Recitation 3: Mapping, Folding and More List Functions

CS3110 Fall 2013

1 Lecture Outline

1. Basic Folding
2. Implementing `List` Functions Using Folding
3. Implementing Folding on a Custom List

2 Basic Folding

Folding is one of the most powerful features of the OCaml language and functional programming in general. As you know, the concept of a for-/while-loop iterating over any data structure is unbeknownst to the paradigm of functional programming. In OCaml, what we use are the two folding functions `List.fold_left` and `List.fold_right`. In a sentence, these two functions iterate over a list, the former starting from the left and going toward the right and the latter from right to left, applying a given function `f` to each element and storing the value successively in a variable known as the **accumulator**. After iterating over the last element, the entire folding expression evaluates to the final value of the accumulator. Here is a simple example using folding to sum all the elements of a list:

```
let sum (lst : int list) =
  let fold_function (acc : int) (elem : int) : int =
    acc + elem
  in
  List.fold_left fold_function 0 lst
```

How this `sum` function works is by applying `fold_function` to the current accumulator and the current element of the list we are iterating over. To test it out,

```
sum [1;2;3;4];;
- : int = 10
```

Here is a breakdown of the iterations:

Iteration	`acc`	`elem`
0	0	1
1	1	2
2	3	3
3	6	4
4	10	–

Since the final value of the accumulator is 10, that is what the entire expression evaluates to, and the toplevel returns us this value.

As another example, this function will determine the length of a list:

```
let length (lst : 'a list) : int =
  let f (acc : int) _ =
    1 + acc
  in
  List.fold_left f 0 lst
```

Notice that the second parameter of the folding function is simply _, the wildcard character that you have seen in `match` statements. In that context, it represents an entity that exists but one that we don't care enough about to actually bind to a variable, and that is exactly the same idea when using a wildcard as a function argument. When writing a function to compute the

length of a list, we don't really care what the particular element is - we just want to acknowledge its existence and move on.

```
length ["zardoz"; "cs3110"];;
- : int = 2
```

As a more complicated example, we are going to write a function that will sum the values of an `int option list` and return the value as an `int option`. If any of the elements in the list is `None`, then we want the whole output to be `None`.

```
(*Will sum up all the values of an int option list
and return as option, returning None if any of the
elements is None*)
let sum_ops (lst : int option list) : int option =
  let f acc elem =
    match acc,elem with
    | (None, _)
    | (_, None) -> None
    | (Some acc', Some v) -> Some(acc' + v)
  in
  List.fold_left f (Some 0) lst
```

Test this function out in a toplevel to see that it matches its specification.

3 Implementing `List` Functions Using Folding

The true power of folding can be seen when we use it to implement the functions in the `List` module. To start off, let's write our own map function:

```
let map (f : 'a -> 'b) (lst : 'a list) : 'b list =
  let fold_function (elem : 'a) (acc : 'b list) =
    (f elem)::acc
  in
  List.fold_right fold_function lst []
```

This is the first example which uses `List.fold_right`. There are situations (like this one) where it is advantageous to go in a particular direction. Since the folding function is using the cons (::) operator, we need to traverse the original list starting from the right so that we maintain the proper order of the elements. Also, the order of the parameters for `fold_function` is different since we are folding from right to left.

You should be familiar with folding enough to see how this function works. The initial accumulator is an empty list and we grow it element-by-element, each time prepending the mapped value of the element we are currently processing from the original list.

Now we will take a look at a new `List` module function: `List.partition`. It takes in a predicate (function) of type $'a \rightarrow bool$ and a list and returns a 2-tuple - the first element is the portion of the list that causes the function to evaluate to `true` and the second is the portion of the list that causes the function to evaluate to `false`. To see it in action, let's rewrite the `sum_ops` function from the previous section:

```
let sum_ops_partition (lst : int option list)=
  let (nones,somes)=List.partition(fun x-> x=None) lst in
  if List.length nones > 0 then
    None
  else
    (*option_apply from recitation 1*)
    let option_apply f a b =
      match a,b with
      | Some(a'), Some(b') -> Some (f a' b')
      | _,_ -> failwith "should never fail"
    in
    List.fold_left (option_apply (+)) (Some 0) somes
```

The way this function differs from the first version is that it only folds if there are no `None` values, otherwise it will evaluate to `None` directly.

Since this section is dedicated to implementing list functions, let's use folding to write our own partition function:

```
let partition pred lst =
  let f elem (trues, falses) =
    if pred elem then
      (elem::trues, falses)
    else
      (trues, elem::falses)
  in
  List.fold_right f lst ([],[])
```

The key point of this example is that our accumulator consists of a 2-tuple. The main advantage to using a tuple as an accumulator is that **you can keep track of all the information you want while folding, not just one entity**. In this partition example, we want to add the current element to one of two lists, which means that we need to keep track of **both** lists at all times. But OCaml only lets us have one accumulator, so the solution is to package both of them into a tuple and then use pattern matching to modify only one of its components. Since the fold expression evaluates to the type of the accumulator, our partition function ultimately returns a tuple of 2 lists.

Using Accumulators Outside of Folding

So far, we have only seen accumulators in the context of folding. However, they play a useful role in recursive functions too. Consider this version of a function that adds all the elements of a list together:

```
let sum_list lst =
  let rec sum_helper acc lst =
    match lst with
    | [] -> acc
    (*tail-recursion!*)
    | h::t -> sum_helper (acc + h) t
  in
  sum_helper 0 lst
```

In the "normal" recursive function, we would ordinarily include the line `h + sum t`, but in this version, we add `h` to the accumulator as a parameter before actually making the call to `sum`. Moreover, **the call to `sum` is the last thing done recursively**. This concept is known as **tail-recursion**.

In the old version, the program would have to keep track of each value of `h` on the stack frame of each call, and add them up once the base case was reached. In the accumulator version, this addition takes place before the recursive call, which is the last thing done by the function, meaning we do not need to preserve the old stack frame. This is better for the computer, and we should always try to make our functions tail-recursive if possible.

4 Implementing Folding on a Custom List

To really understand folding, it helps to write our own folding functions. Let's define our own list type too while we're at it, just to not mix it up with the default OCaml list:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist

let rec fold_left f acc lst =
  match lst with
  | Nil -> acc
  | Cons (h,t) -> fold_left f (f acc h) t

let rec fold_right f lst acc =
  match lst with
  | Nil -> acc
  | Cons (h,t) -> f h (fold_right f t acc)
```

These are the two folding functions for our custom list datatype. These implementations also reveal the "directionality" of the functions. For instance, `fold_left` apply `f` on `h` first before moving on with the folding, implying that the function gets applied from beginning-to-end. It is also worth noting that this function is tail-recursive. All necessary pre-computations are performed in the parameters, and the recursive call to `fold_left` is the last thing done. Conversely, in `fold_right`, the function `f` acts on the tail first, and the head `h` is technically last in the computation, meaning that it operates from right-to-left. Additionally, this function is *not* tail-recursive since there are more computations that need to be done after the recursive call. The tail-recursiveness of our folding functions are also reflected in their `List` module analogues.