MP 4 – Higher-order Functions and Continuation-Passing Style

CS 421 – Fall 2014
Revision 1.1

Assigned September 16, 2014
Due September 23, 2014 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Removed the subsection Using Continuations to Alter Control Flow

2 Objectives and Background

The purpose of this MP is to:

- Help student master higher-order functions.
- Help the student learn the basics of continuation-passing style, or CPS, and CPS transformation. Next week, you will be using your knowledge learned from this MP to construct a general-purpose algorithm for transforming code in direct style into continuation-passing style.

3 Instructions

Instructions for solving the problems related to higher-order functions are the same as instruction for MP3.

The problems related to CPS transformation are all similar to the problems in MP2 and MP3. The difference is that you must implement each of these function in continuation-passing style. In some cases, you must first write a function in direct style (according to the problem specification), then transform the function definition into continuation-passing style.

The problems below have sample executions that suggest how to write answers. Students have to use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, the students are free to choose different names for the arguments to the functions from the ones given in the example execution. We also will use `let rec` to begin the definition of some of the functions that are allowed to use recursion. You are not required to start your code with `let rec`. Similarly, if you are not prohibited from using explicit recursion in a given problem, you may change any function definition from starting with just `let` to starting with `let rec`.

For all these problems, you are allowed to write your own auxiliary functions, either internally to the function being defined or externally as separate functions. All such helper functions must satisfy any coding restrictions (such as not using explicit recursion) as the main function being defined for the problem must satisfy.

Here is a list of the strict requirements for the assignment.

- The function name must be the same as the one provided.

- The type of parameters must be the same as the parameters shown in sample execution.
- Students must comply with any special restrictions for each problem. For several of the problems, you will be required to write a function in direct style, possibly with some restrictions, as you would have in MP2 or MP3, and then transform *the code **you** wrote* in continuation-passing style.

4 Problems

4.1 Higher Order Functions

For problems 1 through 4, you will be supplying arguments to the higher-order functions `List.map`, `List.fold_right`, and `List.fold_left`. You should not need to use explicit recursion for any of these problems.

1. (4 pts) Write a function `pair_sums_map : (int * int) list -> int list` that computes the same results as defined in Problem 9 from MP3. There should be no use of recursion or library functions except `List.map` in the solution to this problem

```
let pair_sums_map l = ...;
val pair_sums_map : (int * int) list -> int list = <fun>
# pair_sums_map [(1,6);(3,1);(3,2)];;
- : int list = [7;4;5]
```

2. (5 pts) Write a value `odd_sum_base : int` and function `odd_sum_rec : (int -> int -> int` such that `(fun l -> List.fold_right odd_sum_rec l odd_sum_base)` computes the same solution as `odd_sum` defined in Problem 8 from MP3. There should be no use of recursion or library functions in the solution to this problem.

```
# let odd_sum_base = ...;;
val odd_sum_base : int = ...
# let odd_sum_rec = ...;;
val odd_sum_rec : int -> int -> int = <fun>
# let odd_sum l = List.fold_right odd_sum_rec l odd_sum_base;;
val odd_sum : int list -> int = <fun>
# odd_sum [1;2;3];;
- : int = 4
```

3. (5 pts) Write a value `count_element_base : int` and function `count_element_rec : 'a -> int -> 'a -> int` such that `(fun l -> fun m -> List.fold_left (count_element_rec m) count_element_base l)` computes the same results as `count_element` defined in Problem 10 from MP3. There should be no use of recursion or library functions in the solution to this problem.

```
# let count_element_base = ...;;
val count_element_base : int = ...
# let count_element_rec = ...;;
val count_element_rec : 'a -> int -> 'a -> int = <fun>
# let count_element l m = List.fold_left (count_element_rec m) count_element_base l;;
val count_element : 'a list -> 'a -> int = <fun>
# count_element [0;1;3;2;1;1;3] 3;;
- : int = 2
```

