Learn You Some Erlang for Great Good!

A Beginner's Guide

Fred Hébert

Foreword by Joe Armstrong

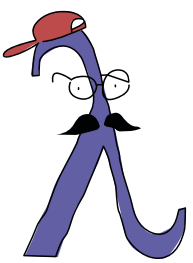
no starch press

6

HIGHER-ORDER FUNCTIONS

An important part of all functional programming languages is the ability to take a function you defined and then pass it as a parameter to another function. This binds that function parameter to a variable, which can be used like any other variable within the function. A function that can accept other functions transported around this way is called a *higher-order function*. As you'll learn in this chapter, higher-order functions are a powerful means of abstraction and one of the best tools to master in Erlang.

Let's Get Functional



The concept behind carrying functions around and passing them to higher-order functions is rooted in mathematics, mainly lambda calculus. Basically, in pure lambda calculus, everything is a function—even numbers, operators, and lists. Because everything is represented as a function, functions must accept other

functions as parameters, and must be able to operate on them with even more functions! (If you want a good, quick introduction to lambda calculus, read the Wikipedia entry for it.)

This concept might be a bit hard to grasp, so let's start with an example (this is nowhere close to real lambda calculus, but it illustrates the point).

```
-module(hhfuns).
-compile(export_all).

one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

Now open the Erlang shell, compile the module, and get going:

```
1> c(hhfuns).
{ok, hhfuns}
2> hhfuns:add(one,two).
** exception error: bad function one
in function  hhfuns:add/2
3> hhfuns:add(1,2).
** exception error: bad function 1
in function  hhfuns:add/2
4> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
```

Confusing? Not so much, once you know how it works (isn't that always the case?). In line 2, the atoms `one` and `two` are passed to `add/2`, which then uses both atoms as function names (`X() + Y()`). If function names are written without a parameter list, then those names are interpreted as atoms, and atoms cannot be functions, so the call fails. This is why the expression on line 3 also fails: The values 1 and 2 cannot be called as functions, and functions are what we need!

To handle this issue, a new notation must be added to the language in order to pass functions from outside a module. This is the purpose of `fun Module:Function/Arity:`, which tells the VM to use that specific function and then bind it to a variable.

So what are the gains of using functions in that manner? Well, a little example might help answer that question. We'll add a few functions to `hhfuns` that work recursively over a list to add or subtract one from each integer of a list.

```
increment([]) -> [];
increment([H|T]) -> [H+1|increment(T)].

decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].
```

Do you see how similar these functions are? They basically do the same thing: cycle through a list, apply a function on each element (+ or -), and then call themselves again. Almost nothing changes in that code; only the applied function and the recursive call are different. The core of a recursive call on a list like that is always the same. We'll abstract all the similar parts in a single function (`map/2`) that will take another function as an argument.

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

incr(X) -> X + 1.
decr(X) -> X - 1.
```

Now let's test this in the shell.

```
1> c(hhfuns).
{ok, hhfuns}
2> L = [1,2,3,4,5].
[1,2,3,4,5]
3> hhfuns:increment(L).
[2,3,4,5,6]
4> hhfuns:decrement(L).
[0,1,2,3,4]
5> hhfuns:map(fun hhfuns:incr/1, L).
[2,3,4,5,6]
6> hhfuns:map(fun hhfuns:decr/1, L).
[0,1,2,3,4]
```

Here, the results are the same, but we have just created a very smart abstraction! Every time we want to apply a function to each element of a list, we only need to call `map/2` with our function as a parameter. However, it is a bit annoying to need to put every function we want to pass as a parameter to `map/2` in a module, name it, export it, compile it, and so on. In fact, it's plainly unpractical. What we need are functions that can be declared on the fly—the type of functions discussed next.

Anonymous Functions

Anonymous functions, or *funs*, address the problem of using functions as parameters by letting you declare a special kind of function inline, without naming that function. Anonymous functions can do pretty much everything normal functions can do, except call themselves recursively (how could they do that if they are anonymous?).

