MP 4 – Continuation-Passing Style

CS 421 – Fall 2012

Revision 1.1

Assigned September 18, 2012
Due September 25, 2012 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.1 In Problem 3, changed `m` < `n` to `n` < `m`,

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'll 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 trivial or straightforward recursive problems. You must first write each of the functions in this MP in direct style (according to the problem specification), then transform the function definition into continuation-passing style. In the first section, you will familiarize yourself with the basics of CPS transformations: returning values, creating continuations, and linearizing computations. In the second section, you will use your knowledge gained from the first section to transform recursive functions similar in complexity to problems you had in MP3 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 defintion 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. In fact, you will find it helpful to do so on several problems. All 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 (one provided by calls to the operating system) are "transformed" into functions that take teh arguments to the original operation and a continuation, and apply the continuation to the result of applying the primitive operation to 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 one integer from another;
   - `timesk` multiplies one integer from another;
   - `plusk` adds two floats;
   - `take_awayk` subtracts one float from another;
   - `multk` multiplies two floats;
   - `catk` concatenates two strings;
   - `consk` created a new list by adding an element onto the front of a list;
   - `lessk` determines if one argument is less than another; and
   - `eqk` tests if two arguments are equal.

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

```
val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
# let plusk x y k = ...;;
val plusk : float -> float -> (float -> 'a) -> 'a = <fun>
# let take_away x y k = k(x -. y);;
val take_away : float -> float -> (float -> 'a) -> 'a = <fun>
# let multk x y k = ...;;
val multk : float -> float -> (float -> 'a) -> 'a = <fun>
# let catk a b k = ...;;
val catk : string -> string -> (string -> 'a) -> 'a = <fun>
# let consk x l k = k(x :: l);;
val consk : 'a -> 'a list -> ('a list -> 'b) -> 'b = <fun>
# let lessk n m k = k(n < m);;
val lessk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# let eqk n m k = k(n=m);;
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>

# subk 10 5 report;;
Result: 5
- : unit = ()
# catk "hi " "there" (fun x -> x);;
- : string = "hi there"
# plusk 3.0 4.0
  (fun x -> multk x x
   (fun y -> (print_string "Result: ";print_float y; print_newline())));;
    Result: 49.
- : unit = ()
# lessk 2 7 (fun b -> (report (if b then 3 else 4)));;
Result: 3
- : 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 `multk` and `plusk` as helper functions, write a function `abcdk`, which takes four float arguments a b c d and "returns" $(a + (b \times c)) + (d \times a)$. You may only use the `multk` and `plusk` operators to do the arithmetic. The order of evaluation of operations must be as indicated by the parentheses in the given formula. Where there are ambiguities, evaluate teh expression on the right first.

```
# let abcdk a b c d k = ...
val abcdk : float -> float -> float -> float -> (float -> 'a) -> 'a = <fun>
# abcdk 2.0 3.0 4.0 5.0 (fun y -> report (int_of_float y));;
Result: 24
- : unit = ()
```

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 3 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. 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.

For each problem in this section, first write the function as described in direct style. Then, write the function again in continuation-passing style; append `k` to the name of the second definition. For example, if a problem asks you to write a function `splat`, then you should define `splat` in direct style and `splatk` in continuation-passing style.

3. **(5 pts total)**

   a. **(2 pts)** Write the function `fact_range`, which takes an integer `n`, multiplies it to all integers less than it down to `m` (inclusive), and returns the result. If `n` < `m`, then return 1.

```
# let rec fact_range n m = ...;;
val fact_range : int -> int -> int = <fun>
# fact_range 5 1;;
- : int = 120
```

   b. **(3 pts)** Write the function `fact_rangek : int -> int -> (int -> 'a) -> 'a` that is the CPS transformation of the code you wrote in part a.

```
# let rec fact_rangek n m k = ...;;
val fact_rangek : int -> int -> (int -> 'a) -> 'a = <fun>
# fact_rangek 7 5 report;;
Result: 210
- : unit = ()
```

4. **(6 pts total)**

   a. **(2 pts)** Write the function `duplicate_evens : 'a list -> 'a list` that takes a list `l` and creates a new list that duplicates every first, third, fifth, etc., (zero-based counting) element in the list. (The resulting list should be approximately 1.5 times as long.)

```
# let rec duplicate_evens l = ...;;
val duplicate_evens : 'a list -> 'a list = <fun>
# duplicate_evens [1; 2; 3; 4];;
- : int list = [1; 2; 2; 3; 4; 4]
```

   b. **(4 pts)** Write the function `duplicate_evensk : 'a list -> ('a list -> 'b) -> 'b` that is the CPS transformation of the code you wrote in part a.

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

5. **(8 pts total)**

   a. **(2 pts)** Write the function `first_or_default : ('a -> bool) -> 'a list -> 'a -> 'a` that takes a predicate `p`, a list `l`, and a value `d`, and returns the first element in the list of values that satisfies the predicate. If there is none, it returns `d`.

