Motivation - Why do maps and folds exist?

Lists are one of the fundamental data structures in Ocaml (and in functional programming), so we need to generate tools to process them. Historically, list recursion takes one of two shapes:

1. Do something to every element
2. Combine all elements of a list into a single result

These patterns appear so frequently that the earliest functional languages (like Lisp (1959)) abstracted them into reusable higher-order functions.

- Lisp created *mapcar* which can be considered the "ancestor" of today's *map.*
    - "*Mapcar applies a function to each element of a list, producing a list of results"* (Wikipedia - early Lisp Manual)
- The Haskell/ml community formalized *fold* as the canonical way to consume a recursive structure.

During the 1970s and 80s, researchers in ML languages refined ideas around pattern matching and structured recursion. While *map* captured the "apply to all" pattern, researchers noticed that many recursive list functions also shared a second essential pattern: replace each constructor of the list with another operation. This led to the formalization of folds.

Graham Hutton succinctly expressed the underlying motivation behind folds in his seminal paper *A Tutorial on the Universality and Expressiveness of Fold* (1998), writing "Folds capture the very essence of structural recursion."

Hutton and others demonstrated that almost every common list function (like sum, length, maximum, etc) can be expressed as a fold. As such, folds generalized the idea of structurally recursing over a data type by replacing the base case ([]) and the recursive constructor (::) with user specified behaviors. Bernie Pope emphasized the pedagogical value of folds: "*Using folds simplifies many definitions, makes common recursion schemes explicit, and avoids errors."*

Definitions

A map is a function that takes another function and a list as arguments and returns that function applied to the list. It can be considered an "apply-to-all".

A fold is also a function that takes another function and a list as arguments, combines the results of applying the function to each of the elements of the list recursively, and then returns that result. Unlike maps, folds can be applied either **left-to-right** or **right-to-left**, and the direction can change the meaning or even the final value depending on the function used.

The key difference between deciding when to use maps vs folds is the following: if you have a **one-to-one** relationship between each element of your input list and each element of your output (with some transformation on that element), then you would use a map. However, if you

have a **many-to-one** relationship where many values of your input list correspond to a singular return value in your output list, you would use a fold.

Below, we highlight the intuition behind the usefulness of map, before subsequently describing the intuition behind folds.

Start with a normal recursive function:

```
let rec add_one l =
match l with
| [] -> []
| hd :: tl -> (hd + 1) :: add_one tl;;
```

This has the form transform head :: recurse on tail.

The transformation (hd + 1) is the *only* piece that changes. So we can abstract it:

```
let rec map f l =
match l with
| [] -> []
| hd :: tl -> f hd :: map f tl;;
```

Therefore:

```
let add_one l = map (fun x -> x + 1) l;;
```

This example demonstrates how **map** abstracts “do something to every element in list l.”

But if, instead, you want something to return *one* value (ie the length of a list, the sum of all elements of a list, the maximum element of a list, etc.), we can capture this with **fold.**

Folds represent the universal pattern of structural recursion on lists. In fact, in Haskell, folds are called *catamorphisms* because they reduce the structure downward into a single value.

The function definitions of both fold_left and fold_right take in a function, a list, and an accumulator. Before diving into the code of fold_right and fold_left, it is important to highlight what each fold *does* with these inputs as it walks through a list.

Suppose we have the list:

```
[1; 2; 3; 4]
```

- With fold_left, given accumulator *acc* and function *f,* you would move left to right. At each step, you update the accumulator:

  ```
  acc --f--> 1 --f--> 2 --f--> 3 --f--> 4 --> result
  ```

  Mathematically, this is represented by ((((acc f 1) f 2) f 3) f 4).

- With fold_right, on the other hand, given accumulator *acc* and function *f,* you would move right to left. At each step, you update the accumulator:

  ```
  acc --f--> 4 --f--> 3 --f--> 2 --f--> 1 --> result
  ```

  Mathematically, this is represented by (1 f (2 f (3 f (4 f acc)))).

Below is the definition of fold_right:

```
let rec fold_right f l acc =
match l with
| [] -> acc
| hd :: tl -> f hd (fold_right f tl acc);;
```

Conceptually, we are recursively folding the tail and then combining the head afterwards. However, as you can see, fold_right is not tail recursive, which poses the risk of stack overflow on long lists. What makes fold_right useful (in comparison to fold_left) is that it matches induction exactly, where the base case is the empty list ([]) and each step is represented by hd :: tl -> f hd (fold_right f tl acc).

