Mattox's Guide to Continuations

©April, 2002
Mattox Beckman

1 Introduction

The purpose of this document is to give the reader an introduction to continuation passing style and continations.

2 Basic Continuations

2.1 Forward Recursion

Consider the program in figure 1. It simply multiplies a list of integers together.

```
(define (multlist lst)
   (if (null? lst) 1
     (* (car lst) (multlist (cdr lst)))))
```

Figure 1: Forward Recursion

If we were to run it with the input `'(2 3 4)` we would get the execution trace in figure 2.

```
(multlist '(2 3 4))
(* 2 (multlist '(3 4)))
(* 2 (* 3 (multlist '(4))))
(* 2 (* 3 (* 4 (multlist '()))))
(* 2 (* 3 (* 4 1)))
(* 2 (* 3 4))
(* 2 12)
24
```

Figure 2: Forward Recursion Sample Run

As the computation progresses, we (literally) stack up recursive calls until we reach the base case, and then return from all the recursive calls to compute the result.

Consider what's happening in line 2 of the sample run: `(* 2 • (multlist '(3 4)))`. A bullet point has been added: it represents a split in the computation. The stuff after the bullet (`(multlist '(3 4))`) is going to be evaluated, and the value given to the stuff that comes before the bullet. When we reach the bullet point, we need to save this location on a stack as we make the function call, because when that function returns we need to give its result back to the rest of the expression, which will process the result further.

2.2 Accumulator Recursion

Another style of recursion avoids the use of the stack by accumulating the value in a separate parameter. The code is in figure 3.

If we were to run it with the input `'(2 3 4)` and 1 we would get

Notice this time that we only need to keep track of a single call to `multlist` at any one time... the result of one call is simply the value of the next call. This is known as `tail-recursion`, and is useful because we

```
(define (amultlist lst acc)
  (if (null? lst) acc
    (amultlist (cdr lst) (* (car lst) acc))))
```

Figure 3: Accumulator Recursion

```
(amultlist '(2 3 4) 1)
(amultlist '(3 4) (* 2 1))
(amultlist '(3 4) 2)
(amultlist '(4) (* 3 2))
(amultlist '(4) 6)
(amultlist '() (* 4 6))
(amultlist '() 24)
24
```

Figure 4: Sample Run of figure 3

do not need to keep track of where in the main expression the result needs to be placed. The result is not going to be handed back to anyone to be processed further, as it was in figure 1. It is simply returned.

When written in this form, the compiler will eliminate the call-stack, which will speed things up.

Notice how the computations are performed. In forward recursion example, the recursions happened first, and the computation occurred as we returned from the recursions. In this examples, the computation occurs first, and are passed into the recursive call.

2.3 Continuation Passing

Figure 3 accumulates a value to be given to the recursive call, but that's not the only kind of accumulation we can make. Instead of accumulating values, we can also accumulate *computations*. Consider figure 5. This is an example of *continuation-passing-style*, which we will refer to as CPS.

```
(define (kmultlist lst k)
  (if (null? lst) (k 1)
    (kmultlist (cdr lst) (lambda (v) (k (* v (car lst)))))))
```

Figure 5: Continuation Passing `multlist`

The main idea behind CPS is that *functions never return*, and therefore you need to specify what should happen next when a result is computed. This is done by giving each function an extra argument, called the *continuation*. This argument usually comes last, and is often called `k`. When the funtion is finished computing its result, it will pass that result into `k` instead of returning.

If figure 5 we can see how the continuations are used. If `kmultlist` is called with the empty list, it is as the base case, which is defined to be 1. So, `kmultlist` passes 1 to `k`. The recursive case is more complex, and may seem unusual at first. Suppose that `kmultlist` is called with some non-empty list `lst`—refer to this as the initial call. In order to compute the result, `kmultlist` needs to make a recursive call on the `cdr` of `lst`. Since this recursive call will not return, we need to give it a continuation to tell it what to do with that result. This continuation should save the result of the recursive call, multiply it by the `car` of `lst`, and then pass that new result to the continuation given to the initial call. All this is done in the last line. The variable `v` saves the result of the recursive call to `kmultlist`, the continuation then multiplies `v` by `(car lst)`, and then passes the result to the initial continuation `k`.

