TUTORIAL

Direct Style from Monadic Style and Back
(Draft: September 15, 2002)

Daniel P. Friedman ♂
Computer Science Department, Indiana University
Bloomington, IN 47405, USA

Consider the following variation on a problem suggested by Mitch Wand. Given a nonnegative integer or a proper list (possibly nested) of nonnegative integers, copy the list, but replace all integers by the largest integer seen so far. The function’s behavior is unspecified for all other values.

(define traverse
  (letrec
    ([traverse
       (lambda (t s)
         (cond
           [(null? t) (CONS '() s)]
           [(integer? t)
             (let ([m (max t s)]
                   (CONS m m))]
           [else
             (let ([pr (traverse (car t) s))]
               (let ([pr-d (traverse (cdr t) (CDR pr))])
                 (CONS (cons (CAR pr) (CAR pr-d)) (CDR pr-d))))))])
      (lambda (t)
        (CAR (traverse t 0))))))

Each invocation of traverse produces a pair of values: the copied term and the largest number seen so far. The CONSs, CARs, and CDRs are for making and accessing the parts of such pairs, but recall that Scheme does not distinguish between upper and lower case symbols, so they are still just cons, cars, and cdrs.

(define test-traverse
  (lambda ()
    (traverse '((((2 4 (1 5 3) 7 6) 4) 2 4))))

> (test-traverse)
((((2 4 (4 5 5) 7 7) 7) 7 7))

...
Here is the monadic-style solution with definitions of unit, unit-max, and bind.

(define unit (lambda (v) (lambda (s) (CONS v s)))))
(define unit-max (lambda (v) (lambda (s) (let ([m (max v s)]) (CONS m))))))
(define bind
(lambda (m w)
 (let ([pr (pr m)])
 (CONS (CAR pr) (CDR pr))))))

(define traverse
(letrec
 ([(traverse
 (lambda (t)
 (cond
 [(null? t) (unit '())]
 [(integer? t) (unit-max t)]
 [else
 (bind
 (traverse (car t))
 (lambda (a)
 (bind
 (traverse (cdr t))
 (lambda (d)
 (unit (cons a d))))))]]))
 (lambda (t)
 (CAR ((traverse t) 0)))))

In the previous note we derived bind and unit for a continuation monad. This
time, we derive direct style from monadic style where its bind and unit produce
the state monad.
Step 1: Replace unit (in the last clause) by its definition.

(define traverse
(letrec
  ([traverse
     (lambda (t)
      (cond
        [(null? t) (unit '())]
        [(integer? t) (unit-max t)]
        [else
         (bind
          (traverse (car t))
          (lambda (a)
           (bind
            (traverse (cdr t))
            (lambda (d)
             ((lambda (v)
                (lambda (s)
                 (CONS v s)))
              (cons a d))))))])
     (lambda (t)
      (CAR ((traverse t) 0)))]))

Step 2: \(\beta\) convert v. Technically, \(\beta\) conversion is over the whole application, but since there is only one application of \(\lambda\) \(\lambda\) \(\lambda\) \(\ldots\), we choose to use this looser characterization.

(define traverse
(letrec
  ([traverse
     (lambda (t)
      (cond
        [(null? t) (unit '())]
        [(integer? t) (unit-max t)]
        [else
         (bind
          (traverse (car t))
          (lambda (a)
           (bind
            (traverse (cdr t))
            (lambda (d)
             ((lambda (s)
                (CONS (cons a d) s))))))])
     (lambda (t)
      (CAR ((traverse t) 0)))]))
Step 3: Replace (the inner) bind by its definition.

(define traverse
 (letrec
   (([traverse
      (lambda (t)
       (cond
         [null? t) (unit ’())
         [integer? t) (unit-max t)]
        [else
         (bind
          (traverse (car t))
          (lambda (a)
           ((lambda (m w)
              (lambda (s)
               (let ([pr (m s)])
                (w (CAR pr)) (CDR pr))))
              (traverse (cdr t))
              (lambda (d)
               (lambda (s)
                (CONS (cons a d) s)))))))]
        (lambda (t)
          (CAR ((traverse t) 0)))))

Step 4: β convert m and w.