Fold_left is defined as so:

```
let rec fold_left f acc l =
match l with
| [] -> acc
| hd::tl -> fold_left f (f acc hd) tl;;
```

In many ways fold_left feels more “natural” because it processes the list left-to-right in the same order that humans typically read and reason about sequences. The accumulator is updated progressively as we traverse the list.

By contrast, fold_right aligns with the structural view of lists: it processes the list by first folding the tail and then combining the head, which mirrors mathematical induction and recursive definitions.

Ultimately, there are **three** principal differences between fold_left and fold_right.

1) One is that, by definition, fold_left is tail recursive while fold_right is not. This can be seen in the code of fold_left and fold_right above.
2) The second difference is the order of the inputs. Fold_left takes in the operation, then the accumulator, followed by the list. Fold_right takes in the operation, then the list, followed by the accumulator.
3) The third major difference is that, when using non-associative operations (such as subtraction, exponentiation, etc), or other operations where order matters (such as string concatenation), the output of fold_left and fold_right will be different due to the order in which inputs are processed. Below is an example of this differentiation in output (when using the non-associative subtraction operator)

- Let's say you call

  ```
  fold_left (-) 0 [1; 2; 3; 4]
  ```

  This would be processed as ((((0 - 1) - 2) -3) -4) = -10

- Now, let's say you call

  ```
  fold_right (-) [1; 2; 3; 4] 0
  ```

  This would be processed as (1 - (2 - (3 - (4 - 0)))) = -2

Generalizing Maps and Folds Beyond Lists

While lists are the simplest recursive data type, the ideas behind map and fold extend naturally to trees, options, and, more generally, any algebraic data type.

1. Maps and folds on trees:

   Consider the type of a binary tree:

   ```
   type 'a tree =
     | Leaf
     | Node of 'a * 'a tree * 'a tree
   ```

   Tree **map** applies a function to every node, structurally mirroring the shape of the tree:

   ```
   let rec tree_map f t =
    match t with
    | Leaf -> Leaf
    | Node (x, tl, tr) -> Node (f x, tree_map f tl, tree_map f tr)
   ```

   This directly mirrors the behavior of list map, but on trees. The logic is, take each element of a tree, apply some function f to it, and return the resultant tree.

   Now, let's explore tree fold. Tree fold takes in an accumulator, some function that combines elements of the tree to a single accumulator value, and a tree. This parallels the function signature of

   ```
   let rec fold_tree acc f t =
    match t with
    | Leaf -> acc
    | Node (x, tl, tr) -> f x (fold_tree acc f tl) (fold_tree acc f tr)
   ```

2. Maps and folds on options:

   Consider the type of option:

```
type 'a option =
 | None
 | Some of 'a
```

Option map can then be written as:

```
let map_option f opt =
 match opt with
 | Some v -> Some (f v)
 | None -> None
```

This captures the same underlying idea as list map: apply a function f to the data *inside* the structure, if it exists, while preserving the shape of that structure. For lists (or trees), map applies f to each element. For options, there is either zero or one element to apply f to.

Option fold can be written as:

```
let fold_option f acc opt =
 match opt with
 | None -> acc
 | Some v -> f v
```

This once again mirrors the idea of list folds: just like fold_right replaces the list constructors [] and (::) with a base value and some combining function, fold_option replaces the constructors None and Some with the accumulator acc and the function f, respectively.

Generalized maps and folds illustrate that recursion patterns do not need to be re-invented for each new data type.

Sources:

Diller, Antoni. *Higher-Order Functions in Haskell, Unit 6.* https://www.cantab.net/users/antoni.diller/haskell/units/unit06.html

Hutton, Graham. *A Tutorial on the Universality and Expressiveness of Fold.* University of Nottingham. https://urldefense.com/v3/__https://people.cs.nott.ac.uk/pszgmh/fold.pdf__;!!LIr3w8kk_Xxm!v-bsWZcIg3qp3XnqTBDz2kDY35SxebkTUFeILl0AGQ1BN1Lcji9FFykUa3IF2XXu90qeGWdFHhyTWD8I$

Odersky, Martin. *Scala Mailing List Discussion on foldLeft and foldRight.* (Archived) https://web.archive.org/web/20150514122827/http://permalink.gmane.org/gmane.comp.lang.scala/9557

Pope, Bernie. "Functional Programming with Folds." *The Monad.Reader*, Issue 6. https://wiki.haskell.org/wikiupload/1/14/TMR-Issue6.pdf

Yorgey, Brent. "Stacks and Queues." *The Math Less Traveled (Blog)*, 2024. https://byorgey.github.io/blog/posts/2024/11/27/stacks-queues.html

Wikipedia. "Fold (Higher-Order Function)." https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Code Examples/Documentation:
https://github.com/shriya-upadhyay/MapsAndFolds