Anonymous functions have the following syntax:

```
fun(Args1) ->
    Expression1, Exp2, ..., ExpN;
  (Args2) ->
    Expression1, Exp2, ..., ExpN;
```

```
    (Args3) ->
      Expression1, Exp2, ..., ExpN
end
```

Here's an example of using an anonymous function:

```
7> Fn = fun() -> a end.
#Fun<erl_eval.20.67289768>
8> Fn().
a
9> hhfuns:map(fun(X) -> X + 1 end, L).
[2,3,4,5,6]
10> hhfuns:map(fun(X) -> X - 1 end, L).
[0,1,2,3,4]
```

And now you're seeing one of the things that make people like functional programming so much: the ability to make abstractions on a very low level of code. Basic concepts such as looping can thus be ignored, letting you focus on what is done, rather than how to do it.

More Anonymous Function Power

Anonymous functions are pretty dandy for such abstractions, but they have more hidden powers. Let's look at another example:

```
11> PrepareAlarm = fun(Room) ->
11>     io:format("Alarm set in ~s.~n",[Room]),
11>     fun() -> io:format("Alarm tripped in ~s! Call Batman!~n",[Room]) end
11> end.
#Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.6.13229925>
13> AlarmReady().
Alarm tripped in bathroom! Call Batman!
ok
```

Hold the phone, Batman! What's going on here? Well, first of all, we declare an anonymous function assigned to `PrepareAlarm`. This function has not run yet. It is executed only when `PrepareAlarm("bathroom")` is called. At that point, the call to `io:format/2` is evaluated, and the "Alarm set" text is output. The second expression (another anonymous function) is returned to the caller and then assigned to `AlarmReady`. Note that in this function, the `Room` variable's value is taken from the "parent" function (`PrepareAlarm`). This is related to a concept called *closures*. But before we can talk about closures, we need to address the idea of scope.

Function Scope and Closures

A function's *scope* can be imagined as the place where all the variables and their values are stored. In the function `base(A) -> B = A + 1.`, for example, `A` and `B` are both defined to be part of `base/1`'s scope. This means that anywhere inside `base/1`, you can refer to `A` and `B` and expect a value to be bound to them. And when I say "anywhere," I ain't kidding, kid. This includes anonymous functions, too.

```
base(A) ->
    B = A + 1,
    F = fun() -> A * B end,
    F().
```

In this example, `B` and `A` are still bound to `base/1`'s scope, so the function `F` can still access them. This is because `F` inherits `base/1`'s scope. As with most kinds of real-life inheritance, the parents can't get what the children have.

```
base(A) ->
    B = A + 1,
    F = fun() -> C = A * B end,
    F(),
    C.
```

In this version of the function, `B` is still equal to `A + 1`, and `F` will still execute fine. However, the variable `C` is only in the scope of the anonymous function in `F`. When `base/1` tries to access `C`'s value on the last line, it finds only an unbound variable. In fact, if you tried to compile this function, the compiler would throw a fit. Inheritance goes only one way.

It is important to note that the inherited scope follows the anonymous function wherever it is, even when it is passed to another function. Here's an example:

```
a() ->
    Secret = "pony",
    fun() -> Secret end.

b(F) ->
    "a/0's password is "++F().
```

Now we can compile it.

```
14> c(hhfuns).
{ok, hhfuns}
15> hhfuns:b(hhfuns:a()).
"a/0's password is pony"
```

Who told a/0's password? Well, a/0 did. While the anonymous function has a/0's scope when it's declared in there, the function can still carry it when executed in b/1, as explained earlier. This is very useful because it lets us carry around parameters and content out of their original context, where the whole context itself is no longer needed (exactly as we did with Batman in the previous section).

You're most likely to use anonymous functions to carry state around when you have defined functions that take many arguments, but one of these arguments remains the same all the time, as in the following example.

```
16> math:pow(5,2).
25.0
17> Base = 2.
2
18> PowerOfTwo = fun(X) -> math:pow(Base,X) end.
#Fun<erl_eval.6.13229925>
17> hhfuns:map(PowerOfTwo, [1,2,3,4]).
[2.0,4.0,8.0,16.0]
```

By wrapping the call to math:pow/2 inside an anonymous function with the Base variable bound in that function's scope, we made it possible to have each of the calls to PowerOfTwo in hhfuns:map/2 use the integers from the list as the exponents of our base.