4.2 Higher Order Functions - Extra Credit

4. (3 pts) Write a function `apply_even_odd : 'a list -> ('a -> 'b) -> ('a -> 'b) -> 'b list` such that `apply_even_odd [x0; x1; x2; x3; ...] f g` returns a list `[f x0; g x1; f x2; g x3; ...]`. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec apply_even_odd l f g = ...;;
val apply_even_odd : 'a list -> ('a -> 'b) -> ('a -> 'b) -> 'b list = <fun>
# apply_even_odd [1;2;3] (fun x -> x+1) (fun x -> x - 1);;
- : int list = [2; 1; 4];;
```

5 Continuation Passing Style

These exercises are designed to give you a feel for continuation-passing style. A function that is written in continuation-passing style does not return once it has finished computing. Instead, it calls another function (the continuation) with the result of the computation. Here is a small example:

```
# let report x =
   print_string "Result: ";
   print_int x;
   print_newline();;
val report : int -> unit = <fun>

# let inck i k = k (i+1)
val inck : int -> (int -> 'a) -> 'a = <fun>
```

The `inck` function takes an integer and a continuation. After adding 1 to the integer, it passes the result to its continuation.

```
# inck 3 report;;
Result: 4
- : unit = ()
# inck 3 inck report;;
Result: 5
- : unit = ()
```

In line 1, `inck` increments 3 to be 4, and then passes the 4 to `report`. In line 4, the first `inck` adds 1 to 3, and passes the resulting 4 to the second `inck`, which then adds 1 to 4, and passes the resulting 5 to `report`.

5.1 Transforming Primitive Operations

Primitive operations are "transformed" into functions that take the arguments of the original operation and a continuation, and apply the continuation to the result of applying the primitive operation on its arguments.

In the helper module `Mp4common`, we have given you a testing continuation and a few low-level functions in continuation-passing style. These are as follows:

```
val report : int -> unit = <fun>
val addk : int -> int -> (int -> 'a) -> 'a = <fun>
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
val float_addk : float -> float -> (float -> 'a) -> 'a = <fun>
val float_divk : float -> float -> (float -> 'a) -> 'a = <fun>
val pairk : 'a -> 'b -> ('a * 'b -> 'c) -> 'c = <fun>
```

You are being asked first to extend that set of functions in continuation-passing style.

5. **(8pts)** Write the following low-level functions in continuation-passing style. A description of what each function should do follows:

   - `divk` divides the first integer by the second;
   - `modk` divides the first integer by the second and returns the remainder;
   - `float_subk` subtracts the second float from the first;
   - `float_mulk` multiplies two floats;
   - `catk` concatenates two strings;
   - `consk` creates a new list by adding an element at the front of a list;
   - `leqk` determines if the first argument is less than or equal to the second argument; and
   - `eqk` determines if the two arguments are equal.

```
# let divk n m k = ...;;
val divk : int -> int -> (int -> 'a) -> 'a = <fun>
# let modk n m k = ...;;
val modk : int -> int -> (int -> 'a) -> 'a = <fun>
# let float_subk a b k = ...;;
val float_subk : float -> float -> (float -> 'a) -> 'a = <fun>
# let float_mulk a b k = ...;;
val float_mulk : float -> float -> (float -> 'a) -> 'a = <fun>
# let catk str1 str2 k = ...;;
val catk : string -> string -> (string -> 'a) -> 'a = <fun>
# let consk e l k = ...;;
val consk : 'a -> 'a list -> ('a list -> 'b) -> 'b = <fun>
# let leqk x y k = ...;;
val leqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# let eqk x y k = ...;;
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>

# divk 12 5 report;;
Result: 2
- : unit = ()
# catk "hello " "world" (fun x -> x);;
- : string = "hello world"
# float_subk 3.0 1.0
    (fun x -> float_mulk x 2.0
        (fun y -> (print_string "Result:"; print_float y; print_newline())));;
    Result:4.
- : unit = ()
# leqk 2 1 (fun b -> (report (if b then 1 else 0)));;
Result: 0
- : unit = ()
```

5.2 Nesting Continuations

