ML2 – Higher-Order Functions and Continuation-Passing Style

CS 421 – Fall 2015

Revision 1.1

Assigned September 15, 2015
Due September 23, 2015 – September 25, 2015
Extension None past the allowed lab sign-up time

1 Change Log

1.1 Clarified wording of the following questions:

- Problem 9: Changed `str_concat` to `concat`.
- Problem 13: Added note that `List.map` may not be used.

1.0 Initial Release.

2 Objectives and Background

The purpose of this ML is to:

- Help the student master higher-order functions.
- Help the student learn the basics of continuation-passing style (CPS) and CPS transformation.

You will be using your knowledge learned from this ML to construct a general-purpose algorithm for transforming code in direct style into continuation-passing style.

3 Instructions

The problems related to higher-order functions ask you to rewrite programs you first encountered on MP2, but this time you will be using higher-order operators `List.fold_right`, `List.fold_left` and `List.map` instead of explicit recursion.

The problems related to CPS transformation are all similar to problems you've seen in ML1 and MP2. The difference is that you must implement each of these functions in continuation-passing style. The problems below have sample executions that suggest how to write your answers. You will have to use the same function name, but the name of the parameters that follow the function name need not be duplicated. That is, you 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 or using CPS) 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 the function must be the same as specified in the problem.
- You must comply with any special restrictions for each problem.

4 Problems

4.1 Higher Order Functions

For Problems 1 through 4, you will be supplying arguments to the higher-order functions `List.fold_right`, `List.fold_left`, and `List.map`. You are not allowed to use any other library functions or any direct use of recursion for any of these problems.

1. (5 pts) Write a base value `split_base` and a step function `split_step` such that `List.fold_right (split_step f) lst split_base` returns a pair of lists where the first list of the pair contains every element `x` of `lst` for which `(f x)` is true, and the second list of the pair contains every element for which `(f x)` is false. The order of the elements in the returned lists should be the same as in the original list. You are not allowed to use any direct use of recursion or any other library functions.

```
# let split_base = ... ;;
val split_base : 'a list * 'b list = ...
# let split_step f x (true_xs, false_xs) = ... ;;
val split_step : ('a -> bool) -> 'a -> 'a list * 'a list -> 'a list * 'a list =
  <fun>

# List.fold_right (split_step (fun x -> x > 2)) [0;2;3;5;1;4] split_base;;
- : int list * int list = ([3; 5; 4], [0; 2; 1])
```

2. (5 pts) Recall that *run-length encoding* (RLE) is a data compression technique in which maximal (non-empty) consecutive occurrences of a value are replaced by a pair consisting of the value and a counter showing how many times the value was repeated in that consecutive sequence.

   For example, RLE would encode the list `[1;1;1;2;2;2;3;1;1;1]` as: `[(1,3);(2,3);(3,1);(1;3)]`.

   Write a function `rle : 'a list -> ('a * int) list` that, when provided with a list `lst`, encodes `lst` using the RLE technique. The definition of `rle` may use `List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` but no direct use of recursion, and no other library functions.

```
# let rle lst = ... ;;
val rle : 'a list -> ('a * int) list = <fun>
# rle [1;1;1;2;2;2;3;1;1;1];;
- : (int * int) list = [(1, 3); (2, 3); (3, 1); (1, 3)]
```

3. (7 pts) Write a function `concat : string -> string list -> string` that, given a string `s` and a list of strings `list`, creates a string consisting of the strings in `list` concatenated together, with the first string `s` inserted between. If the list is empty, you should return the empty string (`""`). If the list is a singleton, you should return just the single string in the list. The definition of `concat` may use `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` but no direct use of recursion, and no other library functions.

```
# let concat s list = ... ;;
val concat : string -> string list -> string = <fun>
# concat " * " ["3"; "6"; "2"];;
- : string = "3 * 6 * 2"
```

