MP 2 – Continuation-Passing Style

CS 421 – Fall 2017
Revision 1.0

Assigned September 21, 2017
Due September 28, 2017 23:59pm
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to:

- 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

The problems related to CPS transformation are all similar to the problems in MP1 and ML2. The difference is that you must implement each of these functions 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 ***you*** *wrote* 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.

For this assignment only, you are not allowed to use external helper funcitons except those provided in the `Common` module and those you write in Problem 1.

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

- The function name and type 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 MP1 or ML2, and then transform *the code* ***you*** *wrote* in continuation-passing style.
- No helper functions should be used on this assignment beyond those in `Common` module and those you define in Problem 1.

4 Continuation Passing Style Problems

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 (fun n -> inck n 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`.

4.1 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 `Common`, 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 : int * int -> (bool -> 'a) -> 'a = <fun>
val leqk : int * int -> (bool -> 'a) -> 'a = <fun>
val ltk : int * int -> (bool -> 'a) -> 'a = <fun>
val gtk : int * int -> (bool -> 'a) -> 'a = <fun>
val eqk : int * int -> (bool -> 'a) -> 'a = <fun>
val neqk : int * int -> (bool -> 'a) -> 'a = <fun>
val notk : bool * (bool -> 'a) -> 'a = <fun>
```

We have artifically restricted the types of the comparison operators to being applied to integers to facilitate testing. You are being asked first to extend that set of functions in continuation-passing style.

1. (0 pts) 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.
   - `truncatek` takes a float n and truncates it in the same way as the `truncate` function which can be found in the pervasives module.

```
# 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 truncatek x k = ... ;;
val truncatek : float -> (int -> 'a) -> 'a = <fun>

# consk (1, []) (List.map string_of_int);;
- : string list = ["1"]
# concatk ("hello", "world") (fun s -> (s, String.length s));;
- : string * int = ("helloworld", 10)
# string_of_intk 0 (fun s -> (s, String.length s));;
- : string * int = ("0", 1)
# truncatek 3.14 string_of_int;;
- : string = "3"
```

2. (5 pts) Using `subk` and `mulk` defined in `Common`, write a function `diff_flipk` that takes one integer argument `p` and "returns" the expression $2 * ((1 - p) * p)$. You may only use `subk` and `mulk` to do the arithmetic.

```
# let diff_flipk p k = ... ;;
val diff_flipk : int -> (int -> 'a) -> 'a = <fun>

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

3. (5 pts) Write a function `quadk` that takes three integer arguments, `a`, `b`, and `c`, and "returns" the result of the expression $(2 * (a^2) + 4 * b) + c$. Again you may only use functions from `Common` to perform the arithmetic.

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

# quadk (1, 1, 1) report_int;;
Result: 7
- : unit = ()
```

4. (5 pts) Write a function `three_freezek` that takes two string arguments `s` and `p` and calculates the string formed by concatenating them as sp. The function will then "return" this string repeated three times in a row, however you should only need to calculate sp once.

```
# let three_freezek (s, p) k = ... ;;
val three_freezek : string * string -> (string -> 'a) -> 'a = <fun>

# three_freezek ("muda", "plop") (fun s -> (s , String.length s));;
- : string * int = ("mudaplopmudaplopmudaplop", 24)
```

5. (5 pts) Write a function `shiftk` that takes a string argument `s` and a float argument `q`. This function will calculate $(q+1.57)^2$ using only arithmetic functions from `Common`. After calculating this value, it should truncate the result, turn it into a string, and then concatenate the string `s` to the beginning and end of the resulting string. This string is then "returned".

```
# let shiftk (s, q) k = ... ;;
val shiftk : string * float -> (string -> 'a) -> 'a = <fun>

# shiftk ("##", 3.14) (fun s -> s);;
- : string = "##22##"
```

4.2 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;;
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 6 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 above in Section 4.1 or in the lecture notes. If you need to make use of primitive operations not covered above, you should include a definition of the corresponding version that takes a continuation as an additional argument. In all problems there must be no use of list library functions.

6. (7 pts total)

   a. (2 pts) Write a function `list_prod : int list -> int` that returns the product of all the elements in an input list, if the list is non-empty and `1` if the list is empty. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

   b. (5 pts) Write the function `list_prodk : int list -> (int -> 'a) -> 'a` that is the CPS transformation of **the function you gave in part a**.

```
# let rec list_prod l = ...
val list_prod : int list -> int = <fun>
# list_prod [1;2;3];;
- : int = 6
let rec list_prodk l k = ...
val list_prodk : int list -> (int -> 'a) -> 'a = <fun>
# list_prodk [1;2;3] report_int;;
Result: 6
- : unit = ()
```

7. (7 pts total)

   a. (2 pts) Write a function `all_positive : int list -> bool` that returns `true` if all the elements in the list are positive, and `false` otherwise. The function is required to use (only) tail recursion (no other form of recursion). You may not use any library functions. This problem does **not** require an auxiliary function.

b. (5 pts) Write the function `all_positivek : int list -> (bool -> 'a) -> 'a` that is the CPS transformation of **the function you gave in part a**.