(define traverse
 (letrec
   (([traverse
      (lambda (t)
       (cond
         [null? t) (unit ’())
         [integer? t) (unit-max t)]
        [else
         (bind
          (traverse (car t))
          (lambda (a)
           (lambda (s)
            (let ([pr ((traverse (cdr t)) s)])
             (((lambda (d)
                (lambda (s)
                 (CONS (cons a d) s)))
                (CAR pr))
             (CDR pr))))))))
        (lambda (t)
          (CAR ((traverse t) 0)))))}
Step 5: β convert d.

(define traverse
  (letrec
   ([traverse
      (lambda (t)
       (cond
        [[(null? t) (unit '())]
        [[(integer? t) (unit-max t)]
        [else
         (bind
          (traverse (car t))
          (lambda (a)
           (lambda (s)
            (let ([pr ((traverse (cdr t)) s)])
             ((lambda (s)
                (CONS (cons a (CAR pr)) s))
              (CDR pr))))))))])
    (lambda (t)
      (CAR ((traverse t) 0))))))

Step 6: β convert s. Now we are done transforming the inner bind.

(define traverse
  (letrec
   ([traverse
      (lambda (t)
       (cond
        [[(null? t) (unit '())]
        [[(integer? t) (unit-max t)]
        [else
         (bind
          (traverse (car t))
          (lambda (a)
           (lambda (s)
            (let ([pr ((traverse (cdr t)) s)])
             (CONS (cons a (CAR pr)) (CDR pr)))))))])
    (lambda (t)
      (CAR ((traverse t) 0))))))
Step 7: Replace (the remaining) bind by its definition.

(define traverse
  (letrec
    ([traverse
       (lambda (t)
         (cond
         [(null? t) (unit '())]
         [(integer? t) (unit-max t)]
         [else
          ((lambda (m w)
             (lambda (s)
               (let ([pr (m s)])
                  ((w (CAR pr)) (CDR pr)))))
            (traverse (car t))
            (lambda (a)
              (lambda (s)
                (let ([pr ((traverse (cdr t)) s)])
                  (CONS (cons a (CAR pr)) (CDR pr)))))))]
          (lambda (t))
          (CAR ((traverse t) 0)))]))

Step 8: β convert m and w.

(define traverse
  (letrec
    ([traverse
       (lambda (t)
         (cond
         [(null? t) (unit '())]
         [(integer? t) (unit-max t)]
         [else
          (lambda (s)
            (let ([pr ((traverse (car t)) s)])
              (((lambda (a)
                 (lambda (s)
                   (let ([pr ((traverse (cdr t)) s)])
                     (CONS (cons a (CAR pr)) (CDR pr)))))
                 (CAR pr))
                 (CDR pr)))]])]
          (lambda (t))
          (CAR ((traverse t) 0)))]))
Step 9: $\beta$ convert a. We have to be careful, here. We can’t just substitute a by (CAR pr). Why? The pr in (CAR pr) would be captured. We get around this problem by renaming the inner pr to pr-d. Always rename the inner one, since we often can’t be sure what coming in. This renaming is called $\alpha$ conversion. The variable and its free occurrences in its body are renamed. It is formally defined over lambda expressions, but recall that (let ([x e]) b) is just shorthand for ((lambda (x) b) e). But, it should be obvious that the names picked for the variables can be changed, provided that we do it carefully.

(define traverse
  (letrec
   ([traverse
      (lambda (t)
        (cond
          [(null? t) (unit '())]
          [(integer? t) (unit-max t)]
          [else
           (lambda (s)
             (let ([pr ((traverse (car t)) s)])
               (let ([s (lambda (s)
                          (let ([pr-d ((traverse (cdr t)) s)])
                            (CONS (cons (CAR pr) (CAR pr-d)) (CDR pr-d)))])))]))
             (lambda (t)
               (CAR ((traverse t) 0))))))

Step 10: $\beta$ convert the inner s. This completes the transformations of the outer bind.