```
# let rec first_or_default p l d = ...;;
val first_or_default : ('a -> bool) -> 'a list -> 'a -> 'a = <fun>
# first_or_default (fun i -> 0 < i) [1; -2; 3; -4; 5] 0;;
- : int = 1
```

   b. **(6 pts)** Write the function `first_or_defaultk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> 'a -> ('a -> 'b) -> 'b` that is the CPS transformation of the code you wrote in part a. Your definition of `first_or_defaultk` must assume the predicate `p` will also be transformed into continuation-passing style; that is, its type is not `'a -> bool`, but `'a -> (bool -> 'b) -> 'b`. You must thread the continuation of `first_or_defaultk` through the call to `p` in your definition.

```
# let rec first_or_defaultk pk l d k = ...;;
val first_or_defaultk :
  ('a -> (bool -> 'b) -> 'b) -> 'a list -> 'a -> ('a -> 'b) -> 'b = <fun>
```

```
# first_or_defaultk (lessk 0) [1; -2; 3; -4; 5] 0 report;;
Result: 1
- : unit = ()
```

6. **(8 pts total)**

   a. **(2 pts)** Write the function `app_all : ('a -> 'b) list -> 'a -> 'b list` that takes a list of functions `flst` and a value `x` and creates the list made by applying each function in `flst` to `x`. The functions should be applied in the order in which they occur in the list, and the results list should be the order corresponding to the function list. You should assume OCaml order of evaluation.

   ```
   # let rec app_all flst x = ...;;
   val app_all : ('a -> 'b) list -> 'a -> 'b list = <fun>
   # app_all [((+) 1); (fun x -> x * x); (fun x -> 13)] 5;;
   - : int list = [6; 25; 13]
   ```

   b. **(6 pts)** Write the function `app_allk : ('a -> ('b -> 'c) -> 'c) list -> 'a -> ('b list -> 'c) -> 'c` that is the CPS transformation of the code you wrote in part a. Your definition of `app_allk` must assume the functions in the input list will also be in continuation-passing style; that is, the type of the list is not `('a -> 'b) list`, but `('a -> ('b -> 'c) -> 'c) list`.

   ```
   # let rec app_allk flstk x k = ...;
   val app_allk : ('a -> ('b -> 'c) -> 'c) list -> 'a -> ('b list -> 'c) -> 'c =
     <fun>
   # app_allk [(addk 1); (fun x -> timesk x x); (fun x -> (fun k -> k 13))] 5 (fun x
   - : int list = [6; 25; 13]
   ```

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 abilty 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 the function `sum_wholesk` that takes a list of integers, a regular continuation, and an exception continuation, and sums all the numbers in the list. If any of the integers is negative, call the exception continuation and pass the offending value to it. 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 `sum_wholes`, though writing it will be helpful for you to convert it to continuation-passing style.)

```
# let rec sum_wholesk l k xk = ...;;
val sum_wholesk : int list -> (int -> 'a) -> (int -> 'a) -> 'a
```

```
# sum_wholesk
    [0; -1; 2; 3]
    report
    (fun i ->
      print_string ("Error: " ^ (string_of_int i) ^ " is not a whole number");
      print_newline());;
Error: -1 is not a whole number
- : unit = ()
```

4.5 Extra Credit

8. **(8 pts)**

   Write the function `average_averagesk` that takes a list of lists of integers, a regular continuation, and an exception continuation for if there are no lists in the list, and an exception continutation for if one of the lists in the list of lists is empty, and calculates the average of the averages of each of the lists in the list of lists. If the list of lists is empty, then the first exception continuation (which is the second continuation) is called. If any of the lists is empty, then call the seconf exception continuation and pass the position of the offending list of lists to it (where we start counting from the left at 0). In that case no lists to the right of the offending list should have its average computed, nor any other data collected about it. In the case of a successful calculation, you should pass the average of the averages to the regular continuation. Your definition must be in continuation-passing style, and must follow the same restrictions about calling primitives as the previous sections' problems. For this problem, you will probably need to convert several more primitive operations to CPS, possibly including `/`, `/.`, `float_of_int` and `int_of_float`.

```
# let rec average_averagek ll k empty_list_list_xk empty_list_in_list_xk = ...;
- : float list list -> (float -> 'a) -> (unit -> 'a) -> (int -> 'a) -> 'a =
<fun>
# average_averagek [[1.; 2.; 3.]; [4.; 5.]; [6.; 7.; 8.]; [9.]]
            (fun a -> print_string "Result: "; print_float a; print_newline())
            (fun () -> print_string "Empty list!"; print_newline())
            (fun n -> print_string "Empty list at position ";
                      print_int n; print_newline());;
Result: 5.625
- : unit = ()
```