OCaml: Fold, with Tail Recursion Review

Jeff Meister

CSE 130, Winter 2011

1 Tail recursion

Let's review some simple recursive functions on lists, as we've seen.

To find the length of an `'a list` (a list of any type), we can recursively walk over the list and count `1` for each element we find:

```
let rec len_list l = match l with
  | [] -> 0
  | _ :: t -> 1 + len_list t
```

To find the sum of a `float list`, we recursively walk over the list and count `h` for each element `h`:

```
let rec sum_list l = match l with
  | [] -> 0.
  | h :: t -> h +. sum_list t
```

To find the max of a positive `int list`, we recursively walk over the list and keep track of the greatest element:

```
let rec max_list l = match l with
  | [] -> 0
  | h :: t -> max h (max_list t)
```

Here's a PA3 freebie... `clone z n` builds a list of length `n` whose elements are all `z`:

```
let rec clone z n =
  if n <= 0 then []
  else z :: clone z (n - 1)
```

OK, we have some list functions, and they work on a small list:

```
# let five_nines = clone 9 5;;
val five_nines : int list = [9; 9; 9; 9; 9]
# len_list five_nines;;
- : int = 5
# sum_list (clone 9. 5);;
- : float = 45.
# max_list five_nines;;
- : int = 9
```

But observe that none of these functions are tail recursive, so they won't work on a large enough input list. To get such a list, we have to fix `clone` first, as the non-tail-recursive version can't help us:

```
# let a_million_twos = clone 2 1000000;;
Stack overflow during evaluation (looping recursion?).
```

To fix it, instead of accumulating the cloned `z` in a stack frame every time, we write a helper function that does it in a parameter instead.

```
let clone_tr z n =
  let rec helper a n =
    if n <= 0 then a
    else helper (z :: a) (n - 1)
  in helper [] n
```

Now we can use `clone` to get a large input list, but the other list functions aren't tail recursive, so they can't handle it:

```
# len_list (clone_tr 2 1000000);;
Stack overflow during evaluation (looping recursion?).
# sum_list (clone_tr 4. 1000000);;
Stack overflow during evaluation (looping recursion?).
# max_list (clone_tr 5 1000000);;
Stack overflow during evaluation (looping recursion?).
```

So, let's modify our other functions to be tail recursive in a similar way, by introducing a helper function with an accumulator parameter.

```
let len_list_tr l =
  let rec helper a l = match l with
    | [] -> a
    | _ :: t -> helper (a + 1) t
  in helper 0 l

let sum_list_tr l =
  let rec helper a l = match l with
    | [] -> a
    | h :: t -> helper (a +. h) t
  in helper 0. l

let max_list_tr l =
  let rec helper a l = match l with
    | [] -> a
    | h :: t -> helper (max a h) t
  in helper 0 l
```

Cool, these work:

```
# len_list_tr (clone_tr 2 1000000);;
- : int = 1000000
# sum_list_tr (clone_tr 4. 1000000);;
- : float = 4000000.
# max_list_tr (clone_tr 5 1000000);;
- : int = 5
```

2 Fold

Take a look at the three list functions above: `len_list_tr`, `sum_list_tr`, and `max_list_tr`. They look really similar, huh? In fact, only two things are varying, aside from the names. Function `len` adds `1` to its accumulator (to count the number of elements), while `sum` adds `h` (to count the contents of the elements), and `max` finds the current maximum. The initial value for the accumulator also varies; `sum` needs a floating-point `0.` because it's adding up `float`s with the `+.` operator. In all three functions we're following a pattern: take a list and some initial accumulator value, then walk over the list, and for every element seen, *do something* with the current value of the accumulator and the element to get a new value for the accumulator. Finally, continue iterating over the rest of the list (via recursive call, of course, this is OCaml after all), doing the same thing, but passing along the new value of the accumulator. When we reach the end of the list, the final accumulator value is returned as the result.