(define traverse
  (letrec
   ([traverse
      (lambda (t)
        (cond
          [(null? t) (unit '())]
          [(integer? t) (unit-max t)]
          [else
           (lambda (s)
             (let ([pr ((traverse (car t)) s)])
               (let ([pr-d ((traverse (cdr t)) (CDR pr))])
                 (CONS (cons (CAR pr) (CAR pr-d)) (CDR pr-d)))))]))
             (lambda (t)
               (CAR ((traverse t) 0))))))
Step 11: Replace \texttt{unit-max} by its definition.

\begin{verbatim}
(define traverse
  (letrec
   ([traverse
      (lambda (t)
        (cond
         [(null? t) (unit '())]
         [(integer? t)
          (lambda (v)
             (lambda (s)
              (let ([m (max v s)]
                   (CONS m m)))]
              t)]
         [else
          (lambda (s)
           (let ([pr ((traverse (car t)) s)]
                (let ([pr-d ((traverse (cdr t)) (CDR pr)))]
                 (CONS (cons (CAR pr) (CAR pr-d)) (CDR pr-d)))))])
              (lambda (t)
               (CAR ((traverse t) 0))))]))

Step 12: $\beta$ convert \texttt{v}.

\begin{verbatim}
(define traverse
  (letrec
   ([traverse
      (lambda (t)
        (cond
         [(null? t) (unit '())]
         [(integer? t)
          (lambda (s)
           (let ([m (max t s)]
                (CONS m m)))]
          [else
           (lambda (s)
            (let ([pr ((traverse (car t)) s)]
                 (let ([pr-d ((traverse (cdr t)) (CDR pr)))]
                  (CONS (cons (CAR pr) (CAR pr-d)) (CDR pr-d)))))])])
        (lambda (t)
         (CAR ((traverse t) 0))))]))
\end{verbatim}
Step 13: Replace unit by its definition.

(define traverse
  (letrec
    ([traverse
      (lambda (t)
        (cond
          [(null? t)
            (lambda (v)
              (lambda (s)
                (CONS v s)))]
          [(integer? t)
            (lambda (s)
              (let ([m (max t s)]
                (CONS m m)))]
          [else
            (lambda (s)
              (let ([pr ((traverse (car t)) s)]
                [pr-d ((traverse (cdr t)) (CDR pr))]
                (CONS (cons (CAR pr) (CAR pr-d)) (CDR pr-d))))]])
      (CAR ((traverse t) 0))))))

Step 14: β convert v.

(define traverse
  (letrec
    ([traverse
      (lambda (t)
        (cond
          [(null? t)
            (lambda (s)
              (CONS '() s))]
          [(integer? t)
            (lambda (s)
              (let ([m (max t s)]
                (CONS m m)))]
          [else
            (lambda (s)
              (let ([pr ((traverse (car t)) s)]
                [pr-d ((traverse (cdr t)) (CDR pr))]
                (CONS (cons (CAR pr) (CAR pr-d)) (CDR pr-d))))]])
      (CAR ((traverse t) 0))))))
Step 15: Lift (lambda (s) ...) above cond clause. This is possible, since the right-hand side of each clause is of the form (lambda (s) ...).

(define traverse
  (letrec
    ([traverse
       (lambda (t)
         (lambda (s)
           (cond
             [null? t] (CONS '() s)]
             [[integer? t]
              (let ([m (max t s)]
                     (CONS m m))]
             [else
              (let ([pr ((traverse (car t)) s)]
                  (let ([pr-d ((traverse (cdr t)) (CDR pr))]
                        (CONS (CONS (CAR pr) (CAR pr-d)) (CDR pr-d))))))])
            (lambda (t)
              (CAR ((traverse t) 0))))))

Step 16: Uncurry the result.

(define traverse
  (letrec
    ([traverse
       (lambda (t s)
         (cond
           [null? t] (CONS '() s)]
           [[integer? t]
            (let ([m (max t s)]
                  (CONS m m))]
           [else
            (let ([pr ((traverse (car t)) s)]
                (let ([pr-d (traverse (cdr t)) (CDR pr)]
                    (CONS (CONS (CAR pr) (CAR pr-d)) (CDR pr-d))))))])
            (lambda (t)
              (CAR ((traverse t 0))))))

The transformations given here are reversible, so we can start at either end and produce the appropriate result. (Exercise: start with the direct style and produce monadic style.)