THE FUNCTIONAL RUBYIST

Jim OKelly

Table of Contents

The Functional Rubyist

By Jim OKelly

Dedicated to my friend and mentor, Jim Weirich

What is Functional Programming

Functional programming is the concept of computing without global variables, mutation, or side effects. Well, that is technically a description of what it isn't.

To go further you can say it is the concept of algebraic computation. Everything is boiled down to a calculation of sorts. These calculations can also be applied partially applied to create larger calculations from smaller ones.

Two Different Models of Computation

Most of you are probably familiar with the Father of so called "modern computing", the good Mr. Turing. Turing is responsible for for what I will call "stateful computing", or the concept of moving parts. Things like global variables belong to the "stateful computing" model. You may have also heard that "side effects" are bad. These two are part of the same model.

By contrast there is another man who unless you were a Math nerd in school, you might not even know his name. It was Church and the model he put forward is essentially the calculator model, or the "algebraic model". In his model of computing it all boils down to mathematical *functions* which return a result. There are no global variables, there are no side effects.

You could say the Turing model is about "Operations" while the Church model is about "Calculations".

Don't make your code depend on the 4th dimension

Pure code (code without side-effects) is beautiful code because it is not polluted by the reality of Time. I know, I know, "What the hell does *polluted by time* even mean man?". Imagine a ledger program. It has a value that is your balance and that number can be positive or negative.

For the moment pretend that value is just a variable:

```
balance = 0.0
```

Now as you update the balance you change the balance variable:

```
balance = 0.0

balance += 1.5
```

So the balance started at 0.0 and then was changed when the 1.5 was added to it. At the end of this code sample the value is now at 1.5. This is the model proposed by Turing. And when coders talk of 'moving parts' they are talking about global variables that change. The `operation` here is an `addition`.

Since we have no idea what the balance was 5 minutes ago, once we change it we have *introduced time* into our application. The result of balance is an every changing value. There is no way to go back if we add a bad value. There is no way of knowing what the value was 6 edits ago.

Using a transaction log

Now let's look at the Church model for doing the same thing:

```
transactions = []

def balance
  transactions.reduce(:+)
end

transactions << 1.5
```

The balance is now a calculation. It is no longer dependent on time. We don't ever *overwrite* the balance. We now have a list of values that can be positive or negative and when you sum them all together (the call to reduce) you get the balance as it is *now*.

You could add another transaction, and then do another calculation minus the recently added transaction and you can get the balance from before you added that new transaction.

That is the big difference between the two models of computing. In the Turing model there is 'no going back', and in the Church model you can essentially have your Delorean and drive it too.

Is Ruby a Functional Programming Language?

Ruby is *almost* a functional language. You can write a lot of purely functional code in it, but the downside is that the language was built with side-effects and mutations in mind. So writing purely functional code will get you a lot more mileage in something like Haskell or Clojure, those languages are not very good for newer programmers, or for that matter, as a tool to teach functional programming.

So why write Funcional Ruby at all?

We write so called functional code to make *our own lives* easier. Because the program doesn't actually change state while it runs, it is much easier to debug. You can simply focus on the inputs and outputs of a specific function without looking at the state of the program as a whole.

If the functions you are trying to reason about don't rely on variables whos scopes can change outside your function, you can learn everything you need to know about the behavoir at hand, because it is all isolated to the specific location where the behavior happens.

Side effecting code, or "stateful code" as I will call it from now on, on the other hand has operations and changes ocurring from far away from the locality of the code we are debugging. Whenever that is the case, instead of just looking at the function in front of you, you have to consider the *program state* as a whole, along with the function you want to debug.