But for my three similar functions, I had to write three similar scraps of boilerplate code. There's a better way! In functional programming, we represent "*do something* with the accumulator and the element" as "*call a function* on the accumulator and the element", and since functions are first-class values, we can just make our *something-doer* a parameter to a more generic list-walking function. This function is called `fold`.

```
let rec fold_left_list f a l = match l with
  | [] -> a
  | h :: t -> fold_left_list f (f a h) t
```

Simple but powerful, `fold_left_list` takes three parameters: a function `f`, an initial value for the accumulator `a`, and a list `l` to process. For any element we encounter in the list, we call `f` on the current value of the accumulator and the element, then use the result as the new accumulator to continue folding over the rest of the list. Examine the type of this function, and make sure you understand it!

```
val fold_left_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

We can use this "generic list visitor" to simplify our three list functions greatly:

```
let len_list_tr' l = fold_left_list (fun a _ -> a + 1) 0 l
let sum_list_tr' l = fold_left_list ( +. ) 0. l
let max_list_tr' l = fold_left_list max 0 l
```

They now become one-liners. Each time, we just pass in a function that "does the right thing" with the accumulator and the element. `fold_left_list` takes care of making sure our function gets called on every element in the list, starting from the left (hence its name).

2.1 Generalizing fold

A slight digression: the idea of folding can be generalized to handle more than just lists. Notice how `clone_tr` looks quite similar to `len_list_tr` and crew, but we can't implement it with `fold_left_list`, because it's iterating over a sequence of natural numbers, namely the descending sequence $n, n-1, n-2, \ldots, 0$. (We could put the numbers in a list, but let's suppose that would occupy too much memory. Plus, how would we get them there? With another function just like clone?) However, we can write a `fold` function for such number sequences:

```
let rec fold_desc_nat f a n =
  if n <= 0 then a
  else fold_desc_nat f (f a n) (n - 1)
```

Now `clone_tr` need only pass a function to `fold_desc_nat` to stick `z` in the accumulator list every time. `fold_desc_nat` takes care of calling the function on the descending sequence starting from `n` and accumulating the `z`s each time.

```
let clone_tr' z n = fold_desc_nat (fun a _ -> z :: a) [] n
```

This is sort of a degenerate case, since usually we fold over a data structure, and our function cares about the elements inside. But we'll see more interesting generalizations of `fold` later on.

2.2 Insertion sort as a fold

Back to folding on lists, here's a slightly more interesting application, the insertion sort. Look it up if you forget what it does, but basically you build a sorted result list by taking an element from the unsorted one and inserting it into its proper place in the result list, which starts out empty. This is how most normal humans would sort a deck of playing cards (and even many computer scientists, who are familiar with more efficient sorting algorithms). The key operation is the function `insert`, which takes an already-sorted list `l` and an element `e`, and inserts `e` into the appropriate place in `l`:

```
let rec insert l e = match l with
  | [] -> [e]
  | h :: t -> if e > h then h :: insert t e else e :: l
```

Look how elegant! If `l` is empty, then we know how to insert `e` in there, it's just `[e]`. Otherwise, we have to check if we've found the right place for `e` (that is, is `e` finally less than the current number we're looking at?). If we haven't, then we continue looking through the rest of the list for `e`'s spot, making sure to keep the element `h < e` before it. But if we found the first element `h` that is bigger than `e`, then we know where to put `e`: right before `h`. The list `l` still has `h` at the front (remember, our pattern-matching did not change it), so we just write `e :: l`. Easy!

```
let insertion_sort l = fold_left_list insert [] l
```

Insertion sort thus becomes trivial: for each element of the input list, we insert it into its appropriate place in the accumulator list, which starts out empty. An iterative implementation would have a loop inside a loop; here we have recursion inside recursion. The function `fold_left_list` is recursively walking over an input list with some length n, and at each step it calls the function `insert`, which recursively walks over the result so far, whose length is a linear function of n. Therefore, the sort overall takes $O(n^2)$ steps. Also note that `insert` is not tail recursive. But if you're working with lists large enough for that to matter, you shouldn't be using insertion sort anyway!