# More OCaml Algorithms, Methods & Diversions

John Whitington

### MORE OCAML

Algorithms, Methods & Diversions

John Whitington

COHERENT PRESS

#### COHERENT PRESS

Cambridge

Published in the United Kingdom by Coherent Press, Cambridge

© Coherent Press 2014

This publication is in copyright. Subject to statutory exception no reproduction of any part may take place without the written permission of Coherent Press.

First published August 2014 Reprinted with corrections July 2015 Reprinted 2016 Updated for OCaml language changes October 2017

A catalogue record for this book is available from the British Library

ISBN 978-0-9576711-1-9 Paperback

by the same author

PDF Explained (O'Reilly, 2012) OCaml from the Very Beginning (Coherent, 2013) A Machine Made this Book: Ten Sketches of Computer Science (Coherent, 2016)

## Chapter 1 Unravelling "Fold"

The List module in OCaml's Standard Library defines two intriguingly-named functions over lists:

```
fold_left : (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta list \rightarrow \alpha
fold_right : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha list \rightarrow \beta \rightarrow \beta
let rec fold_left f a l =
match l with
[] -> a
| h::t -> fold_left f (f a h) t
let rec fold_right f l a =
match l with
[] -> a
| h::t -> f h (fold_right f t a)
```

What do they do? And why are they considered important enough to include in the Standard Library? As we shall see, they abstract the idea of recursion over lists with an accumulator in a most delightfully generic way.

Let us first examine fold\_left and its rather complicated type. The first argument is itself a function, which takes the existing accumulator and an element from the input list, combines them in some fashion, and returns a new accumulator, ready for the next element. So, in general, the first argument has type  $\alpha \rightarrow \beta \rightarrow \alpha$ . Then we have an initial value for the accumulator, which must have type  $\alpha$ , and an input list of type  $\beta$  list. The return value is the final accumulator, so that must have type  $\alpha$ . We can annotate the function as follows:

 We can find the sum of a list of numbers:

$$\begin{array}{rcl} & \mbox{fold\_left} ( \ + \ ) \ 0 \ [1; \ 2; \ 3] \\ \implies & \mbox{fold\_left} \ ( \ + \ ) \ 1 \ [2; \ 3] \\ \implies & \mbox{fold\_left} \ ( \ + \ ) \ 3 \ [3] \\ \implies & \mbox{fold\_left} \ ( \ + \ ) \ 6 \ [] \\ \implies & \ 6 \end{array}$$

Here,  $\alpha$  and  $\beta$  are both **int**. The function (+) has the right type, and we use for the initial accumulator the *identity element* for (+) which is 0 since for all x, x + 0 = x (we cannot take the initial accumulator from the list itself since our function must have a result for the sum of all the integers in an empty list.)

It might appear to the reader that this is more complicated than the simple recursive solution, but to the experienced functional programmer, using fold\_left is in fact *easier* to read. Let us find the maximum number in a list using fold\_left:

```
\begin{array}{rcl} & \mbox{fold\_left max min\_int [2; 4; 6; 0; 1]} \\ \implies & \mbox{fold\_left max 2 [4; 6; 0; 1]} \\ \implies & \mbox{fold\_left max 4 [6; 0; 1]} \\ \implies & \mbox{fold\_left max 6 [0; 1]} \\ \implies & \mbox{fold\_left max 6 [1]} \\ \implies & \mbox{fold\_left max 6 []} \\ \implies & \mbox{6} \end{array}
```

Here max is the built-in function for finding the larger of two things, and min\_int is the built-in value of the smallest possible integer. We can use a similar scheme to define functions on lists of booleans:

all : bool list  $\rightarrow$  bool any : bool list  $\rightarrow$  bool let all l = fold\_left ( && ) true l let any l = fold\_left ( || ) false l

The all function is true if and only if all items in the list are true; the any function if at least one is. What can happen when  $\alpha$  and  $\beta$  are different? How about making the accumulator a list too? We can use List.mem to turn an arbitrary list into a set by consulting the existing accumulator before putting an element in:

setify :  $\alpha$  list  $\rightarrow \alpha$  list let setify l = fold\_left (fun a e -> if List.mem e a then a else e :: a) [] l

We are using List.mem to decide whether to add each element to the accumulator or discard it.

