OCaml: Tail Recursion

Jeff Meister

CSE 130, Winter 2011

All that's necessary for a function to be tail-recursive is that any time it makes a recursive call, the resulting value is immediately returned (no further computation is performed on it by the recursive caller).

For example, the following function is tail-recursive:

```
# let rec for_all p l = match l with
  | [] -> true
  | h :: t -> if p h then for_all p t else false;;
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
```

It takes two parameters, a predicate `p` (function from `'a` to `bool`) and a `list` of `'a`s, and it returns `true` if and only if the predicate returns `true` on every `'a` in the list. There's one recursive call (`forall p t`), and since the result is immediately returned as the value of `forall p l`, this function is tail-recursive.

However, the following function is not tail-recursive:

```
# let rec map f l = match l with
  | [] -> []
  | h :: t -> f h :: map f t;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

It takes two parameters, a function from `'a` to `'b` and a `list` of `'a`s, and it returns the `list` of `'b`s obtained by applying the function to each element in turn. Notice how the result of the recursive call `map f t` is not immediately returned; we have to `::` it with the result of `f h` to get the value of `map f l`. So, the previous stack frame has to wait around for each recursive call to complete before being able to do its work, and this function is not tail-recursive.

To show off the power of tail-recursion (alternatively, demonstrate the failure of a lack thereof), the notes posted along with PA2 call their example summation functions on a large number. That's easy to do when working with numbers, since you can just write a large number like `10000000` literally in the toplevel, but we're working with lists in this post, and I certainly don't want to write down a list with 10000000 elements. However, such repetition is a perfect task for the computer, so I'll write a function to generate the large list for me. Here's an attempt:

```
# let rec make_list n = if n = 0 then [] else n :: make_list (n - 1);;
val make_list : int -> int list = <fun>
# let big_list = make_list 10000000;;
Stack overflow during evaluation (looping recursion?).
```

What went wrong here? Well, my `make_list` function was itself not tail-recursive! Here you can see where an accumulator might come in handy to transform the function into one that is tail-recursive. The function is trying to build (accumulate) a large list to return as its final result, but the strategy of adding a stack frame with a cons until we reach the empty list is naïve. Here's a better version where we do the accumulating in a parameter:

```
# let rec make_list n l = if n = 0 then l else make_list (n - 1) (n :: l);;
val make_list : int -> int list -> int list = <fun>
```

Now it needs an extra argument for the initial value of the accumulator, but that's not so bad, and it works:

```
# let big_list = make_list 10000000 [];;
val big_list : int list =
  [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21;
   22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36; 37; 38; 39;
   40; 41; 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; ...]
```

Now we can test `for_all` and `map` on this large input:

```
# for_all (fun x -> x > 0) big_list;; (* tail-recursive *)
- : bool = true
# map succ big_list;; (* not tail-recursive *)
Stack overflow during evaluation (looping recursion?).
```

Now, the astute reader may have noticed that, although `map` is not tail-recursive, perhaps we could make it so. Like `make_list`, it builds a large output list by cons-ing some value to the front of the list it gets AFTER making a recursive call. How about we transform it to do the cons BEFORE the recursive call, so we can then return immediately?

```
# let rec tailmap f l a = match l with
  | [] -> a
  | h :: t -> tailmap f t (f h :: a);;
val tailmap : ('a -> 'b) -> 'a list -> 'b list -> 'b list = <fun>
# tailmap succ big_list [];;
- : int list =
[10000001; 10000000; 9999999; 9999998; 9999997; 9999996; 9999995; 9999994;
 9999993; 9999992; 9999991; 9999990; 9999989; 9999988; 9999987; 9999986;
 9999985; 9999984; 9999983; 9999982; 9999981; 9999980; 9999979; 9999978;
 9999977; 9999976; 9999975; 9999974; 9999973; 9999972; 9999971; 9999970; ...]
```

Well, it worked, but the value wasn't quite what we expected... the successor list comes out backwards! In fact, this behavior makes sense if you think about it; reread the previous pragraph and it should be more clear why I emphasized after vs. before. Basically, we first tried to do things in one order, and that would have worked if not for the stack overflow... but when we fixed that, we flipped the order around. You may need to stare at the definitions of `map` and `tailmap` for a while to see this.

We can fix our `tailmap` function using a friend from PA1, `listReverse`. If we just call that on the output of `tailmap`, we should get the right answer. However, it's likely that the version of `listReverse` you wrote is not tail-recursive, so it'll overflow the stack processing the huge output of `tailmap`. Like so:

```
# let rec listReverse l = match l with
  | [] -> []
  | h :: t -> (listReverse t) @ [h];;
val listReverse : 'a list -> 'a list = <fun>
# listReverse (tailmap succ big_list []);;
Stack overflow during evaluation (looping recursion?).
```

Fortunately, there is a less obvious implementation of list reversal that is tail-recursive. It may be useful as an exercise for you to try implementing this yourself. But I'm not going to leave you hanging, here it is:

```
# let rev l =
    let rec helper l a = match l with
      | [] -> a
      | h :: t -> helper t (h :: a)
    in helper l [];;
val rev : 'a list -> 'a list = <fun>
```

I will leave it to you to think about why this works, though. But it does, and we can use it to finally define a version of `map` that works properly on our huge list:

```
# let efficient_map f l = rev (tailmap f l []);;
val efficient_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# efficient_map succ big_list;;
- : int list =
[2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21; 22;
 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36; 37; 38; 39; 40; 41;
 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60;
 61; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78; 79; ...]
```