Continuation-Passing Style in Scheme

The original factorial function is:

\[
\text{(define (fact n)}
\quad (\text{if} \; (= \; n \; 0)
\quad \text{1}
\quad (\ast \; n \; (\text{fact} \; (- \; n \; 1))))
\]

This is a recursive procedure (it contains a \textit{recursive call}) and its evaluation on some non-negative integer exhibits a \textit{linear recursive process}, as explained in Section 1.2 of the [A&S] book. For example, its evaluation on input 3 produces the following sequence:

\[
\begin{align*}
\text{(fact 3)} \\
\ast \; 3 \; (\text{fact 2}) \\
\ast \; 3 \; (* \; 2 \; (\text{fact 1})) \\
\ast \; 3 \; (* \; 2 \; (* \; 1 \; (\text{fact 0}))) \\
\ast \; 3 \; (* \; 2 \; (* \; 1 \; 1))) \\
6
\end{align*}
\]

In Section 1.2 of [A&S], different approaches are proposed to recast the factorial procedure into another (equivalent) tail-recursive procedure. Recall a procedure is \textit{tail-recursive} if, whenever it calls itself, the result returned by the called function becomes immediately the result of the calling function. One such tail-recursive implementation is given at the top of page 34 in [A&S]; another one is given in footnote 29 on page 33. Another is the following:

\[
\text{(define (fact1 n)}
\quad (\text{fact-iter n 1}))
\]

\[
\text{(define (fact-iter n answer)}
\quad (\text{if} \; (= \; n \; 0)
\quad \text{answer}
\quad \text{answer}
\quad (\text{fact-iter} \; (- \; n \; 1) \; (* \; n \; \text{answer}))))
\]
The evaluation of any of these tail-recursive procedures exhibits a *linear iterative process*, in the sense explained in Section 1.2 of [A&S]. For example, the evaluation of the preceding procedure on input 3 produces the sequence:

\begin{verbatim}
(fact1 3)
(fact-iter 3 1)
(fact-iter 2 3)
(fact-iter 1 6)
(fact-iter 0 6)
6
\end{verbatim}

Another very different implementation of the factorial is written imperatively (i.e., with *side-effects*) and can be found on page 235 in [A&S]. This implementation is not strictly speaking “tail-recursive”, which is more appropriately reserved to qualify the form of functional (i.e., applicative) programs. Nevertheless, the evaluation of the imperative program on page 235 on a non-negative integer exhibits a *linear iterative process*, in the sense explained in the last paragraph on page 34.

Transforming a procedure into *continuation-passing style* (CPS) is another different way of obtaining a tail-recursive implementation. There is a vast literature on CPS, with many important applications to programming. The rest of this handout is restricted to a few examples and gives you enough of a general technique that will enable you to transform many common procedures into CPS.

Returning to the factorial procedure at the very beginning of this handout, and its evaluation on input 3, we can say that at every recursive call to `fact`, there is a *context* specifying what needs to be done to complete the evaluation. For example, the context

\begin{verbatim}
(* 3 (* 2 [ ] ))
\end{verbatim}

specifies what needs to be done to complete the evaluation after the call `(fact 1)` returns a value. We can call this context the *continuation* of `(fact 1)` in the evaluation of `(fact 3)`. Expressing this continuation in the language of Scheme, we can write:

\begin{verbatim}
(lambda (v) (* 3 (* 2 v)))
\end{verbatim}
If this Scheme expression is \( k \), i.e.,

\[
  k = (\text{lambda} \ (v) \ (* \ 3 \ (* \ 2 \ v)))
\]

then the application \((k \ (\text{fact} \ 1))\) produces

\[
  (* \ 3 \ (* \ 2 \ (\text{fact} \ 1)))
\]

Generalizing the preceding argument, suppose we run \( \text{fact} \) on input 20 and suppose the continuation of, say, \((\text{fact} \ 13)\) in the evaluation of \((\text{fact} \ 20)\) is \( k \). Then the continuation of the preceding recursive call to \( \text{fact} \), i.e. \((\text{fact} \ 12)\), is

\[
  (\text{lambda} \ (v) \ (k \ (* \ 13 \ v)))
\]

The preceding suggests that a CPS transformation of \( \text{fact} \) is

\[
\begin{align*}
  &\text{(define \ (fact-cps \ n \ k)} \\
  &\quad \text{(if \ (= \ n \ 0)} \\
  &\quad \quad (k \ 1)
  \\
  &\quad \quad \text{(fact-cps \ (- \ n \ 1) \ (lambda} \ (v) \ (k \ (* \ n \ v))))))
\end{align*}
\]

\( \text{fact-cps} \) is tail-recursive and its evaluation on input 3 and initial continuation \((\text{lambda} \ (v) \ v)\) produces:

\[
\begin{align*}
  &\text{(fact-cps} \ 3 \ (\text{lambda} \ (v) \ v)) \\
  &\text{(fact-cps} \ 2 \ (\text{lambda} \ (v) \ (* \ 3 \ v)) \\
  &\text{(fact-cps} \ 1 \ (\text{lambda} \ (v) \ (* \ 3 \ (* \ 2 \ v)))) \\
  &\text{(fact-cps} \ 0 \ (\text{lambda} \ (v) \ (* \ 3 \ (* \ 2 \ (* \ 1 \ v)))) \\
  &\text{((lambda} \ (v) \ (* \ 3 \ (* \ 2 \ (* \ 1 \ v)))) \ 1)
\end{align*}
\]

