (cond [(zero? n) (k 1)]
         [else (fact/k (- n 1)
                       (lambda (v)
                         (k (* v n))))]))
(define (fact n) (fact/k n (lambda(p) p)))
```

This too uses a constant amount of stack. How? We added a second argument k that carries the continuation around.

When a procedure A calls another procedure B, via a non-tail call, the procedure B receives an implicit continuation that is responsible for completing what is left of the calling procedure's body plus returning to the calling procedure's continuation. If the call is a tail call, the called procedure simply receives the continuation of the calling procedure.

More examples

```
(define (sum-list xs)
  (if (empty? xs)
      0
      (+ (car xs) (sum-list (cdr xs)))))


(define (sum-list/k xs kont)
  (if (empty? xs)
      (kont 0)
      (sum-list/k (cdr xs)
                  (lambda(sum-cdr-xs)
                    (kont (+ (car xs) sum-cdr-xs))))))

> (sum-list xs (lambda(x) x))
```

More examples

Caution: This is an extremely inefficient way of finding a fibonacci sequence.

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1) (fib (- n 2))))))
```

```
(define (fib/k n k)
  (if (< n 2)
      (k 1)
      (fib/k (- n 1) (lambda (k1)
                       (fib/k (- n 2)
                              (lambda(k2)
                                (k (+ k1 k2)))))))))
```

Notice how we have explicitly serialised the computation. The argument evaluation is now explicit (we go from left to right).

More examples

```
(define (odd? x)
  (cond [(= x 0) #f]
        [(= x 1) #t]
        [else (odd? (- x 2))]))

(define (odd/k? x k)
  (cond [(= x 0) (k #f)]
        [(= x 1) (k #t)]
        [else (odd/k? (- x 2) (lambda (odd-n-2?)
                                (k odd-n-2?)))]))
```

which is the same as:

```
(define (odd/k? x k)
  (cond [(= x 0) (k #f)]
        [(= x 1) (k #t)]
        [else (odd/k? (- x 2) k)]))
```

More examples

```
(define (filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (car lst))
         (cons (car lst)
               (filter pred? (cdr lst)))]
        [else (filter pred? (cdr lst))]))

(define (filter/k pred/k? lst k)
  (cond [(empty? lst) (k empty)]
        [else (pred/k? (car lst)
                       (lambda (pred-result?)
                         (if pred-result?
                           (filter/k pred/k?
                                     (cdr lst)
                                     (lambda (filter-rest)
                                       (k (cons (car lst)
                                                filter-rest))))
                           (filter/k pred/k?
                                     (cdr lst)
                                     k)))))]))
```

Implementation

CPS can be implemented as a source to source transformation and is usually done as a compiler pass. On a Scheme/Racket system, CPS can be done using a macro.

We can also write a CPS interpreter “by-hand” though it becomes painful.

Resources

- “Structure and Interpretations of Computer Programs”, 2/e, Abelson and Sussman, MIT Press.
- “RABBIT: A Compiler for SCHEME” by Guy L. Steele, Jr (MIT AI Lab memo - AITR-474.pdf)
- “Scheme and the art of programming” by Friedman and Springer.
- “Programming languages: Applications and Interpretations”, 1/e and 2/e, Shriram Krishnamurthi.
- “The Scheme programming language” 4/e, by R. Kent Dybvig.
- Will Byrd’s Google Hangout on Continuations.

Questions?

ram@rkrishnan.org
@vu3rdd

“A language that doesn't affect the way you think about programming, is not worth knowing.”
- Alan Perlis (“Epigrams in Programming”)