An execution trace is given in figure 6. Note that `Id` is `(lambda (x) x)`, or a function that takes an argument and returns it unchanged. You can think of it as a "final continuation" or something like that. In real life, a different function would be passed, such as `display`.

```
(kmultlist '(2 3 4) Id)
(kmultlist '(3 4) (lambda (v1) (Id (* v1 2))))
(kmultlist '(4) (lambda (v2) ((lambda (v1) (Id (* v1 2))) (* v2 3))))
(kmultlist '() (lambda (v3) ((lambda (v2) ((lambda (v1) (Id (* v1 2))) (* v2 3))) (* v3 4))))
( (lambda (v3) ((lambda (v2) ((lambda (v1) (Id (* v1 2))) (* v2 3))) (* v3 4))) 1)
((lambda (v2) ((lambda (v1) (Id (* v1 2))) (* v2 3))) (* 1 4))
((lambda (v2) ((lambda (v1) (Id (* v1 2))) (* v2 3))) 4)
((lambda (v1) (Id (* v1 2))) (* 4 3))
((lambda (v1) (Id (* v1 2))) 12)
(Id (* 12 2))
(Id 24)
24
```

Figure 6: Sample Run of Continuation Passing `multlist`

As you can see, at each stage of the recursion we take the old continuation and build a new, larger continuation out of it. Consider line two as an example. The function `kmultlist` is supposed to compute the product of `'(2 3 4)`, and pass the result to `Id`. To do that, it's going to call itself recursively, on the list `'(3 4)`. The result (12) will be placed into v_1, and then the function will multiply it by the current element of the list (2). The current call to `kmultlist` is now ready with its result (24), so it isgiven to the continuation specified for this particular call (i.e., `Id`).

In all of these coding styles, the problem is split into two parts. To multiply the elements of a list, you first multiply the elements of the rest (i.e., the `cdr`) of the list, and when you're done doing that, you multiply the result to the first element of the list.

- In the forward recursion, the recursive call multiplies all the elements together and returns the result to the initial call.
- In the accumulator recursion, the multiplication occurs first, and is given to the recursive call.
- In the CPS version, the recursive call multiplies all the elements together, and then passes the result to the current continuation.

3 Multiple Continuations

You can think of continuations as being a return address, made explicit. Because continuations can be stored in variables, there is no reason we can't keep more than one of them around. In figure 7 there is a new version of the `kmultlist` function, that can "bail out" of a computation. The front end to this function sends two copies of the initial continuation to the auxiliary function. The first copy (`kr`, the "result" continuation) is used as before, but the second copy (`ka`, the "abort" continuation) is kept unchanged. The continuation `kr` represents the function's computation; call it, and the computation occurs. The `ka` continuation, if called, will skip all that computation and exit out of all the levels of recursion, back to the beginning.

Figure 8 shows a sample run where the abort continuation is called. Again, the first continuation argument allows you to pass a result back to the previous recursive call's computation, and the second continuation allows you to pass a result back to the initial function call.

For that matter, why stop at two continuations? This next version interprets negative numbers in a strange way: if it finds −`n`, it undoes `n` levels of recursion, giving the result 1.

The following sample run may illustrate the operation of the new function. Be sure you understand how it works.

```
(define (kmultlist lst k)
  (kmultlistaux lst k k))

(define (kmultlistaux lst kr ka)
  (if (null? lst) (kr 1)
      (if (= (car lst) 0) (ka 0)
          (kmultlistaux (cdr lst) (lambda (v) (kr (* v (car lst)))) ka))))
```

Figure 7: Aborting Continuation

```
(kmultlist '(2 3 0) Id)
(kmultlistaux '(2 3 0) Id Id)
(kmultlistaux '(3 0) (lambda (v1) (Id (* v1 2))) Id)
(kmultlistaux '(0) (lambda (v2) ((lambda (v1) (Id (* v1 2))) (* v2 3))) Id)
(Id 0)
0
```

Figure 8: Sample Run of Continuation Passing `multlist`

```
(define (nth n lst)
   (if (= n 0) (car lst)
       (nth (- n 1) (cdr lst))))


(define (kmultlist lst k)
  (kmultlistaux lst (list k) k))

(define (kmultlistaux lst krl ka)
  (if (null? lst) ((car krl) 1)
      (if (= (car lst) 0) (ka 0)
          (if (< (car lst) 0) ((nth (- (car lst)) krl) 1)
             (kmultlistaux (cdr lst)
                 (cons (lambda (v) ((car krl) (* v (car lst))))
                       krl)
                 ka)))))
```

