
MP 7 – CPS and a Lazy Interpreter for FUN

CS 421 – Summer 2007

Revision 1.6

Assigned Tuesday, July 17, 2007

Due Tuesday, July 24, 2007 at 23:59

Extension two days (20% penalty)

1 Change Log

1.6 Fixed evaluation rule for nullary operations to not recursively call the evaluation function for number 15.

1.5 Fixed `find_min` to pick out the **first** element of the list, as stated. Please also download the updated grader.

1.0 Initial release

2 Overview

There are two problem sections in this MP: the first is on Continuation Passing Style (CPS) and the second asks you to write a CPS interpreter for FUN that supports *lazy evaluation* based on an operational semantics for FUN. The first part is required for the whole class; the second part is required for the graduate students and extra credit for the undergraduate students.

Upon completion of this MP (if you do both parts), you should have the following skills:

- the ability to use CPS effectively,
- the ability to implement a call-by-need language,
- the ability to write a realistic interpreter for an ML-like language, and
- the ability to impress people with your knowledge of CPS at a cocktail party

Enjoy!

3 Files

There are two graders, **mp7Agrader** and **mp7Bgrader**. The first grader is for the first part of this assignment, and the second is for the second part of the assignment.

The only files you should modify and submit are **mp7A.ml** and **mp7B.ml**.

mp7Agrader is fairly simple; it contains only the file **mp7A-skeleton.ml**, which you should rename to **mp7A.ml**.

mp7Bgrader is a bit more complicated, but it is very similar to the grader for the last assignment; in particular, upon running **make** or **gmake** you will get three executables: **mp7intB**, **mp7BintSol** and **grader**.

The file **mp7common.cmo** contains the datatype for expressions (input to your evaluator) and the datatypes for values (output of your evaluator). For this MP, **mp7common.cmo** also defines the `thunk` type, to be explained below.

You are also given files **mp7lex.cmo**, **mp7yacc.cmo**, etc...

4 Continuation Passing Style

These exercises are designed to give you a feel for continuation passing style before you build your interpreter.

A function that is written in continuation passing style does not return once it has finished its computation. Instead, it calls another function (the continuation) with the result.

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 Part A - CPS Problems (required for all)

You have been given a file called `mp7A-skeleton.ml`. You copy this to `mp7A.ml` and write your answers there.

The following six problems demonstrate different techniques of CPS. None of the functions you write may directly return a value; all must be written in CPS.

WARNING: When writing a function in CPS, never use `fun x -> x` as a continuation. This is essentially taking a CPS function and treating it like a non-CPS function. The only place this is allowed is when calling a CPS function at top-level from the interpreter, never in your code.

To put it another way, any time you call a function f that accepts a continuation in the following problems, you must pass a *valid continuation* to f . A valid continuation is defined as a function which, along all branches of control flow, eventually transfers control to a continuation that was passed in to the current scope. Since, conceptually, CPS functions *never return*, all calls to functions in continuation passing style must occur in *tail position*.

Simple Continuations

1. (5 pts) Write the functions `lessk`, `leqk`, `addk`, `subk`, and `andk` in CPS.

`lessk` returns true iff the first argument is less than the second; `leqk` returns true iff the first argument is less than *or equal to* the second; `addk` adds its two integer arguments; `subk` subtracts the second integer argument from the first; and `andk` returns true iff both of its boolean arguments are `true`.

```

val lessk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
val leqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
val addk : int -> int -> (int -> 'a) -> 'a = <fun>
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
val andk : bool -> bool -> (bool -> 'a) -> 'a = <fun>

# lessk 3 4 (fun x -> x);;
- : bool = true
# leqk 3 3 (fun x -> x);;
- : bool = true
# addk 12 14 report;;
Result: 26
- : unit = ()
# subk 10 5 report;;
Result: 5
- : unit = ()
# andk true false (fun x -> x);;
- : bool = false
# addk 1 2 (fun x -> subk x 3 report);;
Result: 0
- : unit = ()
# addk 1 2 (subk 3) report;;
Result: 0
- : unit = ()

```

Nesting Continuations One common technique used in CPS is that of nesting continuations. For example, consider the following code:

```

# 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 `lessk` (and possibly `andk`), write `orderedk x y z k` which passes `true` to `k` iff `x < y` and `y < z`.

Warning: You may **not** use the OCaml operators `<` or `and`. You must instead use `lessk` and `andk`, and use them properly. For example, do not do `lessk x y (fun x -> x)`. Never use `fun x -> x` to coerce a CPS function into a non-CPS function.

