CS421: Progrmg Languages & Compilers
Summer 2015

HW 3: Solutions

University of Illinois, Urbana-Champaign,

1 Theoretical Questions (points)

Problem 1: (30 points)

Compute the following CPS transformation. All parts should be transformed completely.

$$[[\texttt{fn f => fn x => if x > 0 then f x else f ((-1) * x)}]]_{\texttt{(FUN w -> report w)}}$$

Solution:

$[[\texttt{fn f => fn x => if x > 0 then f x else f ((-1) * x)}]]_{\texttt{(FUN w -> report w)}}$

$= \texttt{(FUN w -> report w) (FN f k1 -> } [[\texttt{fn x => if x > 0 then f x else f ((-1) * x)}]]_{\texttt{k1}}\texttt{)}$

$= \texttt{(FUN w -> report w) (FN f k1 -> k1 (FN x k2 -> } [[\texttt{if x > 0 then f x else f ((-1) * x)}]]_{\texttt{k2}}\texttt{))}$

$= \texttt{(FUN w -> report w) (FN f k1 -> k1 (FN x k2 -> } [[\texttt{x>0}]]_{(A)}\texttt{))}$

$= \texttt{(FUN w -> report w) (FN f k1 -> k1 (FN x k2 -> } B\texttt{))}$

$= \texttt{(FUN w -> report w) (FN f k1 -> k1 (FN x k2 -> (FUN y -> IF y THEN } C \texttt{ ELSE } D\texttt{) (x > 0))},$

where

$A = \texttt{FUN y -> IF y THEN } [[\texttt{f x}]]_{\texttt{k2}} \texttt{ ELSE } [[\texttt{f ((-1) * x)}]]_{\texttt{k2}}$

$B = \texttt{(FUN y -> IF y THEN } [[\texttt{f x}]]_{\texttt{k2}} \texttt{ ELSE } [[\texttt{f ((-1) * x}]]_{\texttt{k2}}\texttt{) (x > 0)}$

$C = [[\texttt{f}]]_{\texttt{(FUN v1 -> }[[\texttt{x}]]_{\texttt{(FUN v2 -> v1 v2 k2)}}\texttt{)}}$

$= \texttt{(FUN v1 -> } [[\texttt{x}]]_{\texttt{(FUN v2 -> v1 v2 k2)}}\texttt{) f}$

$= \texttt{(FUN v1 -> (FUN v2 -> v1 v2 k2) x) f}$

$D = [[\texttt{f}]]_{\texttt{(FUN v3 -> }[[\texttt{(-1) * x}]]_{\texttt{(FUN v4 -> v3 v4 k2)}}\texttt{)}}$

$= [[\texttt{f}]]_{\texttt{(FUN v3 -> }[[\texttt{-1}]]_{\texttt{(FUN v5 -> }[[\texttt{x}]]_{\texttt{(FUN v6 -> (FUN v4 -> v3 v4 k2) (v5 * v6))}}\texttt{)}}\texttt{)}}$

$= [[\texttt{f}]]_{\texttt{(FUN v3 -> }[[\texttt{-1}]]_{\texttt{(FUN v5 -> (FUN v6 -> (FUN v4 -> v3 v4 k2) (v5 * v6)) x)}}\texttt{)}}$

$= [[\texttt{f}]]_{\texttt{(FUN v3 -> (FUN v5 -> (FUN v6 -> (FUN v4 -> v3 v4 k2) (v5 * v6)) x) (-1))}}$

$= \texttt{(FUN v3 -> (FUN v5 -> (FUN v6 -> (FUN v4 -> v3 v4 k2) (v5 * v6)) x) (-1)) f}$

Problem 2: (5 points)

A 2–3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements.

Give an OCaml datatype that can represent 2-3 trees with `int` keys and `string` data. Your datatype should be able to represent 2-3 trees in a clear fashion but does not need to enforce balance or ordering requirements (i.e., your representation may admit illegal trees of the right "shape").

Solution:

```
type tree = Empty
| OneLeaf of int * string
| TwoLeaf of int * string * int * string
| TwoNode of tree * int * string * tree
| ThreeNode of tree * int * string * tree * int * string * tree;;
```

Problem 2: (5 points)

A red-black tree is a kind of tree sometimes used to organize numerical data. It has two types of nodes, black nodes and red nodes. Red nodes always have one piece of data and two children, each of which is a black node. Black nodes may have either: 1) one piece of data and two children that are red nodes; 2) one piece of data and two children that are black nodes; or 3) no data and no children (i.e., a leaf node). (This isn't a precise description of red-black trees but suffices for this exercise.)

Write a pair of mutually recursive OCaml datatypes that represent red nodes and black nodes in red-black trees. The *data* should be able to have any type, that is, your type should be polymorphic in the kind of data stored in the tree.

