MP 4 – Continuation-Passing Style

CS 421 – Spring 2014
Revision 1.0

Assigned February 13, 2014
Due February 23, 2014 11:59 PM

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 below 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 being in tail recursive form, or 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

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`.

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.

1. **(10 pts)** Write the following low-level functions in continuation-passing style. A description of what each function should do follows:

   - `addk` adds two integers;
   - `subk` subtracts the second integer from the first;
   - `mulk` multiplies two integers;
   - `posk` determines if the argument is strictly positive;
   - `float_addk` adds two floats;
   - `float_divk` divides the first float by the second float;
   - `catk` concatenates two strings;
   - `consk` created a new list by adding an element at the front of a list;
   - `geqk` determines if the first argument is greater than or equal to the second argument; and
   - `eqk` determines if the two arguments are equal.

   ```
   # let addk n m k = ...;;
   val addk : int -> int -> (int -> 'a) -> 'a = <fun>
   # let subk n m k = ...;;
   val subk : int -> int -> (int -> 'a) -> 'a = <fun>
   ```

```
# let mulk n m k = ...;;
val mulk : int -> int -> (int -> 'a) -> 'a = <fun>
# let posk x k = ...;;
val posk : int -> (bool -> 'a) -> 'a = <fun>
# let float_addk a b k = ...;;
val float_addk : float -> float -> (float -> 'a) -> 'a = <fun>
# let float_divk a b k = ...;;
val float_divk : 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 eqk x y k = ...;;
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# let geqk x y k = ...;;
val geqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>

# addk 1 1 report;;
Result: 2
- : unit = ()
# catk "hello " "world" (fun x -> x);;
- : string = "hello world"
# float_addk 1.0 1.0
  (fun x -> float_divk x 2.0
   (fun y -> (print_string "Result: ";print_float y; print_newline())));;
    Result: 1.
- : unit = ()
# geqk 2 1 (fun b -> (report (if b then 1 else 0)));;
Result: 1
- : unit = ()
```

4.2 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.

2. **(5 pts)** Using `addk` and `mulk` as helper functions, write a function `polyk`, which takes on integer argument x and "returns" $x^3 + x + 1$. 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^3, then compute $x + 1$, and then compute $x^3 + x + 1$.

```
# let poly x k = ...;;
val poly : int -> (int -> 'a) -> 'a = <fun>
# poly 2 report;;
Result: 11
- : unit = ()
```

3. **(5 pts)** Write a function `composek`, which takes as the first two arguments two functions f and g, and as a third argument a value x, and "returns" $(g \circ f)(x)$ (the composition of f with g applied on x). A function h is equal with $g \circ f$ if and only if $h(x) = g(f(x))$ for all elements in the domain of f. You must write `composek` in continuation-passing style and you must assume that the functions f and g are given in the continuation-passing style.

```
# let composek f g x k = ...;;
val composek :
  ('a -> ('b -> 'c) -> 'd) ->
  ('b -> ('e -> 'f) -> 'c) -> 'a -> ('e -> 'f) -> 'd = <fun>
# composek inck inck 1 (fun x -> x);;
- : int = 3
```

