TUTORIAL

From Direct Style to Monadic Style through Continuation-Passing Style (Draft: September 12, 2002)

Daniel P. Friedman †

Computer Science Department, Indiana University
Bloomington, IN 47405, USA

1 Tranforming to Monadic Style

There is a relationship between monadic style and continuation-pasing style. In these notes, we show how to take a typical program, put it into continuation-passing style and then through a series of correctness-preserving transformations, yield code in monadic style. All but one correctness-preserving operation is trivial and the one that is nontrivial is simple enough that it requires little explanation. The program is sufficently general that knowing how to transform this program from direct style to continuation-passing style and then to monadic style should suffice for all programs.

Both styles have their advantages and because of familiarity with continuation-passing style, we start there, but after understanding monadic style, it may be natural to use it as the base concept.

Much has been written about monadic style and simply finding your way to Phil Wadler's (tutorial style) or Eugenio Moggi's (formal style) website would suffice for additional reading, if you want to see other perspectives.

Our goal is simple. Take a single program, `rember8`, in direct style and rewrite it in monadic style. So, here are the proposed before and after programs.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) '()]
      [(pair? (car ls))
       (cons (rember8 (car ls)) (rember8 (cdr ls)))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (cons (car ls) (rember8 (cdr ls)))])))
```

† ...

transforms to this,

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (bind (rember8 (car ls))
         (lambda (a)
           (bind (rember8 (cdr ls))
             (lambda (d)
               (unit (cons a d))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (bind (rember8 (cdr ls))
              (lambda (d)
                (unit (cons (car ls) d))))])))

(define unit
  (lambda (value)
    (lambda (k)
      (k value))))

(define bind
  (lambda (m w)
    (lambda (k^)
      (m (lambda (v) ((w v) k^))))))
```

How do we know when we have a legitimate `unit`/`bind` pair? In order for a pair of functions to be compatible, they must satisfy these three equations:

```
(bind (unit a) w) = (w a)

    (bind m unit) = m

(bind m (bind w h)) = (bind (bind m w) h)
```

The `w` and `h` in the last equation are not correct because of potential nontermination, so it is safer to apply η^{-1} to them when the equation is stated.

Monads can be used to give a semantics of various "computational effects" (such as state, exceptions, or I/O) in functional programming languages. The ideas are based on "Moggi's Principle":

> Computations of type α correspond to values of type Tα. Informally, `(unit` α`)` represents a pure ("effect-free") computation yielding α, while `(bind` m k`)` represents the computation consisting of m's effects followed by the result of applying k to the value computed by m.

The first two steps transform `rember8` into a variant that has been put into continuation-passing style We have followed each of the first three steps by a tester. Thereafter, the tester does not change.

Here, again, is the original program, which removes all 8's from an arbitrary list whose atomic elements are numbers.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) '()]
      [(pair? (car ls))
       (cons (rember8 (car ls)) (rember8 (cdr ls)))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (cons (car ls) (rember8 (cdr ls)))])))

(define test-direct
  (lambda ()
    (equal?
      (rember8 '((((((1 8 3)) 8) 7) 8) 9))
      '((((((1 3))) 7)) 9))))
```

We first transform the original function into continuation-passing style. Recall the way that we can read it. "If the list is empty, we are done and we return the empty list. If the first element of the list is, itself, a list, then we assume that `a` is bound to the result of the recursive call on the `(car ls)` and assume that `d` is bound to the recursive call on the `(cdr ls)`, then we return the result of consing these two values together. If the `car` of the list is the number `8`, then we simply recur on the `cdr`. Otherwise, we assume that `d` is the result of the recursion on the `cdr` and we return the result of consing the `car` of the list to `d`. We observe that we must pass an initial continuation.

```
(define rember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]                              ; <--
      [(pair? (car ls))
       (rember8 (car ls)
         (lambda (a)                                    ; <--
           (rember8 (cdr ls)
             (lambda (d)                                ; <--
               (k (cons a d))))))]                      ; <--
      [(= (car ls) 8)
       (rember8 (cdr ls)
         k)]                                            ; <--
      [else (rember8 (cdr ls)
              (lambda (d)                               ; <--
                (k (cons (car ls) d))))])))             ; <--
```

```
(define test-cps
  (lambda ()
    (rember8 '((((((1 8 3)) 8) 7) 8) 9)
      (lambda (x) (equal? x '((((((1 3))) 7)) 9))))))  ; <--
```

In the next step, we curry `k`, so that instead of passing two arguments to `rember8`, we pass them in one at a time. This is a trivial transformation, since it simply means adding a pair of parentheses for every recursive call.

```
(define rember8
  (lambda (ls)
    (lambda (k)                                        ; <--
      (cond
        [(null? ls) (k '())]
        [(pair? (car ls))
         ((rember8 (car ls))                           ; <--
          (lambda (a)
            ((rember8 (cdr ls))                        ; <--
             (lambda (d)
               (k (cons a d))))))]
        [(= (car ls) 8)
         ((rember8 (cdr ls))                           ; <--
          k)]
        [else ((rember8 (cdr ls))                      ; <--
               (lambda (d)
                 (k (cons (car ls) d))))]))))