4. (8 pts) Write a function `app_all : ('a -> 'b) list -> 'a list -> 'b list list` that takes a list of functions, and a list of arguments for those functions, and returns the list of list of results from consecutively applying the functions to all arguments, in the order in which the functions occur in the list and in the order in which the arguments occur in the list. Each list in the result list corresponds to a list of applications of each function to the given arguments. The definition of `app_all` may use the library function `List.map : ('a -> 'b) -> 'a list -> 'b list` but no direct use of recursion, and no other library functions.

```
# let app_all fs list = ... ;;
val app_all : ('a -> 'b) list -> 'a list -> 'b list list = <fun>
# app_all [(fun x -> x > 0); (fun y -> y mod 2 = 0);
  (fun x -> x * x = x)] [1; 3; 6];;
- : bool list list =
[[true; false; true]; [true; false; false]; [true; true; false]]
```

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_int x =
    print_string "Result: ";
    print_int x;
    print_newline();;
val report_int : 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_int;;
Result: 4
- : unit = ()
# inck 3 inck report_int;;
Result: 5
- : unit = ()
```

In the first example, `inck` increments 3 to be 4, and then passes the 4 to `report_int`. In the second example, 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_int`.

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 `Ml2common`, we have given you a testing continuation and a few low-level functions in continuation-passing style. These are as follows:

```
val report_int : int -> unit = <fun>
val report_float : float -> unit = <fun>

val addk : int -> int -> (int -> 'a) -> 'a = <fun>
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
val mulk : int -> int -> (int -> 'a) -> 'a = <fun>
val modk : int -> int -> (int -> 'a) -> 'a = <fun>
val float_addk : float -> float -> (float -> 'a) -> 'a = <fun>
val float_subk : float -> float -> (float -> 'a) -> 'a = <fun>
val float_mulk : float -> float -> (float -> 'a) -> 'a = <fun>
val geqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
val notk : bool -> (bool -> '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 as a warmup exercise. Note that "misuse" of continuation-passing style (e.g. things like `k ((a + b) - c)`) may cost you up to full credit for the problem.

5. (0 pts) This problem will not be given for you to complete in the CBTF. Instead, we will be providing you with a correct implementation of these functions in `Ml2common` while you are in the CBTF.

   Write the following low-level functions in continuation-passing style. A description of what each function should do follows:

   - `consk` creates a new list by adding an element to the front of a list.
   - `concatk` concatenates two strings in the order they are provided.
   - `string_of_intk` takes an integer and converts it into a string.
   - `ceilk` takes a float, rounds it up to the nearest integer, and returns it as a float.

```
# let consk x l k = ... ;;
val consk : 'a -> 'a list -> ('a list -> 'b) -> 'b = <fun>
# let concatk s1 s2 k = ... ;;
val concatk : string -> string -> (string -> 'a) -> 'a = <fun>
# let string_of_intk n k = ... ;;
val string_of_intk : int -> (string -> 'a) -> 'a = <fun>
# let ceilk n k = ... ;;
val ceilk : float -> (float -> 'a) -> 'a = <fun>

# consk 1 [] print_int_list;;
[1]- : unit = ()
# concatk "hello" "world" print_string;;
helloworld- : unit = ()
# string_of_intk 0 print_string;;
0- : unit = ()
```

```
# ceilk 3.14 print_float;;
4.- : unit = ()
```

The next problems will be among those given in the CBTF.

6. (5 pts) Using addk, mulk, and modk, all defined in Ml2common, write a function add_mul_modk that takes three integer arguments, x, y, and z, and "returns" the expression $((x + y) * y) \bmod z$. You may only use addk, mulk, and modk to do the arithmetic.

```
# let add_mul_modk x y z k = ... ;;
val add_mul_modk : int -> int -> int -> (int -> 'a) -> 'a = <fun>

# add_mul_modk 3 3 3 report_int;;
Result: 0
- : unit = ()
```

7. (5 pts) Write a function parabolak that takes, as the first three arguments, the coefficients to the expression $(ax + b)^2 + c$ (namely, a, b, and c), a fourth argument x, and computes the result of the expression $(ax + b)^2 + c$.

```
# let parabolak a b c x k = ... ;;
val parabolak : int -> int -> int -> int -> (int -> 'a) -> 'a = <fun>

# parabolak 1 0 0 0 report_int;;
Result: 0
- : unit = ()
```