4.3 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
```

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 4 through 6 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 1. If you need to make use of primitive operations not covered in Problem 1, you should include a definition of the corresponding version that takes a continuation as an additional argument, as in Problem 1. In Problem 5 and 6, there must be no use of list library functions.

4. **(6 pts total)**

   a. **(2 pts)** Write a function `inverse_square_series`, which takes an integer n, and computes the (partial) series $1 + \frac{1}{4} + \frac{1}{9} + \ldots + \frac{1}{n^2}$ and returns the result. If $n \leq 0$, then return 0.

   ```
   # let rec inverse_square_series n = ...;;
   val inverse_square_series : int -> float = <fun>
   # inverse_square_series 10;;
   - : float = 1.54976773116654076
   ```

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

   ```
   # let rec inverse_square_seriesk n k = ...;;
   val inverse_square_seriesk : int -> (float -> 'a) -> 'a = <fun>
   # inverse_square_seriesk 10 (fun x -> x);;
   - : float = 1.54976773116654076
   ```

5. **(8 pts total)**

   a. **(2 pts)** Write a function `rev_map` which takes a function f (of type `'a -> 'b`) and a list l (of type `'a list`). If the list l has the form $[a_1; a_2; \ldots; a_n]$, then `rev_map` applies f on a_n, then on a_{n-1}, then $\ldots$, then on a_1 and then builds the list $[f(a_1); \ldots; f(a_n)]$ with the results returned by f. The function f may have side-effects (e.g. `report`). There must be no use of list library functions.

   ```
   # let rec rev_map f l = ...;;
   val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
   # rev_map (fun x -> print_int x; x + 1) [1;2;3;4;5];;
   54321- : int list = [2; 3; 4; 5; 6]
   ```

   b. **(6 pts)** Write the function `rev_mapk` 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 -> 'b`, but `'a -> ('b -> 'c) -> 'c`.

   ```
   # let rec rev_mapk f l k = ...;;
   val rev_mapk :
     ('a -> ('b -> 'c) -> 'c) -> 'a list -> ('b list -> 'c) -> 'c = <fun>
   # let print_intk i k = k (print_int i);;
   val print_intk : int -> (unit -> 'a) -> 'a = <fun>
   # rev_mapk (fun x -> fun k -> print_intk x (fun t -> inck x k))
     [1;2;3;4;5] (fun x -> x);;
   54321- : int list = [2; 3; 4; 5; 6]
   ```

6. **(8 pts total)**

   a. **(2 pts)** Write a function `partition` which takes a list `l` (of type `'a list`), and a predicate `p` (of type `'a -> bool`), and returns a pair of lists (l_1, l_2) where l_1 contains all the elements in l satisfying p, and l_2 contains all the elements in l not satisfying p. The order of the elements in l_1 and l_2 is the order in l. There must be no use of list library functions.

   ```
   # let rec partition l p = ...;;
   val partition : 'a list -> ('a -> bool) -> 'a list * 'a list = <fun>
   # partition [1;2;3;4;5] (fun x -> x >= 3);;
   - : int list * int list = ([3; 4; 5], [1; 2])
   ```

   b. **(6 pts)** Write a function `partitionk` 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 partitionk l p k = ...;;
   val partitionk :
     'a list -> ('a -> (bool -> 'b) -> 'b) -> ('a list * 'a list -> 'b)
     -> 'b = <fun>
   # partitionk [1;2;3;4;5] (fun x -> fun k -> geqk x 3 k) (fun x -> x);;
   - : int list * int list = ([3; 4; 5], [1; 2])
   ```

4.4 Using Continuations to Alter Control Flow

As we have seen in the previous sections, continuations allow us a way of explicitly stating the order of events, and in particular, what happens next. We can use this ability to increase our flexibility over the control of the flow of execution (referred to as control flow). If we build and keep at our access several different continuations, then we have the ability to choose among them which one to use in moving forward, thereby altering our flow of execution. You are all familiar with using an if-then-else as a control flow construct to enable the program to dynamically choose between two different execution paths going forward.

Another useful control flow construct is that of raising and handling exceptions. In class, we gave an example of how we can use continuations to abandon the current execution path and roll back to an earlier point to continue with a different path of execution from that point. This method involves keeping track of two continuations at the same time: a primary one that handles "normal" control flow, and one that remembers the point to roll back to when an exceptional case turns up. As in regular continuation-passing style, the primary continuation should be continuously updated; however, the exception continuation remains the same. The exception continuation is then passed the control flow (by being called) when an exceptional state comes up, and the primary continuation is used otherwise.

7. **(8 pts)** Write a function `findk` which takes a list l, a predicate p, a normal continuation (of type `'a -> 'b`), and an exception continuation (of type `unit -> 'b`), and searches for the first element in l satisfying p. If there is such an element, it calls the normal continuation with the said element; otherwise, it calls the exception continuation with the unit ("`()`"). Your definition must be in continuation-passing style, and must follow the same restrictions about calling primitives as the previous section's problems. (For this problem, you will receive no points for the direct style definition of `find`, though writing it may be helpful when converting it to continuation-passing style.)

   ```
   # let rec findk l p normalk exceptionk = ...;;
   val findk :
     'a list -> ('a -> (bool -> 'b) -> 'b) -> ('a -> 'b) -> (unit -> 'b)
     -> 'b = <fun>
   # findk [1;2;3;4;5] (fun x -> fun k -> eqk x 3 k)
     (fun x -> x) (fun x -> print_string "element not found"; -1);;
   ```

```
- : int = 3
# findk [1;2;3;4;5] (fun x -> fun k -> eqk x 6 k)
  (fun x -> x) (fun x -> print_string "element not found"; -1);;
element not found- : int = -1
```

4.5 Extra Credit

8. **(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
```