A little trap you might fall into when writing anonymous functions is when you try to redefine the scope, like this:

```
base() ->
    A = 1,
    (fun() -> A = 2 end)().
```

This will declare an anonymous function and then run it. As the anonymous function inherits base/0's scope, trying to use the = operator compares 2 with the variable A (bound to 1). This is guaranteed to fail. However, we can redefine the variable if we do that in the nested function's head:

```
base() ->
    A = 1,
    (fun(A) -> A = 2 end)(2).
```

And this works. If you try to compile it, you'll get a warning about shadowing: "Warning: variable 'A' shadowed in 'fun'." *Shadowing* is the term used to describe the act of defining a new variable that has the same name as one that was in the parent scope. This warning is there to prevent some mistakes (usually rightly so), so you might want to consider renaming your variables in these circumstances.

Now that we've covered scope, we can turn to closures. Closure is just the idea of having a function that references some environment along with

it (the value's part of the scope). In other words, a closure is what happens when anonymous functions meet the concept of scope and carrying variables around.

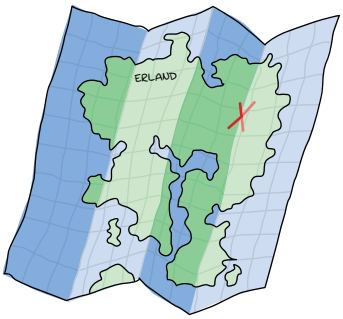
We'll set the anonymous function theory aside for now and explore more common abstractions to avoid needing to write more recursive functions, as I promised at the end of Chapter 5.

Maps, Filters, Folds, and More

At the beginning of this chapter, we took a brief look at how to abstract away two similar functions to get a `map/2` function:

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

Such a function can be used for any list where we want to act on each element. However, there are many other similar abstractions to build from commonly occurring recursive functions.



Filters

First, we'll look at filters. Consider the following 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 of these functions 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 keeps only those that are males over 60.

The similarities are a bit harder to find here than in the previous examples, but we have some common points. Both functions operate on lists and have the same objective of keeping elements that succeed some test (also called a *predicate*) and then dropping the others. From this generalization, we can extract all the useful information we need and abstract them away, like this:

```
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 pass in a predicate 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},{unkown,174},{male,74}].
[{male,45},{female,67},{male,66},{female,12},{unkown,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 needs to worry only about producing the predicate and the list. The act of cycling through the list to throw out unwanted items is no longer a consideration. This is one important thing about abstracting functional code: Try to get rid of what's always the same, and let the programmer supply the parts that change.

Fold Everything

In Chapter 5, we looked at another kind of recursive list manipulation, where we applied some operation to each element of a list successively to reduce the elements to a single value. This is called a *fold* and can be used to reduce the size of 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).

%% Find the 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` function should be used, we need to determine all the common points of the actions made by these functions, as well as what is different. As mentioned earlier, we always have a reduction from a list to a single value. Consequently, our `fold` function should consider iterating only while keeping a single item—no list building is needed. We need to ignore the guards, because they exist in only some of these functions, not all of them. The guards will need to be included in the function that we pass to `fold`. In this regard, our `fold` function will probably look a lot like `sum`.

A subtle element of all three functions 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, so we can't mess up the calculation by starting there. If we were doing multiplication, we would use 1 given `X = X * 1`.

The functions `min/1` and `max/1` can't have a default starting value. If the list were only negative numbers and we started at 0, the answer would be wrong. So we need to use the first element of the list as a starting point. Sadly, we can't always decide the starting value 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).
```

Let's try it.

```
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 one element can be expressed as a fold.

Strangely enough, 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 folding is universal in the sense that you can implement pretty much any other recursive function on lists with a fold, even maps and filters, like so:

```
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)).
```

These all work in the same way as those written by hand before. How's that for powerful abstractions?

More Abstractions

Map, filters, and folds are only a few 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.

Also available is `dropwhile/2`, which will ignore elements of a list until it finds one that fits a certain predicate. Its opposite, `takewhile/2`, will keep all elements until there is one that doesn't return `true` to the predicate. A complementary function to these is `partition/2`, which will take a list and return two lists: one that has the terms that satisfy a given predicate and one for the others.

Other frequently used list functions include `flatten/1`, `flatlength/1`, `flatmap/2`, `merge/1`, `nth/2`, `nthtail/2`, and `split/2`. You can look up all of these functions in the documentation if you want to learn more about them.

You'll also find other functions such as zipping functions (as shown in Chapter 5), unzipping functions, combinations of maps and folds, and so on. 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 as long as you use what's already been abstracted away by smart people.