THE FUNCTIONAL RUBYIST

Jim OKelly

Table of Contents

The Functional Rubyist

By Jim OKelly

Dedication

> This book is dedicated to my friend and mentor, Jim Weirich. Before him I was just a coder. Jim opened my eyes to the concept of design and I will forever be grateful.

When we lost Jim Weirich we lost not only a funny, respectful, genius, daring, and super kind Rubyist, I fell like we lost a bit of our soul.

This book is a direct result of the Quest dear Jim put me on when he showed me the concept of lazy evalutaion through lambdas and that Ruby was just another LISP, and that LISP was absolutely amazing for the brain.

I miss you big guy. I can only hope you would dig this book if you were still here a strummin your euk.

Acknowledgements

First off, thank you Matz. Thank you for bringing us such a Frankenstein of a language. The sheer genious of giving us a programming language that truely takes the best ideas out there (at the time) and jamming them together.

Thank you Eiffel, LISP, and SmallTalk for giving Ruby it's best stuff.

Thank you Phil Cohen, who constantly challenged me to do better and to stick with shit. Thanks for all the chill out sessions where we just kicked it and talked.

You are an amazing curator of good ideas. An hour in your head is worth more than it's weight in gold. ;)

Thank you Gary Bernhardt for putting your thoughts together into Destroy All Software

and for having paid attention to the stuff going on around you to be another amazing curator of people and ideas. Functional core, imperative shell saved my life.

Thank you Corey Haines for knowing Gary, and also knowing just about anyone worth knowing. Your involvement in XP led me onto J.B. Rainsberger and Kent Beck. I coded with you in India when you were a travelling pair programmer. I had never thought of creative constraints until then. Blew my mind.

Thank you Uncle Bob Martin, for hanging out with Rubyists at least for awhile. If I skipped everything you ever said about programming and just took the rest, my life with still be forever enriched. And you are right, What Killed SmallTalk could kill Ruby.

I have been deeply effected by a lot of things from inside and out of the Ruby community. As is the case in life, there were a few people who turned me on to better people. There is far more to list, but I have that post available online at blog.rubymentor.io so I won't take up more ink here to do so.

Aims, Goals, and Promises

This book is not an answer to any question. It is not a bible for any solution. It is an exploration of trying to make code you love instead of code you hate. It might have more prenthesis than you were previously comfortable with, and at times it might not look like any Ruby you have ever seen before.

I don't even advocate you use anything found here anywhere. However, if you choose to do so, I think you will find that you can make a lot more sense of your code 2, 3, 12 days later, and that instead of a big mess that constantly needs debugging, you will instead find that you eventually will need less tests and the code works more than it fails.

It will almost always take up more memory than the imperative solution, except when it doesn't, so benchmark and experiment. Using laziness can make things run a lot faster with less memory, but writing mostly non-mutable code does have the side effect of more objects and data in memory.

All in all, this kind of code led me personally to writing more functionality with less code and far less bugs. I envy all of you who gets to learn these lessons early in your careers.

No animals were injured in the writing of this book, however a lot of flowers were burned.

Intro to Functional Programming

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".

Functional Core - Imperative Shell

Ruby is almost an entirely Object Oriented language built on top of some Functional core concepts. It is a dirty mix of the two as a friend of mine once put it. While this prevents you from writing the kind of "pure functional code" you would write in Haskell or Clojure, we use a paradigm called, Functional Core - Imperative Shell.

This concept was first introduced to me by Gary Bernhardt, a polyglot who once has gave the Ruby community some really great ideas (before he left our ranks for greener grasses). The concept breaks down like so:

Since a program that changes *no* state isn't much more useful than a calculator, we acknowledge that our program will change *some* state. But we will keep the vast majority of our functions pure.

> A pure function is a method that takes arguments and does some calculation, then returns a new value. It does not alter any program state *outside* itself.

If your function simply takes arguments and returns values, it not only becomes a lot easier to rationalize, it brings it a lot closer to being able to run on multiple cores without creating a disaster.

This is because if the functions don't change any state around them, they can essentially be offloaded to different processors and

Don't 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.

Whiel 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 Functional Ruby at all?

> We write so called functional code to make *our own lives* easier.

By writing the program so it doesn't constantly change state while it runs, it becomes 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 who's 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.

In constrast, side effecting code, or "stateful code" as I will call it from now on, 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 whole *program state*, along with the function you actually want to debug.

Core Functional Concepts