Tail Call Recursion vs Plain Recursion

When comparing both versions of the reverse functions, one using Tail Recursion and the other one using plain recursion, it is possible to observe that both implementations have similar runtime execution times, even with lists that contain a large number of elements ~1M, under such setup, the execution time accounts for 1 second.

Differences between right associative and left associative operators

When comparing the three implementations of `Accumulate`, using plain recursion, iterative tail calls and higher order functions, it is possible to observe that they produce different results. This phenomenon occurs due to the application of a non associative operators, in this case, the subtraction operator.

For instance, when `Accumulate` is applied to the list `20..30`, the result corresponds to -5. This result is obtained due to the application of the operator on a right associative fashion:

```
= 20 - (21 - (22 - (23 - (24 - (25 - (26 -(27 -(28 -(29 -(30 -(0)))))))))))
= 20 -  (21 - (22 - (23 - (24 - (25 - (26 -(27 -(28 -(29 -(30))))))))))
= 20 - (21 - (22 - (23 - (24 - (25 - (26 -(27 -(28 -(-1)))))))))
= 20 - (21 - (22 - (23 - (24 - (25 - (26 -(27 -(29))))))))
= ...
= 20 + 5
= 25
```

In contrast, `AccumulateTail` applies the operator on a right associative fashion, giving a result of -275. This can be obtained after expanding the corresponding recursive call to the operator, as it can be seen next:

```
= (((((((((((0 - 20) - 21) - 22) - 23) - 24) - 25) - 26) - 27) - 28) - 29)
- 30)
= ((((((((((-20 - 21) - 22) - 23) - 24) - 25) - 26) - 27) - 28) - 29) - 30)
= ...
= -275
```

Finally, the higher order application of FoldL presents the same behaviour as `AccumulateTail` due to the application of the substract operator in a left associative fashion. It is important to mention that if the subtract definition `f(x, y) -> x - y` is replaced by `f(x, y) -> y - x`, then the left associative application of the function yields the result of the right associative version.