```

# orderedk;;
- : 'a -> 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# orderedk 3 4 5 (fun x -> x);;
- : bool = true
# orderedk 5 7 2 (fun x -> x);;
- : bool = false

```

Recursion and Continuation 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
```

To put the function into 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. Thus the code becomes:

```
# let rec factorialk n k =
  if n = 0 then k 1 else factorialk (n - 1) (fun m -> k (n * m));;
val factorialk : int -> (int -> 'a) -> 'a = <fun>
# factorialk 5 report;;
120
- : unit = ()
```

To make a recursive call, we must built an intermediate continuation to:

- take the recursive value: m
- build it to the final result: $n * m$

And pass it to the final continuation: $k (n * m)$. Notice that this is an extension of the "nested continuation" method.

Mapping Recursion Suppose we want to use CPS to write a function that increments every element of a list. Here is the resulting code:

```
# let rec incList lst k =
  match lst with
  | [] -> k []
  | x::xs -> inck x (fun v ->
                    incList xs (fun vs ->
                                   k (v::vs)));;
val incList : int list -> (int list -> 'a) -> 'a = <fun>
# incList [2;3;4] report_list;;
Result: [3; 4; 5]
- : unit = ()
```

This is really just again a variation of the "nested continuation" method. The first continuation passed to `inck` saves the result of incrementing x as v . The continuation then calls `incList` on xs , which will increment the tail of the list. This result is saved in the second continuation as vs . This second continuation then combines the parts together, and gives the result to the original continuation.

3. **(10 pts)** Write `doublek lst k` that passes the final continuation a `lst` with its integer elements doubled.

Note: You must use `addk`. Do **not** use `+` or `*` (or any other such OCaml function.)

```
# doublek;;
- : int list -> (int list -> 'a) -> 'a = <fun>
```

```
# doublek [1;2;3;4;5] (List.map report);;
Result: 2
Result: 4
Result: 6
Result: 8
Result: 10
- : unit list = [(); (); (); (); ()]
```

4. **(10 pts)** Without calling `doublek`, Write `quadruple lst` that passes the final continuation a `lst` with its integer elements quadrupled.

Note: You may use `addk`, but do **not** use `+` or `*` (or any other such OCaml function.) **Do not call `doublek`.**

```
# quadruplek [1;2;3;4;5] (List.map report);;
Result: 4
Result: 8
Result: 12
Result: 16
Result: 20
```

Folding Recursion The other common recursion is folding recursion. Look at this example and observe how it differs from the previous example.

```
let rec sumList list k =
  match list with
  [] -> k 0
  | x::xs -> sumList xs (fun y -> addk x y k)
```

Notice here that the recursive call comes first. The result is saved in a variable `y` to remind you of the original `fold_right` definition. You know already that the recursion has to occur before the current element can be combined with it; this definition makes that explicit. This is a major feature of CPS: the order of operations is made explicit.

```
# sumList [2;3;4;5] report;;
Result: 14
- : unit = ()
```

5. **(10 pts)** Write `allk fk lst k`, which passes to `k` the elements of `lst` for which `fk` passes `true` to its continuation.

Note that `fk` is also in continuation passing style, so when you call `fk` you must give it a continuation that finishes the computation. Do **not** pass `fun x -> x` to `fk`.

```
# allk;;
- : ('a -> (bool -> 'b) -> 'b) -> 'a list -> ('a list -> 'b) -> 'b = <fun>
# allk (lessk 0) [1;2;0;-1;3] (List.map report);;
Result: 1
Result: 2
Result: 3
- : unit list = [(); (); ()]
```

Note: You may not use any functions from the List library in this problem.

6. (10 pts) Write `maxk fk lst k`, which passes to `k` Some `x`, where `x` is the maximum element of `lst` for which `fk` passes `true` to its continuation, or `None` if there is no such element in `lst`.

To break ties favor elements closer to the head of `lst`.

Note that `fk` is also in continuation passing style, so when you call `fk` you must give it a continuation that finishes the computation. Do **not** pass `fun x -> x` to `fk`.

```
# maxk;;
- : ('a -> (bool -> 'b) -> 'b) -> 'a list -> ('a option -> 'b) -> 'b = <fun>
# maxk (fun x -> lessk x 0) [1;2;0;-3;-5;2;-1] (fun x -> x);;
- : int option = Some (-1)
# maxk (fun x k -> k false) [1;2;3] (fun x -> x);;
- : int option = None
```

Note: You may not use any functions from the List library in this problem.

