The functional way of thinking about programming

Marcin Pastudzki

August 28, 2018

It's all about composition.

Although functional programming has constantly been gaining popularity in the past few years, there is still much misconception and prejudice about it. Many people understand the term very superficially or don't understand it at all, and still claim to have adopted the way.

Most people never realize that functional programming is something more than just a set of new methods to write computer programs. It goes deeper even than the restrictions imposed upon value assignment and immutability, which makes our programs, sometimes, a little more difficult to write, but a lot easier to read and comprehend. In fact functional programming is a very different approach to programming, a different way of thinking about problems and expressing solutions.

So what exactly is functional programming? In procedural (and object-oriented) paradigm one thinks about their program as a sequence of steps that need to be taken in order to reach a certain goal. Input to the program is usually bound to a variable name, then some operations are executed on that variable and after execution is done, the final value of that variable becomes the result of the program. Let us illustrate this with a simple example implementation of factorial function in JavaScript.

```
function factorial(n) {
    var result = 1;
    for (i = 2; i <= n; i++) {
        result *= i;
    }
    return result
}
```

On the other hand functional approach is to think about data being transformed rather than actions that our program executes. A factorial function might as well be implemented like so:

```
function factorial(n) {
    return n > 1 ? n * factorial(n - 1) : 1;
}
```

Not only is this code a lot shorter and more elegant, but it also fundamentally states things in different manner. Here we do not say: set *result* to 1, in a loop multiply it by some numbers and finally return it. Instead we declare that *if* input is greater than 1 *factorial* **is** n times the factorial of n's predecessor; otherwise it's just 1. Although in JavaScript this is less performant due to the necessity to allocate a stack frame for each recursive call, most modern compilers optimize away that inefficiency with ease[1]. This is how it would look like in a purely functional language Haskell:

```
factorial n = if n > 1 then n * factorial (n - 1) else 1
```

[1] *Tail call optimization*, can be preformed whenever the final value of a function is expressed as pure recursive call without further modifications. In such case there is no need to create a new stack frame, as the old one is no longer needed and can be reused for the new call, effectively transforming the function into a loop. Most recursive functions that don't meet this criterium can be transformed so that they do, usually by adding additional arguments.

A solid type system is a very important tool that can be used to ease the process of selecting proper transformations that turn our input into desired output. Suppose we have input of type a and require some output of type b. Then instead of writing down steps that convert such input into desired output, we *look for* functions in our libraries of type $a \to b$. These are obvious candidates for our program's body. This way we naturally look for code already written and ready for reuse rather than implement the same algorithms over and over again.

Of course real-life problems are a lot more complicated than that and often require multiple transformations between the same types. We develop types and data structures of our own and then we need to define operations on these types ourselves, but in doing so we also look at types of components underlying our structures and define operations on our new structures in terms of operations on their components.

Moreover, it is almost always better to solve our problems with functions rather than built-in language constructs wherever possible. This is because unlike syntactic constructs, functions are *composable*. In the following example we have a list of records containing information about European capital cities. Suppose we need to sort them first by population count and then (in case of equal population) by density:

```
function city(name, population, area) {
    this.name = name;
    this.area = area;
    this.population = population;
}

cities = [
    new city("Rome", 2868000, 1285),
    new city("London", 8136000, 1572),
    new city("Paris", 2244000, 105),
    new city("Berlin", 3000000, 891),
    new city("Moscow", 11920000, 2511),
    new city("Mardid", 3000000, 604)
];

function compare(a, b) {
    if (a > b) return 1;
    if (a < b) return -1;
    return 0;
}

function name(city) {
    return city.name;
}

function area(city) {
    return city.area;
}

function population(city) {
    return city.population;
}

function density(city) {
    return population(city) / area(city);
```

```
}

function compareRecords(getter) {
    return function(a, b) {
        return compare(getter(a), getter(b))
    };
}

function combineComparators(cmp1, cmp2) {
    return function(a, b) {
        let fst = cmp1(a, b);
        return fst == 0 ? cmp2(a, b) : fst;

    }
}

cities.sort(combineComparators(
    compareRecords(population),
    compareRecords(density)
));
```

Notice the functions `name`, `area` etc. below the object constructor. We introduce them as accessor functions for our objects, because in a functional style it's often more convenient than the dot-syntax for accessing object properties. `compareRecords` is a general function that lets us build custom comparators for various objects. It requires as an argument the accessor function `getter` which extracts a single property from an object.

`combineComparators` is even more interesting. It is a composition function that lets us combine simple comparators constructed with `compareRecords` function. It takes two such comparators and returns a new one which runs the first comparator on given records and if they're considered equal then runs the second to further distinguish between them.

Notice how we prefer accessor functions to dot-operator for accessing record's fields. This is because functions can be passed to `compareRecords` and `combineComparators` as arguments, while syntactic dot-operator, which is not a function, cannot.

This is how we could (even more elegantly) express this in Haskell:

```
import Data.List(sortBy)

data City = City {
    name :: String,
    population :: Float,
    area :: Float
}

cities = [
        City { name = "Rome", population = 2868000, area = 1285 },
        City { name = "London", population = 8136000, area = 1572 },
        City { name = "Paris", population = 2244000, area = 105 },
        City { name = "Berlin", population = 3000000, area = 891 },
        City { name = "Moscow", population = 11920000, area = 2511 },
```

```
        City { name = "Mardid", population = 3000000, area = 604 }
    ]

density c = population c / area c

compareRecordsBy getter record1 record2 =
    compare (getter record1) (getter record2)

( >?> ) comparator1 comparator2 rec1 rec2 =
    case comparator1 rec1 rec2 of
        EQ → comparator2 rec1 rec2
        result → result

main =
    let cmp = compareRecordsBy population >?> compareRecordsBy density in
    let sortedCities = sortBy cmp cities in
    print (map name sortedCities)
```

Notice how Haskell derives accessor functions from the record definition for us. In fact the traditional dot-syntax for accessing record fields is not even available in Haskell for it would find little use anyway. Also notice how it allows us to declare a custom operator (`( >?> )` in this case) just as easily as any regular function.

A careful observer could have spotted another difference as compared to JavaScript implementation: `compareRecordsBy` and `( >?> )` both have more arguments and they don't return functions at all. How can that be? A language feature called *partial function application* is here at work. When a function is applied to less arguments than it requires, instead of crashing, it just returns a new function, waiting for the remaining arguments to be delivered later[2].

The code above is the result of thinking along the lines of „what transformations do I need to apply to my inputs in order to obtain desired output". Literally no steps or actions are mentioned here at all. All this code says is „A is B, C is D and F is A with C plus E". Code developed this way not only is more elegant, but also more generic and reusable. Notice how these functions above (both in JavaScript and in Haskell implementation) can be used to compare and sort **any** objects in **any** project at all.

In order to write the code like this a major mindset shift is required. „What to do?" seems a completely natural question to ask when facing a problem to be solved by a computer program. However, another approach is both possible and advisable. Instead of „what to do?" one should ask „what do I have?", „what do I need?" and finally „what's the relationship between the two?". Instead of „how to do it?" we ask „what is it?". Then we split the problem into smaller pieces and solve them recursively one by one; then finally we combine the results and we're done.

As it was mentioned before, the type system can be used to ease this process. „What functions are there to transform input of type I have into the type I need?" „How can I combine those functions to obtain desired result?" These are the questions one should ask and the type system of a strongly-typed language (such as Haskell or OCaml) from an opponent in the struggle to make things work, becomes a wise guide and powerful ally.

[2] What we just said is a huge oversimplification, but I think it describes the phenomenon well enough without boring the reader with excessive details.