Solution:

```
type 'a black_node = Leaf
| RNode of 'a * 'a red_node * 'a red_node
| BNode of 'a * 'a black_node * 'a black_node
and 'a red_node = RedNode of 'a * 'a black_node * 'a black_node;;
```

2 Machine Problems (points)

Higher order functions

Problem 1: (4 point)

The objective of this problem is to write a function `minmax_list` that takes an `int list list` and returns an `int * int list` such that the i^{th} tuple contains the minimum and maximum values of the i^{th} list.

To perform this operation, write a function `minmax` using tail recursion (explicit or using `fold_left`) that takes a list of integers and returns a 2-tuple of (min,max). Empty list should map to (0,0). Use this function and `List.map` to obtain the said function `minmax_list`.

```
# let minmax lst = ...;;
val minmax : int list -> int * int = <fun>
# minmax [1;-2;4;5;-8];;
- : int * int = (-8, 5)
# let minmax_list lst = ...;;
val minmax_list : int list list -> (int * int) list = <fun>
# minmax_list [[1];[-1;2];[1;3;4];[]];;
- : (int * int) list = [(1, 1); (-1, 2); (1, 4); (0, 0)]
```

Solution:

```
let minmax lst = match lst with
       [] -> (0,0)
    | x::xs -> List.fold_left (fun (min,max) y
           -> (if y < min
                 then (y,max)
                 else if max < y
                    then (min,y)
                    else (min,max)))
           (x,x) xs;;

let minmax_list lst = List.map minmax lst;;
```

Problem 2: (4 point)

The objective of this problem is to write a function `cumlist` that takes `'a list` and returns `'a list list` such that i^{th} list is the sublist from the beginning to the element i using the library routine `List.fold_right`.

To perform this operation, write an auxillary function `cumlist_step` that you can pass to `List.fold_right`. You can write this function in any manner you see fit.

```
# let cumlist_step <args> = ...;;
val cumlist_step : <type> = <fun>
# let cumlist lst = List.fold_right cumlist_step lst <base_case>;;
val cumlist : 'a list -> 'a list list = <fun>
# cumlist [1; 2; 3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]]
```

Solution:

```
let cumlist_step x lst = [x] :: (List.map (fun l -> x::l) lst);;
let cumlist lst = List.fold_right cumlist_step lst [];;
```

Problem 3: (4 point)

The objective of this problem is to write a function `revsplit`, that, when applied to a test function `f`, and a list `lst`, returns a pair of lists. The first list of the pair should contain every element `x` of `lst` for which `(f x)` is true; and the second list contains every element for which `(f x)` is false. The order of the elements in the returned lists should be the reverse of the order in the original list.

To perform this operation, write an auxillary function `revsplit_step` that you can pass to `List.fold_left`.

```
# let revsplit_step f <args> = ...;;
val revsplit_step : <type> = <fun>
# let revsplit f lst = List.fold_left (revsplit_step f) <base_case> lst;;
val revsplit : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
# revsplit (fun x -> x < 0) [-1; 4; -2; 3; 0; 1];;
- : int list * int list = ([-2; -1], [1; 0; 3; 4])
```

Solution:

```
let revsplit_step f (tr,fl) x = if (f x) then (x::tr,fl) else (tr,x::fl);;
let revsplit f lst = List.fold_left (revsplit_step f) ([],[]) lst;;
```

Continuation-passing style

Problem 4: (8 points)

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

- `andk` performs boolean 'and' of the two bolean values following the 'and' rules of evaluation;
- `ork` performs boolean 'or' of the boolean values following the 'or' rules of evaluation;
- `landk` performs bitwise 'and' of the two integers;
- `lork` performs bitwise 'or' of the two integers;
- `lxork` performs bitwise 'xor' of the two integers;
- `expk` computes e raised to the float provided;
- `logk` computes log of the float provided to base e;
- `powk` raises the first float to the second float (exponentiation);

Note that all these operations have primitive functions.

```
# let andk a b k = ...;;
val andk : bool -> bool -> (bool -> 'a) -> 'a = <fun>
# let ork a b k = ...;;
val ork : bool -> bool -> (bool -> 'a) -> 'a = <fun>
# let landk a b k = ...;;
val landk : int -> int -> (int -> 'a) -> 'a = <fun>
# let lork a b k = ...;;
val lork : int -> int -> (int -> 'a) -> 'a = <fun>
# let lxork a b k = ...;;
```

```
val lxork : int -> int -> (int -> 'a) -> 'a = <fun>
# let expk a k = ...;;
val expk : float -> (float -> 'a) -> 'a = <fun>
# let logk a k = ...;;
val logk : float -> (float -> 'a) -> 'a = <fun>
# let powk a b k = ...;;
val powk : float -> float -> (float -> 'a) -> 'a = <fun>
```

Solution:

```
let andk a b k = k (a && b);;
let ork a b k = k (a || b);;
let landk a b k = k (a land b);;
let lork a b k = k (a lor b);;
let lxork a b k = k (a lxor b);;
let expk a k = k (exp a);;
let logk a k = k (log a);;
let powk a b k = k (a ** b);;
```

Nesting Continuations

Problem 5: (8 points)

Implement integer modulo arithmetic operations in continuation-passing style as described below.