Saving Continuations Recall from lecture that by saving the continuation passed to the initial call of a recursive function we have the ability to abort a computation before it has even begun. The usual method of accomplishing this is to make a local auxiliary function do all the recursion.

```
let prodList list ka =
  let rec aux list k =
    match list with
    | [] -> k 1
    | 0::xs -> ka 0
    | x::xs -> aux xs (fun y -> k (x * y))
  in aux list ka
```

If a zero is encountered, the continuation `k` is thrown out and the original continuation `ka` is used instead.

7. (10 pts) Write `diffk y lst ka` that passes to `ka` the list with each corresponding element of `lst` subtracted from `y`.

You should use an auxiliary function with an auxiliary continuation initialized to the original continuation.

If at any point the difference between `y` and some element in the list is negative, then the auxiliary function should directly supply to the original continuation the empty list.

You must use `subk` and `lessk` (or `leqk`) instead of the corresponding OCaml operators.

```
# diffk;;
- : int -> int list -> (int list -> 'a) -> 'a = <fun>
# diffk 10 [1;2;3;4] (List.map report);;
Result: 9
Result: 8
Result: 7
Result: 6
- : unit list = [(); (); (); ()]
# diffk 10 [1;2;11;4] (List.map report);;
- : unit list = []
```

Other Problems

8. (10 pts) Write `find_mink lst k` that supplies the continuation `k` with `Some (x, lst')`, where `x` is the **first occurrence** of the minimum element of `lst` and `lst'` is `lst` with that occurrence removed, or `None`, if `lst` is empty.

Note: For comparisons, you must use `lessk`. **Note:** The third example below has been fixed in version 1.5. Please also download the latest grader.

```
# find_mink;;
- : 'a list -> (('a * 'a list) option -> 'b) -> 'b = <fun>
# find_mink [2;3;4;3;5] (fun x -> x);;
- : (int * int list) option = Some (2, [3; 4; 3; 5])
# find_mink [4;3;8;3;7] (fun x -> x);;
- : (int * int list) option = Some (3, [4; 8; 3; 7])
# find_mink [] (fun x -> x);;
- : ('a * 'a list) option = None
```

9. (10 pts) Write `sortk lst k`, which passes to `k` `lst` in sorted order.

Hint: You should call `find_mink`.

```
# sortk;;
- : 'a list -> ('a list -> 'b) -> 'b = <fun>
# sortk [2;4;8;9;3;1] (List.map report);;
Result: 1
Result: 2
Result: 3
Result: 4
Result: 8
Result: 9
- : unit list = [(); (); (); (); (); ()]
```

6 Part B (extra credit for undergrads)

Before you start any work, there are a few new things with which you should familiarize yourself.

6.1 Values and Memories

Since we are evaluating FUN, we must talk about the values in the language. The values of a language encompass everything to which an expression can evaluate. They are defined in `mp7common.cmo` as the `value` disjoint datatype.

We also need to keep track of memories (value environments), which are mappings from variables to their values. Compare this with type environments, which map variables to their types. In general, throughout this writeup, the symbol `m` is used to denote an arbitrary memory.

6.2 Thunks and Forcing

Evaluation for this MP is **call-by-need**, meaning that an expression is not evaluated until it is first used. The difference between this and the lazy evaluation of the lambda-calculus (which is **call-by-name**) is that once an expression is evaluated, it is never evaluated again. Instead, its value is memoized. For an example of this, see problem 17.

To accomplish this, we introduce a special kind of value, called a *thunk* (Thunk). There are two kinds of thunks:

- A suspended thunk, `Suspension`, which holds an expression and a memory.
- An evaluated thunk `Value`, which just points to a non-thunk value.

A suspended thunk is similar to a closure. Remember that when a function is created, the result is a triple consisting of: the variable to which the argument is bound, the function expression, and the memory that was active when the function was created. A suspended thunk is similar, except that the enclosed expression is able to be something other than a function, and there is no variable. When a suspension is evaluated, we get the other kind of thunk, a `Value`, which contains an evaluated expression.

This is how we implement lazy evaluation, and in particular, call-by-need parameter passing. In call-by-value, we would evaluate a function’s argument to a value, then pass that value to the function. In call-by-need, we create a thunk out of the expression for the argument, and pass the thunk value to the function. It only gets computed when needed! Whenever an operation does require the actual (non-thunk) value of something, we force the thunk. Forcing means that we actually evaluate the suspended expression to get a value. In addition, the thunk gets changed to an evaluated thunk, where it stores the resulting value. This way, we don’t have to evaluate the thunk again if it is used more than once. Thunks are represented by the `Thunk` constructor, which takes a thunk. The `Thunk` datatype is defined in `mp7common.cmo`. Although you will never have to play with the internals of the `Thunk` datatypes directly, it is worth noting that the `Thunk` datatype stores a **reference** to either a `Suspension` or a `Value`. This is how we are able to change a thunk from a suspended one to an evaluated one (which would be hard without somehow using side-effects).

