Generating all permutations

In this short note, we will look at some ways for generating all permutations in a lazy manner. This is currently very unstructured, but hopefully these code may be helpful.

1 Preliminaries

We will make use of the following basic functions for lazy lists:

```
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq);;

let rec to_list = function
  Nil -> []
| Cons(x, xs) -> x::to_list (xs());;

let get_tail (Cons(_, xs)) = xs ();;
```

2 Using approach 3 (method 1)

We take the steps outlined in approach 3 to convert the non-lazy code into lazy.

```
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq);;

let rec get n (Cons(l, ls)) =
  if n = 0 then []
  else l::(get (n-1) (ls()));;

let rec generate_permutations cur_perm rem_items next =
  if rem_items = [] then Cons(List.rev(cur_perm), next)
  else let rec try_next used_items rem nxt =
         if rem = [] then nxt()
        else generate_permutations (List.hd(rem)::cur_perm) (used_items @
            List.tl(rem))
                   (fun () -> try_next (List.hd(rem) :: used_items) (List.tl(rem))
                        (nxt))
      in try_next [] rem_items next;;


let perms = generate_permutations [] [1;2;3;4] (fun () -> Nil);;
```

Now we can test for correctness:

```
get 10 perms;;
(*
val x : int list list =
  [[1; 2; 3; 4]; [1; 2; 4; 3]; [1; 3; 2; 4]; [1; 3; 4; 2]; [1; 4; 3; 2];
   [1; 4; 2; 3]; [2; 1; 3; 4]; [2; 1; 4; 3]; [2; 3; 1; 4]; [2; 3; 4; 1];
   [2; 4; 3; 1]; [2; 4; 1; 3]; [3; 2; 1; 4]; [3; 2; 4; 1]; [3; 1; 2; 4];
   [3; 1; 4; 2]; [3; 4; 1; 2]; [3; 4; 2; 1]; [4; 3; 2; 1]; [4; 3; 1; 2];
   [4; 2; 3; 1]; [4; 2; 1; 3]; [4; 1; 2; 3]; [4; 1; 3; 2]]
*)
```

And we can also verify that in each call of `get` one new permutation is revealed.

```
(* Check that the implementation is indeed generating the permutations
   in a lazy way. *)
#trace generate_permutations;;
let get_tail (Cons(_, xs)) = xs ();;
let a = get_tail perms;;
let a = get_tail a;;
let a = get_tail a;;
```

```
(*
#trace generate_permutations;;
generate_permutations is now traced.
# let a = get_tail perms;;
generate_permutations <-- [<poly>; <poly>; <poly>]
generate_permutations --> <fun>
generate_permutations* <-- [<poly>]
generate_permutations* --> <fun>
generate_permutations** <-- <fun>
generate_permutations <-- [<poly>; <poly>; <poly>; <poly>]
generate_permutations --> <fun>
generate_permutations* <-- []
generate_permutations* --> <fun>
generate_permutations** <-- <fun>
generate_permutations** --> Cons ([<poly>; <poly>; <poly>; <poly>], <fun>)
generate_permutations** --> Cons ([<poly>; <poly>; <poly>; <poly>], <fun>)
val a : int list seq = Cons ([1; 2; 4; 3], <fun>)
# let a = get_tail a;;
generate_permutations <-- [<poly>; <poly>]
generate_permutations --> <fun>
generate_permutations* <-- [<poly>; <poly>]
generate_permutations* --> <fun>
generate_permutations** <-- <fun>
generate_permutations <-- [<poly>; <poly>; <poly>]
generate_permutations --> <fun>
generate_permutations* <-- [<poly>]
generate_permutations* --> <fun>
generate_permutations** <-- <fun>
generate_permutations <-- [<poly>; <poly>; <poly>; <poly>]
generate_permutations --> <fun>
generate_permutations* <-- []
generate_permutations* --> <fun>
generate_permutations** <-- <fun>
generate_permutations** --> Cons ([<poly>; <poly>; <poly>; <poly>], <fun>)
generate_permutations** --> Cons ([<poly>; <poly>; <poly>; <poly>], <fun>)
generate_permutations** --> Cons ([<poly>; <poly>; <poly>; <poly>], <fun>)
val a : int list seq = Cons ([1; 3; 2; 4], <fun>)
# let a = get_tail a;;
generate_permutations <-- [<poly>; <poly>; <poly>]
generate_permutations --> <fun>
generate_permutations* <-- [<poly>]
generate_permutations* --> <fun>
generate_permutations** <-- <fun>
generate_permutations <-- [<poly>; <poly>; <poly>; <poly>]
generate_permutations --> <fun>
generate_permutations* <-- []
generate_permutations* --> <fun>
generate_permutations** <-- <fun>
generate_permutations** --> Cons ([<poly>; <poly>; <poly>; <poly>], <fun>)
generate_permutations** --> Cons ([<poly>; <poly>; <poly>; <poly>], <fun>)
val a : int list seq = Cons ([1; 3; 4; 2], <fun>)
*)
```

