PRAGMATIC FUNCTIONAL JAVASCRIPT

by

Marcelo Camargo

Table of Contents

6. Types

7. Transforming values

8. Monads, monoids and functors

9. Async programming

10. Welcome to Fantasy Land

11. Hacking the compiler

12. A bit of theory

13. Functional languages targeting JavaScript

14. Solving real world problems

15. Final considerations

Introduction

What does "pragmatism" mean?

Pragmatism, noun, a **practical** aproach to problems and affairs. There is no word that can describe this book better. We are going to focus on practical applications of the beautiful and enchanting theory built around functional programming. This book is made for people with basic knowledges on JavaScript programming; you definitely shouldn't have exceptional abilities to understand the things here written: it is made to be practical, simple, concise and after that we except you to be able to write real world applications using the functional paradigm and finally say "I **do** work with functional programming!".

When possible, we'll provide the problem that we are trying to solve with alternative implementations and techniques in other paradigms. Programming is made to solve problems, but sometimes solving a problem may generate several other small problems. Before continuing, we need to set some premises:

- **Programming languages are not perfect**: seriously. They are built and designed by humans, and humans are not perfect (far from that!). They may contain failures and arbitrarily defined features, however, most times there is a reasonable explanation about the "workaround".
- **Problem solving may generate other problems**: if you are worried only about "getting shit done", this might be a bit controversial for you. When you solve a problem, have in mind that your solution is not free from introducing problems that other people will have to solve later. Being open to criticism is a good thing here; this is how technology evolves.
- **The right tool for the right job**: this is pragmatism. We pick the tool that solves the problem and introduces the lowest number of side-effects. Good programmers analyse trade-offs. There are a lot of programming languages published and dozens of paradigms, and they are not made/discovered only because "somebody likes writing that way" or "somebody wants to have their name in a programming language" (at least most times), but because new problems arise. It is sensible, for example, to use Erlang on telephone systems, Agda on mathematical proofs and Go on concurrent systems, but it is definitely insane to use Brainfuck on web development and PHP on compiler development!

The second chance

> And God said, "let there be functions", and there were functions.

In the beginning, computers were very slow, way slower than running Android Studio on your 2GB RAM machine! When the first physical computers appeared and programming languages started to be designed and implemented, there were mainly 2 mindsets:

1. Start from the Von Neumann architecture and **add abstraction**;
2. Start from mathematics and **remove abstraction**.

When it all started, dealing with high levels of abstraction was very hard and costly, memory and storage were really small in power, so imperative programming languages got a lot more visibility than functional ones because imperative languages had a lower level of abstraction that was way easier to implement and map directly to the hardware. It could not be the best way to design programs and express ideas, but at least it was the faster one to run.

But computers improved a lot, and functional programming finally got its deserved chance! Functional programming is not a new concept. I'm serious, it is really old and came directly from mathematics! The core concepts and the functional computational model came even before physical computers existed. It had its origins on lambda calculus, and it was initally only a formal system developed in the 1930s to investigate computability.

A lot of modern programming languages support well functional programming. Even Java surrended to this and implemented lambda expressions and streams. Most of you program in languages that were created; I want to show you the one which was discovered.

Why functional programming?

If imperative programming and other paradigms, like object orientation, were good enough, why do we need to learn a new way to code? The answer is: survival. Object orientation cannot save us from the cloud monster anymore. More complex problems have arisen and functional programming fits them very well. Functional programming is not "another syntax", as some people think; it is another mindset and a declarative way to solve problems. While in imperative programming you specify steps and how things happen, in declarative (functional and logic) programming you specify what has to be done, and the computer should decide the best way to do that. We've evolved a lot to do the job that a computer can do hundreds of times better than us.

Testability and maintainability

Modular and functional code bases are way easier to test and get a high coverage on unit tests. Things are very well isolated and independent, and by following all the main principles you get composable programs that work well together and have less bugs.

Parallelism

This is really variant with the implementation, but in languages that can track all sort of effects, you get parallelism and the possibility of clustering your program for free.

Optimization

Functional languages tend to be a lot easier to optimize and are more predictable for compilers. Knowing all sort of things that can haven in a program gives the possibility to the compiler to know the semantics of your code before even running it. Haskell can be even faster than C if you write idiomatic code! The main JavaScript engine, V8, has invested extensivily in otimizations for the functional paradigm.

Less is more

Having a lot of ways to do a job and to solve a problem in the same context is always good, right? Well, not always. If you pluck most programming languages that are largely used by the market, then you have a functional language — and this is valid also to JavaScript. JavaScript has strong ties with Scheme and has the perfect potential to be a very good functional language. Remove loops, mutability, references, conditional statements, exceptions, labels, blocks and you virtually have a dynamically typed OCaml dialect. I'm serious, you can even define functions in JavaScript without giving importance to the order they occur, just like OCaml em Haskell do — this is what we can call hoisting and we'll be seing how to take advantage of this in the next chapters.

Don't be miser. Do you really need all that different unaddressed language constructs? In the chapters of this book we'll discard a large part of JavaScript and we'll focus only in an expressive subset: functions and bindings, it is everything we'll need for now! Before learning functional programming with JavaScript, you first will have to think about unlearning some things. This is necessary to avoid language vices and to estimulate your brain to solve problems in a declarative way. For now, you'll have to trust, but in the course of this book you'll see how everything makes sense and fits into the purpose.

Null considered harmful

And by `null`, we also mean `undefined`. It is hard to see a programmer we never had any sort of problems with null references, null pointers or anything that can represent that absence of a value. For JavaScript programmers, `undefined is not a function`, for the C# young men in suits, `NullReferenceException`, and for unlucky one who use need to use Java, the classic `java.lang.NullPointerException`. Do you think we really need `null` as a special language construct? In this point, there is also a consensus between functional programmers and object-oriented programmers: `null` was a secular mistake.

It increases complexity

When you have to compare for null to avoid runtime errors, your code logic forks and creates a new path of possibilities. If you use lots of `null` values, the chance of having spaghetti code that becomes hard to maintain is really big. Object-oriented languages recommend the null object pattern, while functional languages use monads for that. Luckily, there are lots of libraries and monadic implementations for JavaScript to abstract and avoid this problem. We'll see monads in details and how to properly handle `null` later — for now we'll be presenting the problems and seeking the solution will be a task for the next chapters.

Unpredictable types

Why would you return `null` in a function that retrieves the list of users of the database when I can just return the **empty representation** of the expected type? — an empty array in this situation. The fact is that `null` can inhabit **any** type, and this makes the task of debugging a lot harder. Which one is more pragmatic?

```
const showUsers = () => {
  const users = getUsers()
  if (users !== null) {
    users.forEach(console.log)
  }
}
```

```
const showUsers = () => getUsers().forEach(console.log)
```

Don't make it complex when you can make it simple.

True, false and... null!?

Because who needs concise boolean logic? I do!

Unless you have a **very** good excuse, don't return `null` when all you need to do is giving a simple boolean answer. First thing: `null` is not `false`, `null` is the absence of value, and in this case we don't have a boolean representation, but a three-state model. Booleans should never be nullable.

Can I use undefined?

No. It is still another name and implementation for absence of value, but ECMAScript initial specifications had the two forms, so they were both kept to avoid breaking changes. After we'll learn how to wrap values that maybe exist and avoid increasing code complexity, wait for it!