8. (5 pts) Write a function keep_the_changek that takes two floats, s and t, and (much like what happens when when you go make purchases with some debit cards), does the following computation: $ceil(s+(s*t))-(s+(s*t))$. The value of $s + (s * t)$ should only be computed once.

```
# let keep_the_changek s t k = ... ;;
val keep_the_changek : float -> float -> (float -> 'a) -> 'a = <fun>

keep_the_changek 3.00 0.50 report_float;;
Result: 0.5
- : unit = ()
```

9. (5 pts) Write a function usernamek that takes an integer n and two strings, x and s, and returns a string where the parts are concatenated as follows: $xsnx$. The order of computation should be as follows: first concatenate x and s, then convert n into a string. Following that, concatenate the initial string (xs) and n. Finally, concatenate the resulting string with x. You may only use the methods concatk and string_of_intk from Problem 5.

```
# let usernamek n x s k =
val usernamek : int -> string -> string -> (string -> 'a) -> 'a = <fun>

# usernamek 47 "xx" "HelloKitty" (fun s -> (s,52));;
- : string * int = ("xxHelloKitty47xx", 52)
```

5.2 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_int;;
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 10 through 13 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 earlier in this ML or in the lecture notes.

If you need to make use of primitive operations not covered in Problem 5, you should include a definition of the corresponding version that takes a continuation as an additional argument, as in Problem 5.

You may not use library functions for any of Problems 11 through 15, and it is assumed that any parameters which are functions (or predicates) that are arguments to the CPS versions of the functions will be in continuation-passing style.

10. (8 pts total)

    a. (2 pts) Write a function `even_odd` that takes a list of integers l and splits it into a pair of lists such that the left list contains all even numbers and the right list contains all odd numbers, in order of how they appear in l.

    b. (6 pts) Write the function `even_oddk` that is the CPS transformation of the code you wrote for Part (a).

```
# let rec even_odd l = ... ;;
val even_odd : int list -> int list * int list = <fun>
# let rec even_oddk l k = ... ;;
val even_oddk :  int list -> (int list * int list -> 'a) -> 'a = <fun>

# even_odd [-1;0;-1];;
- : int list * int list = ([0], [-1; -1])
# even_oddk [-1;0;-1] (fun (e,o) -> e @ o);;
- : int list = [0; -1; -1]
```

11. (8 pts)

    a. (2 pts) Write a function `enlike` that takes a list of strings l and, for every string s in l, prepends s with the string, "`like, `".

    b. (6 pts) Write the function `enlikek` that is the CPS transformation of the code you wrote for Part (a).

```
# let rec enlike l = ... ;;
val enlike : string list -> string list = <fun>
# let rec enlikek l k = ... ;;
val enlikek : string list -> (string list -> 'a) -> 'b list = <fun>

# enlike ["totally"; "that's so fetch"];;
- : string list = ["like, totally"; "like, that's so fetch"]
# enlikek ["totally"; "that's so fetch"] (fun x -> x);;
- : string list = ["like, totally"; "like, that's so fetch"]
```

12. (8 pts)

   a. (2 pts) Write a function `low_five` that takes a list of integers l and for every element x in l, if x is less than 5, replaces x with the result of multiplying x by 5.

   b. (6 pts) Write the function `low_fivek` that is the CPS transformation of the code you wrote for Part (a).

```
# let rec low_five l = ... ;;
val low_five : int list -> int list = <fun>
# let rec low_fivek l k = ... ;;
val low_fivek : int list -> (int list -> 'a) -> 'b list = <fun>

# low_five [1;3;5;7;9];;
- : int list = [5; 15; 5; 7; 9]
# low_fivek [1;3;5;7;9] (fun x -> x);;
- : int list = [5; 15; 5; 7; 9]
```

13. (8 pts)

   a. (1 pt) Write a function `map` that takes a function f and a list l and applies f to every element in l. You may not use `List.map` in your solution.

   b. (7 pts) Write the function `mapk` that is the CPS transformation of the code you wrote for Part (a). You must assume that the function f is also transformed into continuation-passing style.

```
# let rec map f l = ... ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let rec mapk f l k = ... ;;
val mapk : ('a -> ('b -> 'c) -> 'c) -> 'a list -> ('b list -> 'c) -> 'c =

# map (fun x -> x + 1) [0;1;2];;
- : int list = [1; 2; 3]
# mapk (addk 1) [0;1;2] (fun x -> (x,List.length x));;
- : int list = ([1; 2; 3], 3)
```