2

There are different ways of CPS transforming the same procedure! Here is another CPS transformation of \( \text{fact} \):

\[
\begin{align*}
  &\text{(define \ (fact-cps-1} \ n \ k) \\
  &\quad \text{(if \ (= \ n \ 0)} \\
  &\quad \quad (k \ 1)
  \\
  &\quad \quad \text{(fact-cps-1} \ (- \ n \ 1) \ (\text{lambda} \ (v) \ (* \ n \ (k \ v))))))
\end{align*}
\]

In the substitution model, we have:
(fact-cps-1 3 (lambda (v) v))
(fact-cps-1 2 (lambda (v) (* 3 v)))
(fact-cps-1 1 (lambda (v) (* 2 (* 3 v))))
(fact-cps-1 0 (lambda (v) (* 1 (* 2 (* 3 v)))))
(((lambda (v) (* 1 (* 2 (* 3 v)))) 1)
6
Another Easy Example

We used the procedure length several times in the past:

(define (length lst)
  (if (null? lst)
      0
      (+ (length (cdr lst)) 1)))

Based on what we did for the factorial function, here is a CPS-transformation of length:

(define (length lst) (length-cps lst (lambda (v) v)))

(define (length-cps lst k)
  (if (null? lst)
      (k 0)
      (length-cps (cdr lst) (lambda (v) (k (+ v 1))))))
Another Still Easy Example

There are different definitions of append. Here is one (see page 103 in [A&S]):

(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))))

Taking the two preceding examples as a guide, here is a CPS-transformation of append:

(define (append lst1 lst2)
  (append-cps lst1 lst2 (lambda (v) v)))

(define (append-cps lst1 lst2 k)
  (if (null? lst1)
      (k lst2)
      (append-cps (cdr lst1) lst2
                  (lambda (v) (k (cons (car lst1) v)))))
))
A Subtle Example

The reverse procedure takes a list as input argument and returns a list of the same elements in reverse order. Here is a possible definition in direct (as opposed to continuation-passing) style. (Two other interesting alternatives are proposed in Exercise 2.39, page 122 in [A&S].)

(define (reverse lst)
  (if (null? lst)
      '()
      (append (reverse (cdr lst)) (list (car lst)))))

Using the preceding examples as guide, a CPS-transformation of this procedure is:

(define (reverse lst) (reverse-cps lst (lambda (v) v)))

(define (reverse-cps lst k)
  (if (null? lst)
      (k '())
      (reverse-cps (cdr lst)
                   (lambda (v)
                        (k (append v (list (car lst))))))))

If you try the preceding procedure on a few examples, it works like magic, i.e., it works but it is difficult to understand why. Here is an alternative definition of reverse whose CPS-transformation is perhaps easier to understand. First, we define the procedure last-pair:

(define (last-pair lst)
  (if (null? (cdr lst))
      lst
      (last-pair (cdr lst))))

which works properly only if the input argument is a nonempty list. Note last-pair is already tail-recursive, so we do not bother to CPS-transform it. Second, we define the procedure remove-last-pair:
(define (remove-last-pair lst)
  (cond ((null? lst) '())
        ((null? (cdr lst)) '())
        (else (cons (car lst)
                     (remove-last-pair (cdr lst))))))

The CPS-transformation of remove-last-pair produces:

(define (remove-last-pair-cps lst k)
  (cond ((null? lst) (k '()))
        ((null? (cdr lst)) (k '()))
        (else (remove-last-pair-cps (cdr lst)
                                   (lambda (v) (k (cons (car lst) v)))))))

Third, we define the procedure reverse based on last-pair and remove-last-pair as follows:

(define (reverse lst)
  (if (null? lst)
      ()
      (cons (car (last-pair lst))
            (reverse (remove-last-pair lst)))))

We are ready to write a CPS-transformation of the preceding “direct-style” implementation of reverse, namely:
(define (reverse lst) (reverse-cps-1 lst (lambda (v) v)))

(define (reverse-cps-1 lst k)
  (if (null? lst)
      (k '())
      (reverse-cps-1
       (remove-last-pair-cps lst (lambda (v) v))
       (lambda (v)
          (k (cons (car (last-pair lst)) v))))))

Finally, thinking directly in terms of continuations (this is subtle!), here is one more CPS implementation of reverse:

(define (reverse lst) (reverse-cps-2 lst (lambda (v) v)))

(define (reverse-cps-2 lst k)
  (if (null? lst)
      (k '())
      (reverse-cps-2 (cdr lst)
       (lambda (v) (cons (car lst) (k v)))))))
Another Example: The Fibonacci Function

A standard definition of the fibonacci function is:

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

A CPS-transformation of fib is:

(define (fib-1 n) (fib-cps-1 n (lambda (v) v)))

(define (fib-cps-1 n k)
  (if (< n 2)
      (k 1)
      (fib-cps-1
        (- n 1)
        (lambda (v) (fib-cps-1
                      (- n 2)
                      (lambda (w) (k (+ v w))))))))

Here is another CPS transformation of fib. But is it really in tail-recursive form? Look at the second recursive call to fib-cps-2:

(define (fib-2 n) (fib-cps-2 n (lambda (v) v)))

(define (fib-cps-2 n k)
  (if (< n 2)
      (k 1)
      (fib-cps-2
        (- n 1)
        (lambda (v) (k (+ v (fib-cps-2
                            (- n 2)
                            (lambda (w) w)))))))))
Here is yet another CPS transformation of fib. And again, is it really in tail-recursive form?

(define (fib-3 n) (fib-cps-3 n (lambda (v) v)))

(define (fib-cps-3 n k)
  (if (< n 2)
    (k 1)
    (fib-cps-1
     (- n 1)
     (lambda (v) (k (+ v (fib-3 (- n 2))))))))