Continuation–passing style in ML [*]

Helmuth Trefftz, Ph.D.
Distance Education Instructor
Computer Science Department
Maharishi University of Management
htrefftz@eafit.edu.co

June, 2002

1 Introduction

This document describes how to implement iterations in continuation-passing style in ML. One example is implemented in the different styles that are presented

2 Recursivity

In imperative languages, iterations are accomplished using a *for*, *while* or similar instructions. In these instructions, there is usually a variable that keeps the state of the computation, either a counter, an accumulator a boolean flag or similar. As the computation advances, the value stored in the variable changes destructively and a test is performed on the variable to check if there is a need for further iteration.

In functional languages, iteration is accomplished through recursivity, that is, a function that invokes itself. Naturally, there must be a condition to determine when the function must not call itself anymore.

This document presents different styles of iterations, concluding with continuation-passing style, which will be revisited later in the course.

3 Example

The example we will use is a function that adds the numbers in a list. That is,

```
sum [1, 2, 3, 4, 5]
```

shall return 15.

[*]This document is based on the following web page by Dexter Kozen: http://www.cs.cornell.edu/Courses/cs312/2001FA/lecture/lec22.htm, .

3.1 Straightforward recursion

A simple way of solving the problem is to form an addition of the head of the list and the summation of the tail of the list, which is obtained by invoking the function on the tail of the list.

A simple implementation in ML would be:

```
fun sum1 [] = 0
 |  sum1 (x::t) = x + sum1 t;
```

Note that each time the function is invoked, a new operand is pushed into the stack and the sum operation is postponed until the recursive call returns.

The operations on our list are performed in the following order:

$(1 + (2 + (3 + (4 + (5 + 0)))))$

Note that operations are performed in a right-associative fashion.

3.2 Tail recursion

In tail recursion, operations are not deferred.

Look at the following implementation:

```
fun sum2 s =
  let
    fun sum [] a = a
     |  sum (x::t) a = sum t (x + a)
  in
    sum s 0
  end;
```

Note that, at each iteration, the head of the list is added to an accumulator. The accumulator contains the summation of the elements visited so far. The auxiliary function `sum` is invoked the first time with 0 as accumulator.

Operations are performed as follows:

$(((((1 + 0) + 2) + 3) + 4) + 5)$

Note that operations are performed in a left-associative way. Since + is an associative operator, both straightforward recursion and tail recursion produce correct results. This would not be the case of a non-associative operator, such as exponentiation.

3.3 Continuation–passing style

This concept is particularly elegant and is naturally suited to describe certain properties of languages, as we will see later on in the course.

See the following implementation:

```
fun sum3 s =
  let
    fun sum [] (k:int -> int) = k 0
    |  sum (x::t) k = sum t (fn a => k(x + a))
  in
    sum s (fn x => x)
  end;
```

In this case the operations performed at each iteration are combined as nested application of functions. At each iteration, a *continuation* `k` is received, modified, and passed down to the next iteration. The continuation contains the deferred operations of the previous iterations. Note that `a`, the parameter expected by the continuation, encapsulates the return value of all the *future* continuations. Generally, deferred operations are passed down to the next iteration, but they could also be passed on to anoher function to be performed.

The nested application of the continuation function can also be described as function combination, since `fn a => k(x + a)` is equivalent to `k o (fn a => x + a)`

In our example, operations take place like this:

```
(fn x => x) o (fn a => 1 + a) o (fn a => 2 + a) o ...
   o (fn a => 5 + a) 0
```

or

```
(1 + (2 + (3 + (4 + (5 + 0)))))
```

In the case of straightforward iteration, explained in section 3.1, pending operations are stored in the stack. In the case of continuation-passing style, the pending operations are stored as closures.

In order to convert a straightforward or tail-recursive recursive function to continuation-passing style, the following things need to be done:

- Add a continuation (`k` in our example) as a parameter to the recursive function .
- The continuation has to be applied to the final element that stops the recursive sequence of calls (`k 0` in our example)
- The *next* continuation, passed down to the next iteration (`(fn a => k(x + a))`) is defined as a function that combines the current element (`x`) with the result returned by the future continuations (`a`) inside a call to the *current* continuation received as parameter (`k`).
- The recursive function can be called the first time using the identity function (`fn x => x`)as continuation.