```
# let rec all_positive l = ...
val all_positive : int list -> bool = <fun>
all_positive [5;3;6;(-1);7];;
- : bool = false
# let rec all_positivek l k = ...
val all_positivek : int list -> (bool -> 'a) -> 'a = <fun>
# all_positivek [5;3;6;(-1);7] (fun b -> if b then "true" else "false");;
- : string = "false"
```

8. (8 pts total)

   a. (2 pts) Write a function `even_count` that takes a list of integers and computes the number of even integers found in the input list as in ML2. The only form of recursion you may use for this problem is forward recursion. You may not use library functions in this part.

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

```
# let rec even_count l = ... ;;
val even_count : int list -> int = <fun>
# even_count [1;2;3];;
- : int = 1
# let rec even_countk l k = ... ;;
val even_countk : int list -> (int -> 'a) -> 'a = <fun>
# even_countk [1;2;3] report_int;;
Result: 1
- : unit = ()
```

4.3 Continuations for Higher-Order Functions

9. (8 pts total)

   a. (2 pts) Write a function `find_all : ('a -> bool) * 'a list -> 'a list)` that returns a list of all the elements in the input list `l` for which the input predicate `p` returns `true`, preserving the order and number of occurrences of the element satisfying `p`. You may not use library functions in this part.

   b. (6 pts) Write the function `find_allk : ('a -> (bool -> 'b) -> 'b) * 'a list -> ('a list -> 'b) -> 'b` that is the CPS transformation of **the code you wrote for Part (a)**. Be careful of the types.

```
# let rec find_all (p, l) = ... ;;
val find_all : ('a -> bool) * 'a list -> 'a list = <fun>
# find_all ( (fun x -> x mod 2 = 0), [-3; 5; 2; -6] );;
- : int list = [2; -6]

# let rec find_allk (p, l) k = ... ;;
val find_allk : ('a -> (bool -> 'b) -> 'b) * 'a list -> ('a list -> 'b) -> 'b =
  <fun>
# find_allk ((fun x -> fun k -> modk (x, 2) (fun n -> eqk (n, 0) k)),
```

```
              [-3; 5; 2; -6] ) print_int_list;;
[2; -6]
- : unit = ()
```

10. (8 pts total)

    a. (2 pts) Write a function `sum_all : (float -> bool) * float list -> float` that returns the floating point sum of all the elements in the input list `l` for which the input predicate `p` returns `true`, including multiplicity. The only form of recursion you may use for this problem is forward recursion. You may not use library functions or helper functions in this part.

    b. (6 pts) Write the function `sum_allk : (float -> (bool -> 'a) -> 'a) * float list -> (float -> 'a) -> 'a` that is the CPS transformation of **the code you wrote for Part (a)**. Be careful of the types.

```
# let rec sum_all (p, l) = ...;;
val sum_all : (float -> bool) * float list -> float = <fun>
# sum_all ( (fun x -> truncate x >= 2),  [1.3;2.5;3.9] );;
- : float = 6.4

# let rec sum_allk (p,l) k = ...;;
val sum_allk :
  (float -> (bool -> 'a) -> 'a) * float list -> (float -> 'a) -> 'a = <fun>
# sum_allk ((fun x -> fun k -> truncatek x (fun y -> geqk (y,2) k)),
            [1.3;2.5;3.9] ) report_float;;
Result: 6.4
- : unit = ()
```

4.4 CPS - Extra Credit

11. (6 pts total)

    a. (2 pts) Write a function `list_compose : (int -> int) list -> int` that takes a list of functions $l = [f_0; f_1; ...; f_n]$, an from right to left applied f_n to 0, and then f_{n-1} is applied to the output of f_n, etc down to f_0 is applied to the output of f_1. The result is the composition $f_0(f_1(\ldots(f_n 0)\ldots))$. If the list of functions is empty, `list_compose` should return the empty list.

    b. (4 pts) Write the function `list_composek` that is the CPS transformation of **the code you wrote for Part (a)**. In the CPS version, you need to assume that the list of functions to which `list_composek` might be applied has already been put in CPS.

```
# let rec list_compose fs = ...;;
val list_compose : (int -> int) list -> int = <fun>
# list_compose [(fun x -> x * x) ; (fun x -> x + 2)];;
- : int = 4
# let rec list_composek fsk k = ...;;
val list_composek : (int -> (int -> 'a) -> 'a) list -> (int -> 'a) -> 'a =
  <fun>
# list_composek [(fun x -> fun k -> mulk (x, x) k) ;
                 (fun x -> fun k -> addk (x, 2) k)] report_int;;
Result: 4
- : unit = ()
```