There are some support functions in `mp7common.cmo` to help you use thunks:

- `mk_thunk` — takes an expression and an environment, and creates a thunk with them.
- `force` — takes a thunk and a continuation, forces the thunk, and passes the resulting thunk to the continuation. Suspensions are evaluated and turned into values. Values are “returned” (passed to the given continuation) as is.

6.3 Types

Since you cannot access the source for `mp7common.cmo`, here are the important types:

```
(* variables *)
type id = string
(* Type environment *)
type tyenv = (id * tysch) list
(* Expressions *)
type exp = UnitExp | BoolExp of bool | IntExp of int | FloatExp of float
          | IdExp of id
          | PairExp of exp * exp
          | FunExp of id * exp
          | AppExp of exp * exp
          | IfExp of exp * exp * exp
          | LetExp of id * exp * exp
          | LetRecExp of id * exp * exp
.
.
.
type memory = (id * value) list
and value =
  UnitVal
  | BoolVal of bool
  | IntVal of int
  | FloatVal of float
```

```

| PairVal of value * value
| ListVal of value list
| Closure of id * exp * memory
| RecVar of id * exp * memory
| Think of thunk (* for MP7 *)
(* MP7 stuff *)
and thinker = Suspension of exp * memory | Value of value
and think = thinker ref

(* think helper *)
let mk_thunk e m = ref (Suspension(e, m))

```

6.4 The Evaluator

The main function is `eval_expk ex m k`. It takes three arguments: an expression, a memory, and a continuation. The continuation, when invoked, takes a value and then completes the rest of the evaluation based on that value.

This function is appropriately called from the interpreter.

To run the interpreter, you should be able to type `gmake` and then type `mp7intB`, `mp7BintSol`, or `grader`.

7 Evaluator Problems (extra credit for undergrads)

7.1 Notation

There will be a text description of each of the features, along with a mathematical description. We will use the following notation. A suspension will be represented by $\langle e, m \rangle$. The e is the suspended expression, and m is the memory (environment). Value thunks will be represented by $\langle v \rangle$. Closures will be represented as $\langle x, e, m \rangle$. Memory elements will be represented by $\{x \rightarrow v\}$, indicating that variable x is mapped to value v (either a thunk, a `rec(...)`, or a unit, bool, list, etc...). The evaluator for expressions will be represented by a three-argument function $\mathcal{E}_e(e, m, k)$, where e is the expression to evaluate, m is the memory, and k is the continuation. The function “force” is represented by \mathcal{F} .

You should be sure to use an “iterative” method for programming. A lot of pain can be avoided by making sure that you get one feature working before trying to add another.

10. Simple Types (5 pts)

Create `eval_expk ex m k` and make it handle unit, bools, ints, and floats.

$$\begin{aligned}
 \mathcal{E}_e((), m, k) &= k () \\
 \mathcal{E}_e(b, m, k) &= k b \\
 \mathcal{E}_e(i, m, k) &= k i \\
 \mathcal{E}_e(f, m, k) &= k f
 \end{aligned}$$

```

# 3;;
- : int = 3
# ();;
- : unit = ()
# false;;
- : bool = false

```

11. **Identifiers (thunks)** (5 pts) Extend `eval_exprk` to handle identifiers that go to thunks.

At the point where we actually *need* to use the value of an identifier bound to a thunk, the thunk must be forced. Note that if the thunk has been forced before, then it is already a value thunk and nothing will be done. If it is a suspension, however, it will be evaluated to a value thunk.

$$\mathcal{E}_e(x, m, k) = \mathcal{F}(v_t, k) \quad \text{when } x \rightarrow v_t \text{ is in } m, v_t \text{ is a thunk value}$$

12. **Pairs** (5 pts) Extend `eval_exprk` to handle pairs.