3 Using approach 3 (method 2)

We start with the core part of gen_perms and then we proceed to fill in the missing parts of the implementation that are mostly standard lazy functions.

```
let rec gen_perms = function
| [] -> Cons([], fun () -> Nil)
(* Note that we need to convert rem_items to a lazy list to avoid
   executing all at once *)
```

```
| rem_items -> flatten (smap (fun x -> all_cons x (gen_perms (remove x rem_items)))
    (make_lazy rem_items));;
```

Note that we have essentially replaced the for-loop which was in the form of a continuation style and turned it into a lazy map application over the remaining elements[1].

```
let rec smap f = function
  Cons(x, xs) -> Cons(f x, fun () -> smap f (xs()))
| Nil -> Nil;;

let rec make_lazy = function
| [] -> Nil
| x::xs -> Cons(x, fun () -> make_lazy xs);;

(* Hardest part *)
let rec flatten xs = match xs with
| Nil -> Nil
| Cons(y, ys) -> match y with
   | Nil -> flatten(ys())
   | Cons(z, zs) -> Cons(z, fun () -> flatten (Cons(zs(), ys)));;

(* Simple function that removes an element from the list. *)
let rec remove x = function
| [] -> []
| y::ys ->
   if y = x then remove x ys (* or can terminate here *)
   else y::remove x ys;;

(* Appends the element to all lists. *)
let all_cons x ys = smap (fun y -> x::y) ys;;
```

We can verify correctness and laziness using the same code as above:

```
let perms = gen_perms [1;2;3;4];;

let x = to_list perms;;

#trace gen_perms;;
let a = get_tail perms;;
let a = get_tail a;;
let a = get_tail a;;
let a = get_tail a;;
```

4 Using approach 1

The `get_next()` function is a bit complicated for permutations. To get the next permutation (e.g. of [0;1;3;6;5;4;2]), we need to:

1. Find the longest suffix that is decreasing (e.g. [6;5;4;2] and the rest are [0;1;3]).
2. Find the element x just after this suffix (e.g. 3 in this case).
3. Swap x with the element y just greater than it in the suffix (e.g. $y = 4$ and [6;5;3;2]).
4. Reverse the suffix and append it to the modified prefix (e.g. [0;1;4] @ [2;3;5;6]).

We implement these functions as follows:

```
let rec get_sorted ls acc = match ls with
  [] -> acc, []
| [x] -> [x], []
| x::y::xs -> if x > y then x::acc, (y::xs)
              else get_sorted (y::xs) (x::acc);;
```

[1]Maybe it is a bit better to combine the `smap` with `make_lazy` function.

```
get_sorted (List.rev [0;1;3;9;8;7;2]) [];;

let find_just_greater ls x =
  List.fold_left min (max_int) (List.filter (fun y -> y > x) ls);;

let rec find_and_replace_just_greater ls x acc = match ls with
  [y] -> List.rev acc @ [x], y
| y::ys -> if y > x then List.rev acc @ x::ys, y
           else find_and_replace_just_greater ys x (y::acc);;

find_and_replace_just_greater [1;2;7;8;9] 3 [];;
```

An this allows us to define the `get_next()` function:

```
let get_next perm =
   let suf, p::pref = get_sorted (List.rev perm) [] in
   let new_suf, c = find_and_replace_just_greater (List.rev suf) p [] in
      List.rev pref @ c::new_suf;;
```

Now putting it all together:

```
(* Check if this is the last permutation. *)
let rec is_last = function
  [x] -> true
| x::y::xs -> if x < y then false
              else is_last (y::xs);;

let rec get_permutations cur =
   Cons(cur, fun () ->
      if is_last cur then Nil
      else get_permutations (get_next cur));;
```

And so, we can perform similar correctness and laziness checks as before:

```
List.length (to_list (get_permutations [1;2;3;4])) = 24;;
List.length (to_list (get_permutations [1;2;3;4;5])) = 120;;

#trace get_permutations;;
let get_tail (Cons(_, xs)) = xs ();;
let a = get_permutations [1;2;3;4];;
let a = get_tail a;;
let a = get_tail a;;
let a = get_tail a;;
let a = get_tail a;;
```