MAPS, FILTERS, FOLDS AND MORE

At the beginning of this chapter, I briefly showed how to abstract away two similar functions to get a `map/2` function. I also affirmed that such a function could be used for any list where we want to act on each element. The function was the following:

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].
```

However, there are many other similar abstractions to build from commonly occurring recursive functions. Let's first take a look at these two functions:

```
%% only keep even numbers
even(L) -> lists:reverse(even(L,[])).

even([], Acc) -> Acc;
even([H|T], Acc) when H rem 2 == 0 ->
even(T, [H|Acc]);
even([_|T], Acc) ->
even(T, Acc).

%% only keep men older than 60
old_men(L) -> lists:reverse(old_men(L,[])).

old_men([], Acc) -> Acc;
old_men([Person = {male, Age}|People], Acc) when Age > 60 ->
old_men(People, [Person|Acc]);
old_men([_|People], Acc) ->
old_men(People, Acc).
```

The first one takes a list of numbers and returns only those that are even. The second one goes through a list of people of the form `{Gender, Age}` and only keeps those that are males over 60. The similarities are a bit harder to find here, but we've got some common points. Both functions operate on lists and have the same objective of keeping elements that succeed some test (also a *predicate*) and then drop the others. From this generalization we can extract all the useful information we need and abstract them away:

```
filter(Pred, L) -> lists:reverse(filter(Pred, L,[])).

filter(_, [], Acc) -> Acc;
filter(Pred, [H|T], Acc) ->
case Pred(H) of
true -> filter(Pred, T, [H|Acc]);
false -> filter(Pred, T, Acc)
end.
```

To use the filtering function we now only need to get the test outside of the function. Compile the `hhfuns` module and try it:

```
1> c(hhfuns).
{ok, hhfuns}
2> Numbers = lists:seq(1,10).
[1,2,3,4,5,6,7,8,9,10]
3> hhfuns:filter(fun(X) -> X rem 2 == 0 end, Numbers).
[2,4,6,8,10]
4> People = [{male,45},{female,67},{male,66},{female,12},{unknown,174},{male,74}].
[{male,45},{female,67},{male,66},{female,12},{unknown,174},
{male,74}]
5> hhfuns:filter(fun({Gender,Age}) -> Gender == male andalso Age > 60 end, People).
[{male,66},{male,74}]
```

These two examples show that with the use of the `filter/2` function, the programmer only has to worry about producing the predicate and the list. The act of cycling through the list to throw out unwanted items is no longer necessary to think about. This is one important thing about abstracting functional code: try to get rid of what's always the same and let the programmer supply in the parts that change.

In the previous chapter, another kind of recursive manipulation we applied on lists was to look at every element of a list one after the other and reduce them to a single answer. This is called a *fold* and can be used on the following functions:

```
%% find the maximum of a list
max([H|T]) -> max2(T, H).

max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).

%% find the minimum of a list
min([H|T]) -> min2(T,H).

min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).

%% sum of all the elements of a list
sum(L) -> sum(L,0).

sum([], Sum) -> Sum;
sum([H|T], Sum) -> sum(T, H+Sum).
```

To find how the fold should behave, we've got to find all the common points of these actions and then what is different. As mentioned above, we always have a reduction from a list to a single value. Consequently, our fold should only consider iterating while keeping a single item, no list-building needed. Then we need to ignore the guards, because they're not always there: these need to be in the user's function. In this regard, our folding function will probably look a lot like sum.

A subtle element of all three functions that wasn't mentioned yet is that every function needs to have an initial value to start counting with. In the case of `sum/2`, we use 0 as we're doing addition and given `X = X + 0`, the value is neutral and we can't mess up the calculation by starting there. If we were doing multiplication we'd use 1 given `X = X * 1`. The functions`min/1` and `max/1` can't have a default starting value: if the list was only negative numbers and we started at 0, the answer would be wrong. As such, we need to use the first element of the list as a starting point. Sadly, we can't always decide this way, so we'll leave that decision to the programmer. By taking all these elements, we can build the following abstraction:

```
fold(_, Start, []) -> Start;
fold(F, Start, [H|T]) -> fold(F, F(H,Start), T).
```

And when tried:

```
6> c(hhfuns).
{ok, hhfuns}
7> [H|T] = [1,7,3,5,9,0,2,3].
[1,7,3,5,9,0,2,3]
8> hhfuns:fold(fun(A,B) when A > B -> A; (_,B) -> B end, H, T).
9
9> hhfuns:fold(fun(A,B) when A < B -> A; (_,B) -> B end, H, T).
0
10> hhfuns:fold(fun(A,B) -> A + B end, 0, lists:seq(1,6)).
21
```

Pretty much any function you can think of that reduces lists to 1 element can be expressed as a fold.

What's funny there is that you can represent an accumulator as a single element (or a single variable), and an accumulator can be a list. Therefore, we can use a fold to build a list. This means fold is universal in the sense that you can implement pretty much any other recursive function on lists with a fold, even map and filter:

```
reverse(L) ->
fold(fun(X,Acc) -> [X|Acc] end, [], L).

map2(F,L) ->
reverse(fold(fun(X,Acc) -> [F(X)|Acc] end, [], L)).

filter2(Pred, L) ->
```

```
F = fun(X,Acc) ->
case Pred(X) of
true -> [X|Acc];
false -> Acc
end
end,
reverse(fold(F, [], L)).
```

And they all work the same as those written by hand before. How's that for powerful abstractions?

Map, filters and folds are only one of many abstractions over lists provided by the Erlang standard library (see`lists:map/2`, `lists:filter/2`, `lists:foldl/3` and `lists:foldr/3`). Other functions include `all/2` and `any/2` which both take a predicate and test if all the elements return true or if at least one of them returns true, respectively. Then you have `dropwhile/2` that will ignore elements of a list until it finds one that fit a certain predicate, its opposite, `takewhile/2`, that will keep all elements until there is one that doesn't return true to the predicate. A complimentary function to the two previous ones is `partition/2`, which will take a list and return two: one that has the terms which satisfy a given predicate, and one list for the others. Other frequently used lists functions include`flatten/1`, `flatlength/1`, `flatmap/2`, `merge/1`, `nth/2`, `nthtail/2`, `split/2` and a bunch of others. You'll also find other functions such as zippers (as seen in last chapter), unzippers, combinations of maps and folds, etc. I encourage you to read the documentation on lists to see what can be done. You'll find yourself rarely needing to write recursive functions by using what's already been abstracted away by smart people.

Source : http://learnyousomeerlang.com/higher-order-functions