5.3 Extra Credit

14. (4 pts) For two lists L_1 and L_2, L_2 is called a *sub-list* of L_1 if: (a) all the elements of L_2 occur in L_1, and (b) their order in L_1 is *exactly* the same as their order in L_2. Write a function `sub_list : 'a list -> 'a list -> bool` that takes two lists as input and determines whether the second list is a sub-list of the first one. The definition of `sub_list` may use `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` but no direct use of recursion, and no other library functions.

```
# let sub_list l1 l2 = ... ;;
val sub_list : 'a list -> 'a list -> bool = <fun>
# sub_list [1;1;2;1;1;4;1] [1;2;1;1;1];;
- : bool = true
```

Similar to Problems 10 through 13, all uses of primitive functions in questions requiring you to define a function in continuation-passing style should use the corresponding functions defined earlier in this ML or in the lecture notes.

If you need to make use of primitive operations not covered in Problem 5, you should include a definition of the corresponding version that takes a continuation as an adidtional argument, as in Problem 5.

You may not use library functions for any of Problems 15 through 17, and it is assumed that any parameters which are functions (or predicates) that are arguments to the CPS versions of the functions will be in continuation-passing style.

15. (4 pts) Write a function `distributek` that takes as its first three parameters functions f, g, and h, its second three parameters integers x, y, and z, and does the following computation: multiply the result of applying f to x with that of g to y, then apply h to the previous result and z. You **must** follow this order of computation: First, compute the application of f to x, then that of g to y. Next, compute the product of the two results, and then finally, apply h to that result and z, in that order.

```
# let distributek f g h x y z k = ... ;;
val distributek :
  ('a -> (int -> 'b) -> 'c) ->
  ('d -> (int -> 'e) -> 'b) ->
  (int -> 'f -> 'g -> 'e) -> 'a -> 'd -> 'f -> 'g -> 'c = <fun>

# distributek (addk 1) (addk 1) addk 1 2 3 report_int;;
Result: 9
- : unit = ()
```

16. (5 pts)

a. Write a function `map_split_pair` that takes a predicate f, a function g, and a list of integers l, and returns a pair of lists where the left list of the pair is a list of elements with g applied to them (in order of appearance in l) where f applied to x returns true, and the right list a list of elements with g applied to them (in order of appearance in l) where f applied to x returns false.

b. Write a function `map_split_pairk` that is the CPS transformation of the code you wrote in Part (a). You must assume that both f and g are also transformed into continuation-passing style.

```
# let rec map_split_pair f g l = ... ;;
val map_split_pair :
  ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list * 'b list = <fun>
# map_split_pair (fun x -> x >= 5) (fun x -> x + 1) [2;4;6;8];;
- : int list * int list = ([7; 9], [3; 5])

# let rec map_split_pairk f g l k = ... ;;
val map_split_pairk :
  ('a -> ('b -> 'b) -> bool) ->
  ('a -> ('c -> 'd) -> 'd) -> 'a list -> ('c list * 'c list -> 'd) -> 'd =
  <fun>
# map_split_pairk (fun x -> geqk x 5) inck [2;4;6;8] (fun (x,y) -> x @ y);;
- : int list = [7; 9; 3; 5]
```

17. (5 pts)

a. Write a function `transform_if` that takes a predicate f, a function g, and a list l, and for every element x in l, if f applied to x returns true, replaces x with the result of applying g to x.

b. Write a function `transform_ifk` that is the CPS transformation of the code you have written for Part (a). You must assume that the predicate f and the function g are also transformed into continuation-passing style.

```
# let rec transform_if f g l = ... ;;
val transform_if : ('a -> bool) -> ('a -> 'a) -> 'a list -> 'a list = <fun>
# transform_if (fun x -> x >= 2) (fun x -> x * 2) [1;2;3];;

# let rec transform_ifk f g l k = ... ;;
val transform_ifk :
  ('a -> (bool -> bool) -> bool) ->
  ('a -> ('a -> 'b) -> 'b) -> 'a list -> ('a list -> 'b) -> 'b = <fun>
# transform_ifk (fun x -> geqk x 2) (fun x -> mulk x 2) [1;2;3] (fun x -> x);;
- : int list = [1; 4; 6]
```