- `modaddk a b n k` computes `(a + b) mod n`;
- `modsubk a b n k` computes `(a - b) mod n`;
- `modmulk a b n k` computes `(a * b) mod n`;
- `modeqk a b n k` tells if `a` and `b` are congruent in mod n;

you can only use operations in CPS that are defined in Mp3common.

```
# let modaddk a b n k = ...;;
val modaddk : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# modaddk 5 7 10 rep_int;;
Result: 2
- : unit = ()

# let modsubk a b n k =...;;
val modsubk : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# modsubk 12 4 7 rep_int;;
Result: 1
- : unit = ()

# let modmulk a b n k = ...;;
val modmulk : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# modmulk 3 4 8 rep_int;;
Result: 4
- : unit = ()

# let modeqk a b n k = ...;;
val modeqk : int -> int -> int -> (bool -> 'a) -> 'a = <fun>
# modeqk 6 11 5 rep_bool;;
Result: true
- : unit = ()
```

Solution:

```
let modaddk a b n k = addk a b (fun ab -> modk ab n k);;
let modsubk a b n k = subk a b (fun ab -> modk ab n k);;
let modmulk a b n k = mulk a b (fun ab -> modk ab n k);;
let modeqk a b n k = subk a b (fun ab -> modk ab n (fun d -> eqk d 0 k));;
```

Transforming recursive functions

Problem 6: (6 points)

a. **(2 pts)**

Write a function `power`, which takes two integers a, n; and computes a^n by multiplying a to itself n times. It should return 1 if $n \leq 0$.

```
# let rec power a n = ...;;
val power : int -> int -> int = <fun>
# power 2 3;;
- : int = 8
```

Solution:

```
let rec power a n = if n <= 0 then 1 else a * (power a (n - 1));;
```

b. **(4 pts)**

Write the function `powerk` which is the CPS transformation of the code you wrote in part a.

```
# let rec powerk a n k = ...;;
val powerk : int -> int -> (int -> 'a) -> 'a = <fun>
# powerk 2 3 rep_int;;
Result: 8
- : unit = ()
```

Solution:

```
let rec powerk a n k =
       leqk n 0 (fun b -> (if b then (k 1) else (subk n 1
                            (fun s -> powerk a s (fun m -> mulk a m k)))));;
```

Problem 7: (6 points)

a. **(2 pts)**

Write the function `dup_alt` that takes a list `l` and creates a new list that duplicates every alternate element starting from the second.

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

Solution:

```
let rec dup_alt lst =
       match lst with [] -> []
       | x::[] -> [x]
       | x::y::xs -> x::y::y:: dup_alt xs;;
```

b. **(4 pts)**

Write the function `dup_altk` that is the CPS transformation of the code you wrote in part a.

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

Solution:

```
let rec concatk l1 l2 k =
       match l1 with [] -> k l2
       | x::xs -> concatk xs l2 (fun l -> consk x l k);;
let rec dup_altk lst k =
       match lst with [] -> k []
       | x::[] -> k [x]
       | x::y::xs -> dup_altk xs (fun d -> concatk [x] [y] (fun xy -> concatk xy [y] (fun xyy
           -> concatk xyy d k)));;
```

Problem 8: (10 points)

a. **(3 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]
```

Solution:

```
let rec rev_map f l =
  match l with
  | [] -> []
  | h :: t -> let t = rev_map f t in (f h) :: t;;
```

b. **(7 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]
```

Solution:

```
let rec rev_mapk f l k =
  match l with
  | [] -> k []
  | h :: t -> rev_mapk f t
      (fun t1 -> f h
        (fun t2 -> consk t2 t1 k));;
```

Problem 9: (10 points)

a. **(3 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])
```

Solution:

```
let rec partition l p =
  match l with
  | [] -> ([], [])
  | h :: t -> match (partition t p) with
    | (l1, l2) -> if (p h) then (h :: l1, l2) else (l1, h :: l2);;
```

b. **(7 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])
```

Solution:

```
let rec partitionk l p k =
  match l with
  | [] -> k ([], [])
  | h :: t -> partitionk t p
      (fun t -> match t with
        | (l1, l2) -> p h
            (fun t2 -> if t2 then (consk h l1 (fun t3 -> k (t3, l2)))
                              else (consk h l2 (fun t3 -> k (l1, t3)))));;
```