#### What about fold\_right?

Here is the function again:

fold\_right :  $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha$  list  $\rightarrow \beta \rightarrow \beta$ let rec fold\_right f l a = match l with [] -> a | h::t -> f h (fold\_right f t a)

The fold\_left function applied the given function over the elements in the input list from the left hand side. In contrast, fold\_right processes them from the right, by changing the evaluation order. Consider, for example, our summation example:

```
fold_left ( + ) 0 [1; 2; 3]
                    fold_left ( + ) 1 [2; 3]
             \implies
             \implies
                    fold_left ( + ) 3 [3]
             \Rightarrow
                    fold_left ( + ) 6 []
                    6
             \implies
       fold_right ( + ) [1; 2; 3] 0
       (+) 1 (fold_right (+) [2; 3] 0)
\implies
       (+) 1 ((+) 2 (fold_right (+) [3] 0))
\implies
       (+) 1 ((+) 2 ((+) 3 (fold_right (+) [] 0)))
\implies
       (+) 1 ((+) 2 ((+) 3 0))
\implies
       (+) 1 ((+) 2 3)
\implies
       (+)15
\implies
       6
\implies
```

See how the accumulating of values starts from the right hand side. Note also that fold\_right is not tail-recursive (the intermediate expression it builds is proportional to the size of the input). We can define map simply as a use of fold\_right.

map :  $(\alpha \rightarrow \beta) \rightarrow \alpha$  list  $\rightarrow \beta$  list let map f l = fold\_right (fun e a -> f e :: a) l []

Who would have thought that fold\_right was the more fundamental function? At the cost of a list reversal, we can make fold\_right tail-recursive by defining it in terms of fold\_left:

```
fold_right : (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha list \rightarrow \beta \rightarrow \beta
let fold_right f l e =
fold_left (fun x y -> f y x) e (List.rev l)
```

Sometimes we want to provide an initial accumulator value which is not the identity element for the computation. For example, applying :: over an input list with fold\_right is not very interesting, yielding a function which returns a copy of its input:

copy :  $\alpha$  list  $\rightarrow \alpha$  list let copy l = fold\_right (fun e a -> e :: a) l []

But if we supply a non-empty list as the initial value of the accumulator, we have the append function:

append :  $\alpha$  list  $\rightarrow \alpha$  list  $\rightarrow \alpha$  list let append x y = fold\_right (fun e a -> e :: a) x y

We can use a more complicated accumulator, such as a tuple. In this example, we replicate the List.split function which, given a list of pairs, yields a pair of lists:

```
split : (\alpha \times \beta) list \rightarrow \alpha list \times \beta list

let split l =

fold_right

(fun (x, y) (xs, ys) -> (x :: xs, y :: ys))

l

([], [])
```

For example, split [(1, "one"); (2, "two")] evaluates to ([1; 2], ["one"; "two"]).

#### A word of caution

One very simple definition for the function concat which concatenates all lists in a list of lists is given by:

concat :  $\alpha$  list list  $\rightarrow \alpha$  list let concat l = fold\_left (@) [] l We use the append function to accumulate the lists into a single list one by one:

```
fold_left (@) [] [[1;2]; [3]; [4;5]]

⇒ fold_left (@) [1; 2] [[3]; [4; 5]]

⇒ fold_left (@) [1; 2; 3] [[4; 5]]

⇒ fold_left (@) [1; 2; 3; 4; 5] []

⇒ [1; 2; 3; 4; 5]
```

However, the order of evaluation is such that the append function @ (which takes time proportional to the length of its first argument) is used inefficiently – we process the list again and again.