We are not going to give you the rule for this one; you are on your own.

$$\mathcal{E}_e((e_1, e_2), m, k) = \dots$$

```
# (8, false);;
- : int * bool = (8, false)
# (((), ((), (1, (9, 3.0)))));;
- : unit * (unit * (int * (int * float))) = (((), ((), (1, (9, 3.)))))
```

13. **Let...In... Declarations** (5 pts)

Extend `eval_exprk` `ex m k` to handle `let...in` declarations.

$$\mathcal{E}_e(\text{let } x = e_1 \text{ in } e_2, m, k) = \mathcal{E}_e(e_2, \{x \rightarrow \langle e_1, m \rangle\} + m, k)$$

```
# let x = 3 in 5;;
- : int = 5
# let x = 3 in x;;
- : int = 3
# let x = 3 in let y = 4 in (x, y);;
- : int * int = (3, 4)
```

14. **Let...Rec...In... Declarations** (5 pts)

Extend `eval_exprk` `ex m k` to handle `let...rec...in` declarations.

$$\mathcal{E}_e(\text{let rec } x = e_1 \text{ in } e_2, m, k) = \mathcal{E}_e(e_2, \{x \rightarrow \langle e_1, \{x \rightarrow \text{RecVar}(x, e_1, m)\} + m \rangle\} + m, k)$$

```
# let rec x = 3 in x;;
- : int = 3
```

Note: More interesting examples will only work after you have complete the problem on *recursive identifiers* below.

15. **Application of built in operators** (10 pts) Extend `eval_exprk` to handle applications of built-in operators.

$$\begin{aligned} \mathcal{E}_e(x, m, k) &= k(\text{get_nullary}(x)) && \text{when } \text{is_nullary}(x) \\ \mathcal{E}_e(x \ e, m, k) &= \mathcal{E}_e(e, m, \lambda v. k(\text{app_unary}(x, v))) && \text{when } \text{is_unary}(x) \\ \mathcal{E}_e(x \ e_1 \ e_2) &= \mathcal{E}_e(e_1, m, \lambda v_1. \mathcal{E}_e(e_2, m, \lambda v_2. k(\text{app_binary}(x, v_1, v_2)))) && \text{when } \text{is_binary}(x) \end{aligned}$$

Note that when we are evaluating built-in operators, evaluation is strict.

```

# 3 + 4 * 2;;
- : int = 11
# (not true) or false;;
- : bool = false
# [];;
- : 'a list = []
# 3::[];;
- : int list = [3]

```

16. Functions (10 pts)

Extend `eval_expk` to handle functions. You will need pass a closure to the given continuation.

$$\mathcal{E}_e(\text{fun } x \rightarrow e, m, k) = k\langle x, e, m \rangle$$

```

# fun x -> x + x;;
- : int -> int = <fun>

```

17. Function application (10 pts)

Extend `eval_expk` to handle function application in a lazy fashion.

$$\mathcal{E}_e(e_1 e_2, m, k) = \mathcal{E}_e(e_1, m, \lambda\langle x, e_1', m' \rangle. \mathcal{E}_e(e_2, \{x \rightarrow \langle e_2, m \rangle\} + m', k))$$

You first evaluate e_1 to a closure $\langle x, e_1', m' \rangle$. Then you evaluate e_2 in an environment where $x \rightarrow \langle e_2, m \rangle$. This is a thunk representing the evaluation of e_2 in m . Note that this evaluation will not actually take place until it is needed.

Note: In your implementation, you are adding an additional case (the case where e_1 evaluates to a closure) to your code from problem 15.

```

# (fun x -> fun y -> x y) (fun x -> x + x) 10;;
- : int = 20

```

Is your evaluation lazy (call-by-need)?

```

# (fun x -> 4) (print 1);;
- : int = 4
# (fun x -> (x, (x, 4))) (print 1);;
1
- : unit * (unit * int) = ((), ((), 4))

```

The first time, `x` is never referred to so nothing gets printed. The second time, `x` is referred to twice but `1` is only printed once, because once a thunk is evaluated, its value is memoized and never computed again.

Note: In call-by-name, `1` would have been printed twice because it was referred to twice.

Note: The operator `print` was added to `mp7common.cmo` for this MP.

18. **If constructs** (10 pts)

Extend `eval_expk` to handle if constructs. We are not going to give you the rule for this one; you are on your own.

$$\mathcal{E}_e(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, m, k) = \dots$$

Hint: you will need to call \mathcal{E}_e on e_1, m and k , and pass the result to some continuation that knows what to do with the result.

```
# 3 + (if 3 > 2 then 3 else 2);;
- : int = 5
```

19. **Recursive Identifiers** (10 pts)

Extend `eval_expk` to handle recursive identifiers. These are identifiers that go to $\text{rec}\langle x, e, m \rangle$ for some variable x , expression e , memory m .

$$\mathcal{E}_e(x, m, k) = \mathcal{E}_e(e', \{y \rightarrow \text{rec}\langle y, e', m' \rangle\} + m', k) \quad \text{when } x \rightarrow \text{rec}\langle y, e', m' \rangle \text{ is in } m$$

```
# let rec f = (fun x -> if x < 1 then 1 else x * f(x - 1)) in f 10;;
- : int = 3628800
```