```
# let add3k a b c k =
    addk a b (fun ab -> addk ab c k);;
```

```
val add3k : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# add3k 1 2 3 report;;
Result: 6
- : unit = ()
```

We needed to add three numbers together, but `addk` itself only adds two numbers. On line 2, we give the first call to addk a function that saves the sum of `a` and `b` in the variable `ab`. Then this function adds `ab` to `c` and passes its result to the continuation k.

6. **(5 pts)** Using `addk` (defined in `MP4common` and the lectures notes) and `mulk` (defined above) as helper functions, write a function `polyk`, which takes on integer argument x and "returns" $x^4 + x + 1$. You may only use the `addk` and `mulk` operators to do the arithmetic. The order of evaluation of operations must be as follows: first compute x^2, then using its result with one more multiplication, compute x^4. Next, compute $x + 1$, and finally compute $x^4 + x + 1$.

```
# let poly x k = ...;;
val poly : int -> (int -> 'a) -> 'a = <fun>
# poly 2 report;;
Result: 19
- : unit = ()
```

7. **(8 pts)** Write a function `distributek` that takes, as the first two arguments, two functions f and g, and, as third and fourth arguments, values x and y, and "returns" $g(f(x))(f(y))$ (distributing application of f over x and y, before applying g to the results). The order of computation is $f(x)$, then $f(y)$, then $g(f(x))(f(y))$ (g takes three arguments, where the last argument is a continuation). You must write `distributek` in continuation-passing style and you must assume that the functions f and g are given in the continuation-passing style.

```
let distributek f g x y k = ...
# val distributek :
  ('a -> ('b -> 'c) -> 'c) ->
  ('b -> 'b -> ('d -> 'e) -> 'c) -> 'a -> 'a -> ('d -> 'e) -> 'c = <fun>
# distributek inck addk 1 2 (fun x -> x);;
- : int = 5
```

5.3 Transforming Recursive Functions

How do we write recursive programs in CPS? Consider the following recursive function:

```
# let rec factorial n =
if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

We can rewrite this making each step of computation explicit as follows:

```
# let rec factoriale n =
      let b = n = 0 in
          if b then 1
          else let s = n - 1 in
```

```
               let m = factoriale s in
               n * m;;
val factoriale : int -> int = <fun>
# factoriale 5;;
- : int = 120
```

Now, to put the function into full CPS, we must make factorial take an additional argument, a continuation, to which the result of the factorial function should be passed. When the recursive call is made to factorial, instead of it returning a result to build the next higher factorial, it needs to take a continuation for building that next value from its result. In addition, each intermediate computation must be converted so that it also takes a continuation. Thus the code becomes:

```
# let rec factorialk n k =
      eqk n 0
      (fun b -> if b then k 1
                else subk n 1
                    (fun s -> factorialk s
                              (fun m  -> timesk n m k)));;
# factorialk 5 report;;
Result: 120
- : unit = ()
```

Notice that to make a recursive call, we needed to build an intermediate continuation capturing all the work that must be done after the recursive call returns and before we can return the final result. If `m` is the result of the recursive call in direct style (without continuations), then we need to build a continuation to:

- take the recursive value: `m`
- build it to the final result: `n * m`
- pass it to the final continuation `k`

Notice that this is an extension of the "nested continuation" method.

In Problems 8 through 10 you are asked to first write a function in direct style and then transform the code into continuation-passing style. When writing functions in continuation-passing style, all uses of functions need to take a continuation as an argument. For example, if a problem asks you to write a function `partition`, then you should define `partition` in direct style and `partitionk` in continuation-passing style. All uses of primitive operations (*e.g.* +, −, *, <=, <>) should use the corresponding functions defined in Problem 5.1 or in the lecture notes. If you need to make use of primitive operations not covered in Problem 5.1, you should include a definition of the corresponding version that takes a continuation as an additional argument, as in Problem 5.1. In Problem 9 and 10, there must be no use of list library functions.

