Blocks, Procs and Lambdas, Oh My!

*f*unctional Programming in Ruby

by Paul Barry

What is Functional Programming?

A programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.

-- Wikipedia

Keywords here are style and emphasizes. Functional programming is a style. If you want to, you can program in a functional style in almost any language. A language is considered functional if it has features that emphasize programming in a functional style.

All programming languages are opinionated

Languages are opinionated in their design. Language designers build support into the syntax and core libraries of the language for the techniques that they think should be emphasized. The things that are emphasized by a language drives what becomes idiomatic in that language.

Functional Programming Techniques

- λ First-class Function
- λ Higher-order Functions
- λ Immutability
- λ Recursion
- λ Pure Functions (side-effect free)

These are some of the most common programming techniques found in functional programming languages.

Functional Programming Techniques in Ruby

- First-class Function
- Higher-order Functions
- Immutability
- Recursion
- Pure Functions (side-effect free)

The only two of these functional programming techniques that Ruby emphasizes are first-class functions and higher-order functions. Other techniques can be used in Ruby, but are used more commonly in other languages because they are emphasized by the language's design. I'm going to cover the first two techniques are they are used in Ruby. For the rest, I'm going to show how they are baked-in to other languages and how you can use them in Ruby, despite the fact that Ruby has no explicit support for them.

First-class functions

- Assigned to variables
- Stored in data structures

Higher-order functions

- Passed as arguments to other functions
- Return other functions as a result

Ruby’s First-Class Functions: Proc

- Functions represented by Proc objects
- Two ways to define Procs, different semantics

```
> p = Proc.new {|x,y| x.to_i + y.to_i }
  => #<Proc:0x00000001011c5d20@(irb):14>

> l = lambda {|x,y| x.to_i + y.to_i }
  => #<Proc:0x00000001011b98e0@(irb):15>
```

Since a proc and a lambda have different semantics, they are commonly referred to as “a proc” and “a lambda”, even though they are both objects that are instances of the Proc class

Ruby's Higher-Order Functions: Blocks

```ruby
5.times do
  puts "hello"
end
```

Matz could have left it at that, but added block syntax because he wanted to put an emphasis on higher-order functions, especially the case of passing one function

All methods can have a block

```
"foo".reverse do
  puts "Never Gonna Happen"
end
```

Calling the block

```
class Fixnum
  def times
    for n in 0...self
      yield n
    end
  end
end
```

Name your block

```
def m(&block)
  p block
end

> m {}
#<Proc:0x0000000000000000@(irb):1>
```

Imperative

```
squares = []
for n in 0..3
  squares << n * n
end
p squares # => [0, 1, 4, 9]
```

Here we are building up an array using an imperative style, by creating an intially empty array and then mutating that array by appending a value to it on each iteration of the loop

Functional

```
> p (0..3).map{|n| n * n }
 => [0, 1, 4, 9]
```

This is functional, we calculate the new array by applying the function that is specified in the block to each value in the array. In this sense, Ruby is functional in that it gives us the block syntax, thereby encouraging the use of higher-order functions

Wrapping with Blocks

```
def trace(message)
  puts "start #{message}"
  yield
  puts "end #{message}"
end
```

```
def foo
  puts "in foo"
end

trace "foo" do
  foo
end

start foo
in foo
end foo
```

Lots of nice ways to use blocks

Before/After Functionality

```
def silence
  begin
    original_log_level = Logger.level
    Logger.level = :error
    yield
  ensure
    Logger.level = original_log_level
  end
end
```

Custom Initialization

```
class Person
  attr_accessor :name
  def initialize(name, &block)
    @name = name
    instance_eval(&block)
  end
end

p = Person.new("HELLO") do
  @name.downcase!
end

puts p.name # => hello
```

Immutability

Values > References

What does this return?

```
x = 42
foo x
puts x # => 42
```

Integers are immutable values in Ruby

What does this do?

```
x = [1, 2, 3]
foo x
p x # => [1, 2, 3, :wtf]
```

x is a reference to a mutable array

Clojure

- Lisp Dialect for the JVM
- Emphasis on immutability

Functional programming language specifically designed to address this problem

What does this do?

```
(def x [1 2 3])
(foo x)
(println x) ; => [1 2 3]
```

Why?

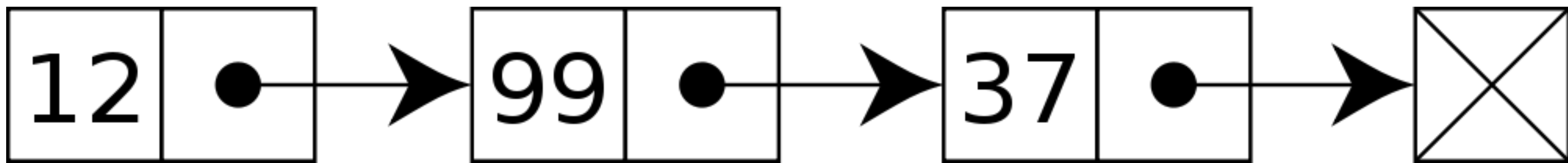
- All core data structures in Clojure are immutable
- Numbers, Strings, Lists, Arrays, Hashs, Sets, etc.
- Doesn't that suck?