(define test-cps-curried.
  (lambda ()
    ((rember8 '((((((1 8 3)) 8) 7) 8) 9))              ; <--
     (lambda (x) (equal? x '((((((1 3))) 7)) 9))))))
```

The next step is a bit tricky and relies on our knowledge that the tests are really just doing a dispatch that does not rely on `k`. That is, the `cond` tests only invoke trivial primitives. Thus, we can *push* `(lambda (k)` from outside the `cond` to just past each clause's left-hand side. It is also the case that arguments passed to a continuation must be similarly simple. Furthermore, one other trivial "primitive" is the function constructor, `lambda`.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls)
       (lambda (k)                                    ; <--
         (k '()))]
      [(pair? (car ls))
       (lambda (k)                                    ; <--
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               (k (cons a d)))))))]
      [(= (car ls) 8)
       (lambda (k)                                    ; <--
         ((rember8 (cdr ls))
          k))]
      [else (lambda (k)                               ; <--
              ((rember8 (cdr ls))
               (lambda (d)
                 (k (cons (car ls) d)))))])))
```

η-conversion transforms something of the form `(lambda (x) (`M `x))` to M provided two things are true. First, `x` is not free in M and second, it can be determined that M terminates. (This is not completely accurate, but suffices for our purposes.) We η-convert the right-hand side of the third clause, since `(rember8 (cdr ls))` quits quickly returing something of the form `(lambda (k) ...)` even if all the remaining values in the list are 8s. Why? Because `(unit '())` returns a `(lambda (k) ...)`.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (lambda (k) (k '()))]
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               (k (cons a d)))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]             ; <--
      [else (lambda (k)
              ((rember8 (cdr ls))
               (lambda (d)
                 (k (cons (car ls) d)))))])))
```

Now, we introduce `unit` in the first clause. It is easy to see that `(lambda (k) (k '()))` is the same as `((lambda (v) (lambda (k) (k v))) '())` and `(lambda (v) (lambda (k) (k v)))` is `unit`.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]                          ; <--
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               (k (cons a d)))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (lambda (k)
              ((rember8 (cdr ls))
               (lambda (d)
                 (k (cons (car ls) d)))))])))

(define unit
  (lambda (v)
    (lambda (k)
      (k v))))
```

The next step allows us to wrap a `(let ((k k)) ...)` around a piece of code, since the outer k is bound to the inner k anyway, there is no change in the semantics. We model the `let`, however, with `((lambda (k) ...) k)`

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               (k (cons a d)))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (lambda (k)
              ((rember8 (cdr ls))
               (lambda (d)
                 ((lambda (k)                          ; <--
                    (k (cons (car ls) d)))
                  k))))]))))                           ; <--
```

Now, we recognize that `(lambda (k) (k (cons (car ls) d)))` is `(unit (cons (car ls) d))`.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               (k (cons a d)))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (lambda (k)
              ((rember8 (cdr ls))
               (lambda (d)
                 ((unit (cons (car ls) d))             ; <--
                  k))))])))
```

We apply the same two steps in the second clause.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               ((unit (cons a d))                    ; <--
                k))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else  (lambda (k)
               ((rember8 (cdr ls))
                (lambda (d)
                  ((unit (cons (car ls) d))
                   k))))])))
```

For now, we work only on the last clause. We want to move the `(lambda (k)` above the `(lambda (d) ...)`. To do that, requires inventing a procedure `bind*` that connects the two pieces together.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               ((unit (cons a d))
                k))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (bind*                                    ; <--
              (rember8 (cdr ls))
              (lambda (k)                             ; <--
                (lambda (d)
                  ((unit (cons (car ls) d))
                   k))))])))

(define bind*
  (lambda (m q)
    (lambda (k^)
      (m (q k^)))))
```

Now, comes the trickiest part of the whole transformation. We push the `k` in even further, but must introduce a rather subtle function, `bind`, whidh does the connecting.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               ((unit (cons a d)) k))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (bind                                    ; <--
              (rember8 (cdr ls))
              (lambda (d)
                (lambda (k)                          ; <--
                  ((unit (cons (car ls) d))
                   k))))])))

(define bind
  (lambda (m w)
    (lambda (k^)
      (m (lambda (v) ((w v) k^))))))
```

Now, we can do an η-conversion causing `k` to vanish from the last clause.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (lambda (k)
         ((rember8 (car ls))
          (lambda (a)
            ((rember8 (cdr ls))
             (lambda (d)
               ((unit (cons a d))
                k))))))]
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (bind
              (rember8 (cdr ls))
              (lambda (d)
                (unit (cons (car ls) d))))])))         ; <--
```

Last, we transform both recursive calls in the second clause, yielding the promised result of causing the threaded `k` to disappear completely.

```
(define rember8
  (lambda (ls)
    (cond
      [(null? ls) (unit '())]
      [(pair? (car ls))
       (bind                                          ; <--
         (rember8 (car ls))
         (lambda (a)
           (bind                                      ; <--
             (rember8 (cdr ls))
             (lambda (d)
               (unit (cons a d))))))]                 ; <--
      [(= (car ls) 8) (rember8 (cdr ls))]
      [else (bind
              (rember8 (cdr ls))
              (lambda (d)
                (unit (cons (car ls) d))))])))
```

See if you can define a `unit`/`bind` pair that produces the semantics of the original program. The primary purpose of extra arguments being threaded throughout a set of functions is to model various flavors of side-effects. What we have seen in this short note is that once a program is in monadic style, it is easy to hide the threading in `unit` and `bind`. Later, we see examples of `unit` and `bind` that are defined for a slightly more complicated program, an interpreter.

Final observation: If you look closely at the definition of `bind`, you will notice that the argument to `m` is a function, so that the work of `w` clearly takes place after the work of `m`.