Technical Explorations

by Keith R. Bennett

Intro to Functional Programming in Ruby

Nov 5, 2012 • keithrbennett

Ruby is a flexible and versatile language. Although it's almost always used as an object oriented language, it can be used for functional programming as well.

In versions prior to Ruby 1.8, doing so was more awkward because there would be a lot of `lambdas` cluttering the code. In 1.9, however, we have the `->` shorthand, which makes functional style code more concise and more similar to traditional FP languages.

This post is inspired by Jim Weirich's keynote at RubyConf in Denver last Friday (Nov. 2, 2012), in which he abundantly illustrated Ruby's FP abilities. His code looked so different from most Ruby code that one attendee entering late whispered to the person next to him, *what language is that?*

Here's a walk through some basic functional programming in Ruby. A file containing the source code for this article, and some `puts` statements to illustrate the code, is here.

We'll start with some simple examples and work up to creating the execution of a workflow defined as an array of lambdas.

An "Add" Function

First, here's a simple function that returns the sum of two numbers.

```
>add = ->(x, y) { x + y }
```

The lvalue is `add`, and is a variable that will contain a reference to the lambda, or function.

The rvalue is `->(x, y) { x + y }`, and represents a function.

The `->()` indicates that this a function, and the terms inside the parentheses are the arguments it expects. The code within the curly braces is the body of the function. A very important thing to keep in mind is that the function is created and returned, but not evaluated (called). It could be considered a function literal, as we have array, hash, and regex literals in Ruby. In this case, we're assigning it to the variable `add`.

This is called like a regular function, except that we need a dot after the variable name to tell the Ruby interpreter that this is a proc and not a class' member function. Another way of looking at it is that the dot is a shorthand for `.call`, which was required for calling a lambda in pre-1.9 versions of Ruby. This is how it would look in irb:

```
>1.9.3-p286 :001 > add = ->(x, y) { x + y }
 => #<Proc:0x007fdddb25c680@(irb):1 (lambda)>
1.9.3-p286 :002 > add.(3,4)
 => 7
```

A better implementation of add, that would take a variable number of arguments is:

```
>add = ->(*numbers) { numbers.inject(:+) }
```

A “Multiple” Function

Now, let’s create a function that returns a function that will return multiples of a number:

```
>mult = ->(multiplier) { ->(n) { multiplier * n } }
```

The function will be stored in the `mult` variable. We can then call `mult` to get a function that will double its argument:

```
>double = mult.(2)
```

`mult` and `double` are instances of class `Proc`. We can now call double:

```
>double.(111)  # 222
```

Similarly, we can create a function named `power` that raises a number to a specified power, and then, using `power`, create functions `square` and `square_root` that return the square and square root of a number, respectively:

```
>power = ->(exponent) { ->(n) { n ** exponent } }
square = power.(2)
square_root = power.(0.5)
```

This practice of calling a function that takes `n` parameters to create a new function that only requires `< n` (some of) those parameters is called *currying*.

A “Hypotenuse” Function

We can now assemble a hypoteneuse function like this:

```
>hypoteneuse = ->(a, b) { square_root.(square.(a) + square.(b)) }
```

A “Chain” Function

Now let’s compose a function that will chain functions together:

```
>chain = ->(*procs) { ->(x) { procs.inject(x) { |x, proc| proc.(x) } } }
```

It’s a little complex, but if we go from the inside out it’s more manageable:

```
>procs.inject(x) { |x, proc| proc.(x) }
```

`procs` is an array of functions. We call `inject` to successively call each function with the value returned by the previous one. Finally, the last return value (stored in `x`) is returned.

```
>->(x) { *** }
```

In the code above I’ve replaced the previous code example with `***` so you can see what was added to it. We’ve wrapped the expression in a lambda that expects a single argument that will be referred to in the lambda as `x`.

```
>chain = ->(*procs) { *** }
```

Here, we define a function named `chain` that will take 0 or more arguments and assemble them into an array named `procs`, which will then be accessible to the `inject` in the inner code. Using the chain function, we can create a function that doubles then squares an argument:

```
>double_then_square = chain.(double, square)
```

A File Writer and CSV Parser

For later examples, we'll need a file containing the text "fruit,mango". Let's write a lambda that will do that, and then call it:

```
>write_file = ->(filespec, contents) { File.write(filespec, contents) }
write_file.('favorites.txt', 'fruit,mango')
```

Now let's write a trivially simple (and admittedly inadequate for real world use) CSV (comma separated values) parser lambda:

```
>parse_csv = ->(string) { string.split(',') }
```

A "Favorite" Class

For the purposes of this example, we'll need a `Favorite` class and a formatter and a parser for it:

```
>Favorite = Struct.new(:type, :instance)

format_favorite = ->(favorite) { "Favorite #{favorite.type} is #{favorite.instance}" }

parse_favorite = ->(string) {
  fav = Favorite.new
  fav.type, fav.instance = *parse_csv.(string)
  fav
}
```

Assembling a Workflow

First let's write a couple of utility functions:

```
>read_file_lines = ->(filespec) { File.readlines(filespec) }
first = ->(object) { object.first }
```

Here is the workflow we have defined. Note that although it is executable code, it is implemented as an array of objects.

```
>transformations = [
  read_file_lines,
  first,
  parse_favorite,
  format_favorite
]
```

Now we curry `chain` to create a function `transform_chain` that will execute the transformations we want:

```
>transform_chain = chain.(*transformations)
```

Then, we call the function to get the final result:

```
>result = transform_chain.('favorites.txt')
# Result will be: "Favorite fruit is mango."
```

This could also have been expressed more succinctly by removing the `transform_chain` intermediate variable:

```
>result = chain.(*transformations).('favorites.txt')
```

Conclusion

You may be wondering about the value of functional programming, thinking that it's merely an alternate implementation, maybe even a regression from object oriented programming. Unfortunately, I'm not that knowledgeable about it and don't have too much wisdom to offer. However, I can guess at these advantages:

1. One of the challenges of multithreaded programming is minimizing the risk that running code in one thread modifies data used by another. Objects are designed to carry state with them, whereas functions typically do not (or if they do, it is nonmodifiable state not accessible to code outside of the function). Using functions rather than conventional objects may therefore be a better choice in cases where this is important.
2. ETL products on which I've worked use objects with a `run` method for assembling workflows. This is nice because they can carry state (this can be a good thing too) and can enjoy the design benefits of inheritance. However, the the lambda approach is a more lightweight method.
3. Because lambdas are simply objects in memory, and are not bound to classes, they are much more easily manipulable. The resulting metaprogramming possibilities dwarf Ruby's built-in metaprogramming, itself no slouch. Of course, leveraging the "code is data" approach of lambdas may result in code that is more complex and obtuse.

As always, it depends.

Feel free to comment, enlighten, correct, etc.

- Keith Bennett

Published with GitHub Pages