FPV Week 6 [*]

Zixuan Fan

April 2022

1 List Folding

There two kinds of folding in OCaml, fold_left and fold_right. They are of similar usage, but iterate the list in different directions.

- val fold_left : ('a → 'b → 'a) → 'a → 'b list → 'a

  this function takes 3 variables, a function, an accumulator of type a and a list of type b. What it does is taking the first element of the list. Apply it and the accumulator to the given function. Save the result as the new accumulator and pass it recursively to the next call with the same function, new accumulator and the tail of the previous list.

  EXAMPLE: *let sum = fold_left (+) 0 [1;2;3];;*

  We calculate the sum of 1, 2 and 3. Evaluating the function step by step, we have

[*] All contents are based on the Artemis exercises and lecture slides of Prof. Seidl. No content is guaranteed to be totally correct.

$$\begin{aligned}
let\ sum &= fold\_left\ (+)\ 0\ [1;2;3] \\
&= fold\_left\ (+)\ ((+)\ 1\ 0))\ [2;3] \\
&= fold\_left\ (+)\ (1+0)\ [2;3] \\
&= fold\_left\ (+)\ 1\ [2;3] \\
&= ... \\
&= 6
\end{aligned}$$

- val fold_right : ('a → 'b → 'b) → 'a list → 'b →'b

  Similar to the previous function, but this time we start from the last element of the list, hence the recursive structure should be similar to what we used for the rev function.

  EXAMPLE: *let sum = fold_right (+) [1;2;3] 0*

  We calculate the sum of 1, 2 and 3. Evaluating the function step by step, we have

$$\begin{aligned}
let\ sum &= fold\_left\ (+)\ [1;2;3]\ 0 \\
&= fold\_left\ (+)\ [1;2]\ ((+)\ 3\ 0)) \\
&= fold\_left\ (+)\ [1;2]\ (3+0) \\
&= fold\_left\ (+)\ [1;2]\ 3 \\
&= ... \\
&= 6
\end{aligned}$$

Obviously, we might have the same result using both of the folding functions when calculating the sum. However, in most scenarios, an order is important. Hence choosing a suitable folding function is very important.

2 Operator functions

As you might have seen in the example of the previous chapter, the function (+) was applied. It is just an addition function. Similarly, you might assume that the normal addition of integers and floats are also implemented via a function, rather than an internal arithmetic. That is exactly what happens here.

In Ocaml, all of the operators are just coding sugar, they are by nature still functions. This also leads to an operator override or creating user-defined operator functions using operators that are not yet supported.

An example in the list module:

let (@) l1 l2 = append l1 l2

Open question: Can you define an override for addition and multiplication for the Peano's nat you defined in the last week's homework?