Pure Recursion or map, reduce & filter

William Frid

Spring 2024

Introduction

Assume you have a list of integers in Elixir. From that list, you can do things such as filtering, bulk actions, or simply use all the integers for a calculation. These three types can be categorized into filtering, mapping, and reducing, respectively.

But how does one do this? There are two ways, either by using pure recursion, or through higher-order functions such as `map/2`, `reduce/3`, and `filter/2`. All are combined with lambda functions used for selections, checks, and mathematical operations.

This report will start by exploring the most basic case; the purely recursive functions. Then dive deeper into higher-order functions followed by pipelines used for for readable and more concise code.

Basic Recursive Functions

The basic recursive functions are written in three different ways depending on the type of function it is (that is filtering, mapping, or reducing). Take the filter-function `even/1` for instance that returns all even integers of a provided integer list (see snippet below). The snippet below shows that two functions are required, one general and one for the last check (empty list). The logic is simple and results in a non-tail recursive function.

```
def even([ ]) do [ ] end
def even([h|t]) do
    if rem(h, 2) == 0 do [h | even(t)] else even(t) end
end
```

Next, we have a map-function. Take `inc/2` (snippet below) for instance that takes in an integer list and an integer, then increases all list records by the provided integer. This logic, compared to the previous function, is even simpler. Once again two functions for pattern matching are required, but the function itself is a "one-liner" (non-trail recursive).

```
def inc([ ], _) do [ ] end
def inc([h | t], i) do [h + i | inc(t, i)] end
```

Lastly, there's the reduce-functions. Take `prod/1` as an example with an integer list as the only argument (snippet below this paragraph). Its purpose is to find the product of all the integers in the given list and return that (that is just an integer). Note how three versions are required to be able to catch single-element and empty lists. Also, keep in mind that this function is non-tail recursive.

```
def prod([ ]) do 0 end
def prod([ x ]) do x end
def prod([h | t]) do h * prod(t) end
```

Note how all the functions above have been called "non-tail recursive". This is equivalent to "simply recursive".

Filter, Map, Reduce

The higher-order functions can be used to achieve the same thing as in the previous section combined with lambda functions. Writing these functions is however more complicated. The logic required for `map/2` and `reduce/3` is somewhat simpler than what `filter/2` requires. Thus that's what we'll be focusing on.

The function `filter/2` can be written in three ways; non-tail recursive, tail recursive (result in reverse), and adjusted tail recursion (result in correct order). Our non-tail recursive function `filterr/2` requires in this case less logic and can be written as only two functions (snippet below). We see how it uses the provided lambda function to check whether to include the integer in question or not.

```
def filterr([ ], _) do [ ] end
def filterr([h | t], op) do
    if op.(h) do [h | filterr(t, op)] else filterr(t, op) end
end
```

Assume we wanted to write this function as tail recursive for performance reasons. Then we could write it as `filterl/3` shown below. This requires that the last action done by the function is recursive, and because of this, additional logic is required to handle the single element left in the list. Also, note how a third argument (and empty list) is required as this will hold the result.

The problem with this function, however, is that it returns the result in reverse. Meaning if the answer would be $[1, 2, 3]$, it would return $[3, 2, 1]$. This can be fixed by swapping the order `lst2` is combined with the list entities.

```
def filterl([ ], _, _) do [ ] end
def filterl([ x ], lst2, op) do
    if op.(x) do [ x | lst2 ] else lst2 end
end
def filterl([h1 | t1], lst2, op) do
    if op.(h1) do
        filterl(t1, [h1 | lst2], op)
    else
        filterl(t1, lst2, op)
    end
end
```

But which version is the most efficient? Tail recursive or basic recursive? The answer is in this case basic recursive! This is because if it was tail recursive, it would have to store the whole list, in each recursive call and then start filtering. Assume the integer list is 10 in length, this would result in 100 stored list entries. The basic recursive call would result in a total of $10, 9, 8, ..., 1$ stored entries, decreasing for each recursion.

Pros. of Higher-Order Functions

Now that we have our higher-order functions, let's re-implement the function `even/1`, `inc/2`, and `prod/1` that were written under the section "Basic Recursive Functions". As we can see below, the implementation was indeed improved as it requires less code.

```
def new_even(lst) do
    filterlf(lst, [], fn(x) -> rem(x, 2) == 0 end)
end
def new_inc(lst, n) do map(lst, fn(x) -> x + n end) end
def new_prod(lst) do reducer(lst, fn(x, y) -> x * y end) end
```

This usage is great when it comes to manipulating each list entry in multiple steps. For instance, assume we wish to implement a function that takes in a list and returns the sum of the squares of all entries with values less than n. This can be easily accomplished in the way seen in the snippet below. The drawback with this implementation is that unclear syntax (could be even worse if you put the function in function). This can be fixed with something called the "pipe operator".

```
def foo(lst, n) do
    less_than_n = filterlf(lst, [], fn(x) -> x < n end)
    squares = map(less_than_n, fn(x) -> x * x end)
    reducel(squares, 0, fn(x, y) -> x + y end)
end
```

The Pipe Operator

As shown in the previous section, taking the output from one function and passing it into the next can build up and lead to a chaotic code structure that's hard to follow. This is why the pipe operator was created. It takes an argument, in our case the list, and then it passes this as the first argument of the next function. That function output is passed as the first argument to the next function after that, and so on.

This is carried out until the end where the output, in this case of `reducel/3` is returned. Note how the function `bar/2` has the same purpose as `foo/2` above but is written more clearly.

```
def bar(lst, n) do
    lst |>
        filterlf([], fn(x) -> x < n end) |>
        map(fn(x) -> x * x end) |>
        reducel(0, fn(x, y) -> x + y end)
end
```

Discussion

Since no benchmarks were run for this report it's hard to tell whatever tail recursion or regular recursion was faster in the case of `filter/3`. One can only argue based on knowledge and math which one is fastest. This can however not be fully trusted due to other factors that might play in. Thus benchmarks competing all three implementations should have been run to strengthen these claims.

Conclusion

Using higher-order functions can simplify the creation of other functions that fall under the same category. It's a skill important to master as the results lead to great improvements.

Furthermore, learning when to write the function as tail recursive is important to achieve higher performance. And where performance isn't required, it might lead to shorter and cleaner code than typical recursive functions.

Lastly, we have the pipeline operator, which is an important tool when manipulating data as this is often done in multiple steps. Clear structured code can simply the debugging and help other developers better understand what's going on without having to memorize each variable.