#### Folding over trees

For the usual definition of a binary tree, we can define a fold. There are two accumulators, one for everything from the left sub-tree, and one for everything from the right sub-tree. The supplied function combines both into a new accumulator.

```
fold_tree : (\alpha \rightarrow \beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha tree \rightarrow \beta

type 'a tree =

Lf

| Br of 'a * 'a tree * 'a tree

let rec fold_tree f e t =

match t with

Lf -> e

| Br (x, l, r) -> f x (fold_tree f e l) (fold_tree f e r)
```

Here is an example tree:



Functions for the size of a tree, and the sum of an integer tree are now easy, without explicit recursion:

tree\_size :  $\alpha$  tree  $\rightarrow$  int tree\_sum : int tree  $\rightarrow$  int let tree\_size t = fold\_tree (fun \_ l r -> 1 + l + r) 0 t let tree\_sum t = fold\_tree (fun x l r -> x + l + r) 0 t

The standard tree traversals can be written easily with a list accumulator. A little typographical manipulation shows the pleasing symmetry:

```
\begin{array}{l} \mbox{tree\_preorder} : \alpha \mbox{tree} \rightarrow \alpha \mbox{list} \\ \mbox{tree\_inorder} : \alpha \mbox{tree} \rightarrow \alpha \mbox{list} \\ \mbox{tree\_postorder} : \alpha \mbox{tree} \rightarrow \alpha \mbox{list} \\ \mbox{let} \mbox{tree\_preorder} t = \mbox{fold\_tree} (\mbox{fun} \ x \ l \ r \ -> \ [x] \ @ \ l \ @ \ r) \ [] \ t \\ \mbox{let} \ \mbox{tree\_postorder} t = \mbox{fold\_tree} (\mbox{fun} \ x \ l \ r \ -> \ l \ @ \ [x] \ @ \ r) \ [] \ t \\ \mbox{let} \ \mbox{tree\_postorder} t = \mbox{fold\_tree} (\mbox{fun} \ x \ l \ r \ -> \ l \ @ \ r \ @ \ [x] \ @ \ r) \ [] \ t \\ \mbox{let} \ \mbox{tree\_postorder} t = \mbox{fold\_tree} (\mbox{fun} \ x \ l \ r \ -> \ l \ @ \ r \ @ \ [x] \ @ \ r) \ [] \ t \\ \mbox{let} \ \mbox{tree\_postorder} t = \mbox{fold\_tree} (\mbox{fun} \ x \ l \ r \ -> \ l \ @ \ r \ @ \ [x] \ ) \ [] \ t \\ \mbox{let} \ \mbox{fold\_tree} \ \mbox{fun} \ \mbox{fun} \ x \ l \ r \ -> \ l \ @ \ r \ @ \ [x] \ ) \ [] \ t \\ \mbox{let} \ \mbox{fun} \ \mbox{fun
```

On our example list:

tree\_preorder (Br (1, Br (0, Lf, Lf), Br (6, Br (4, Lf, Lf), Lf)))
⇒ [1; 0; 6; 4]
tree\_inorder (Br (1, Br (0, Lf, Lf), Br (6, Br (4, Lf, Lf), Lf)))
⇒ [0; 1; 4; 6]
tree\_postorder (Br (1, Br (0, Lf, Lf), Br (6, Br (4, Lf, Lf), Lf)))
⇒ [0; 4; 6; 1]

#### Questions

- 1. Write a function which, given a list of integers representing expenses, removes them from a budget, again represented by an integer.
- 2. Calculate the length of a list using one of the fold\_ functions.
- 3. Use one of the fold\_ functions to find the last element of list, if any. Behave sensibly if the list is empty.
- 4. Write a function to reverse a list, using one of the fold\_ functions.
- 5. Write a version of List.mem using one of the fold\_ functions. Now setify can be defined entirely using folds.
- 6. Use a fold to write a function which, given a list of non-empty strings representing words, returns a single string where the words are separated by spaces. Comment on its efficiency.
- 7. Use fold\_tree to write a function which calculates the maximum depth of a tree. What is its type?
- 8. Compare the time efficiency of one or more of your functions with the system implementation of the same function (for example, our fold-based member function vs. List.mem) with regard to both computational complexity and actual time taken.
- 9. Comment on whether the use of folds in each of Questions 1–7 is good style.