8. **(6 pts total)**

   a. **(2 pts)** Write a function `alternate_series`, which takes an integer n, and computes the (partial) series $-1 + 2 - 3 + \ldots + (-1)^n \cdot n$ and returns the result. If $n \leq 0$, then return 0.

   ```
   # let rec alternate_series n = ...;;
   val alternate_series : int -> int = <fun>
   # alternate_series 10;;
   - : int = 5
   ```

   b. **(4 pts)** Write the function `alternate_seriesk` which is the CPS transformation of the code you wrote in part a.

```
# let rec alternate_seriesk n k = ...;;
val alternate_seriesk : int -> (int -> 'a) -> 'a = <fun>
# alternate_seriesk 10 (fun x -> x);;
- : int = 5
```

9. **(8 pts total)**

   a. **(2 pts)** Write a function `rev_iter` which takes a function f (of type `'a -> unit`) and a list l (of type `'a list`). If the list l has the form $[a_1; a_2; \ldots; a_n]$, then `rev_iter` applies f on a_n, then on a_{n-1}, then ..., then on a_1. There must be no use of list library functions.

```
# let rec rev_iter f l = ...;;
val rev_iter : ('a -> unit) -> 'a list -> unit = <fun>
# rev_iter (fun x -> print_int x) [1;2;3;4;5];;
54321- : unit = ()
```

   b. **(6 pts)** Write the function `rev_iterk` that is the CPS transformation of the code you wrote in part a. You must assume that the function f is also transformed in continuation-passing style, that is, the type of f is not `'a -> unit`, but `'a -> (unit -> 'b) -> 'b`.

```
# let rec rev_iterk f l k = ...;;
val rev_iterk : ('a -> (unit -> 'b) -> 'b) -> 'a list
                                            -> (unit -> 'b) -> 'b = <fun>
# let print_intk i k = k (print_int i);;
val print_intk : int -> (unit -> a) -> a = <fun>
# rev_iterk (fun x -> fun k ->
                      print_intk x (fun t -> k t)) [1; 2; 3; 4; 5] (fun x -> x);;
54321- : unit = ()
```

10. **(8 pts total)**

   a. **(2 pts)** Write a function `filter` which takes a list `l` (of type `'a list`), and a predicate `p` (of type `'a -> bool`), and returns the list containing all the elements satisfying p. The order of the elements in the returning list must correspond to the order in l. There must be no use of list library functions.

```
# let rec filter l p = ...;;
val filter : 'a list -> ('a -> bool) -> 'a list = <fun>
# filter [1; 2; 3; 4] (fun x -> x >= 2);;
- : int list = [2; fi3; 4]
```

   b. **(6 pts)** Write a function `filterk` which is the CPS transformation of the code you wrote in part a. You must assume that the predicate p is also transformed in continuation-passing style, that is, its type is not `'a -> bool`, but `'a -> (bool -> 'b) -> 'b`.

```
# let rec let rec filterk l p k = ...;;
val filterk : 'a list -> ('a -> (bool -> 'b) -> 'b)
                             -> ('a list -> 'b) -> 'b = <fun>
# filterk [1; 2; 3; 4] (fun x -> fun k -> leqk 2 x k) (fun x -> x);;
- : int list = [2; 3; 4]
```

5.4 CPS - Extra Credit

11. **(8 pts)**

Write the function `appk` which takes a list l of functions in continuation-passing style (of type `'a -> ('a -> 'b) -> 'b`), an initial value x (of type `'a`) and a continuation k. If the list l is of the form $[f_1; \ldots ; f_n]$, then `appk` evaluates $f_1\ (f_2\ (\ldots (f_n\ x) \ldots))$ and passes the result to k. Intuitively, it evaluates f_n on x, then f_{n-1} on the result, then f_{n-2} on the second result, and so on. Your definition must be in continuation-passing style.

```
# let rec appk l x k = ...;
val appk : ('a -> ('a -> 'b) -> 'b) list -> 'a -> ('a -> 'b) -> 'b = <fun>
# appk [inck;inck;inck] 0 (fun x -> x);;
- : int = 3
```