Figure 9: Aborting Continuation, Part 2

```
> (kmultlist (list 2 3 4 5 6 1) (lambda (x) x))
720
> (kmultlist (list 2 3 4 5 6 1 0 2 4 6) (lambda (x) x))
0
> (kmultlist (list 2 3 4 5 6 -1 0 2 4 6) (lambda (x) x))
120
> (kmultlist (list 2 3 4 5 6 -2 0 2 4 6) (lambda (x) x))
24
```

Continuations, then, are a means of time travel. A continuation allows us to save a point of the program and return to it any time we want.

4 Flow Control

One major benefit of CPS is that it causes a program's flow of control to be made explicit. Consider the following session:

```
> (define (add a b) (+ a b))
> (add (add 2 3) (add 5 6))
16
```

Scheme does not give us any guarantee about what order operations occur. In this particular example, it doesn't matter which order we take, but it's not hard to come up with examples (usually involving `set!` and the like) where it does make a difference.

Suppose we make a new function `kadd` that is written in CPS. Instead of returning, `kadd` will pass its result to a continuation.

```
> (define (id x) x)
> (define (kadd a b k)
    (k (+ a b)))
> (kadd 2 3 id)
5
```

What happens if we try to rewrite `(add (add 2 3) (add 5 6))` using `kadd`? Try it yourself first, before going on.

The answer is this:

```
> (kadd 2 3
    (lambda (v) (kadd 5 6
       (lambda (w) (kadd v w id)))))
16
```

Or, maybe you did it this way:

```
> (kadd 5 6
    (lambda (v) (kadd 2 3
       (lambda (w) (kadd v w id)))))
16
```

Either way, you had to pick which of the `kadd` function calls happened first, and store that in `v`. The second `kadd` result was stored in `w`. The final `kadd` used both `v` and `w`. It is now clear to whoever cares to read this program what operation will happen, no matter what order the underlying programming language happens to pick for evaluating parameters. In fact, you can even change the parameter passing style from call-by-value to call-by-need without changing the result.

5 The `call/cc` function

Consider the expression `(+ 2 (multlist '(2 3 4)) 10)`. At some point, the call to `multlist` is going to be made, and the result placed inside of the rest of the expression `(+ 2 [·] 10)` (where the [·] is). The code surrounding the [·] is called a *context*, and can be represented by using a function, like this one: `(lambda (v) (+ 2 v 10))`. If we only had the `kmultlist` version of our list multiplier, we could use this trick to supply its continuation, like this: `(kmultlist '(2 3 4) (lambda (v) (+ 2 v 10)))`.

If we leave the backets in the original code, we have a way of delimiting our program into two halves, much like what we did with the forward recursion in section 2.1. The code inside of the brakets is executed first, and the result is given to the program outside of the brackets. In other words, the code outside of the brackets represents "the rest of the program."

(+ 2 [(multlist '(2 3 4))] 10)

It would be convenient sometimes if we could somehow take "the rest of the program" and turn it into a continuation automatically. This is the purpose of `call/cc`. What `call/cc` does is create a continuation representing what is just outside its scope, and passes that continuation to its first argument. In this example, it would look like this:

```
> (+ 2 (call/cc (lambda (k) (kmultlist '(2 3 4) k))) 10)
```

The `call/cc` will automatically turn the rest of the program into a function `(lambda (v) (+ 2 v 10))` and pass it into the argument, saving it as `k`.

We can use `call/cc` to write an aborting version of `multlist` that does *not* use CPS.

```
> (define (amultlist lst)
     (call/cc (lambda (k) (amultlistaux lst k))))
> (define (amultlistaux lst k)
     (if (null? lst) 1
         (if (= (car lst) 0) (k 0)
             (* (amultlistaux (cdr lst) k) (car lst)))))

> (amultlist '(2 3 4))
24
> (amultlist '(2 3 0 4))
0
```

The `amultlist` function saves its current continuation and passes that to the auxiliary function, which makes use of it if it needs to break out of the recursion. This technique is often used to implement exceptions: imagine that `call/cc` was renamed to `try`, and the variable `k` was renamed to `throw`.