It would suck if to produce a new array you had to duplicate the whole array and then modify that copy. Luckily in Clojure you don't, Persistent Data Structures FTW

Persistent Data Structures

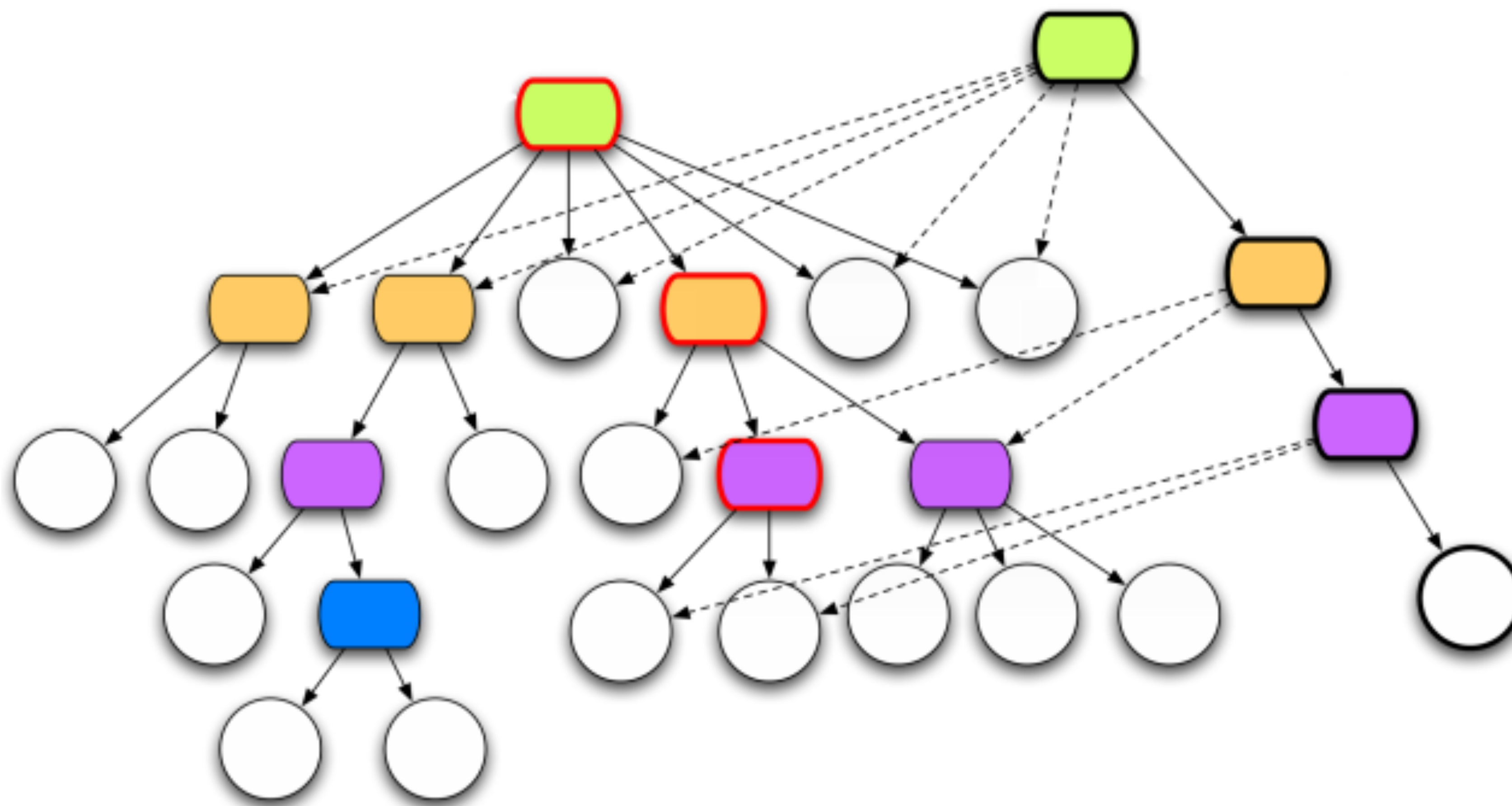
- Nothing to do with a database
- Immutable
- Produce new values by sharing structure with existing values

Linked List



All nodes are immutable. You can append to this list in constant time. Each list shares structure with the previous lists.

Hashes with Structural Sharing



How it actually works is hard to understand, but the important things is that this is a Hash that can produce a new Hash in close to constant time by sharing structure with existing hash

“Modifying” a Hash

```
user=> (def x {:x 1, :y 2})
#'user/x
user=> (def y x)
#'user/y
user=> (def x (assoc x :z 3))
#'user/x
user=> x
{:z 3, :x 1, :y 2}
user=> y
{:x 1, :y 2}
```

Hamster

```
> require "hamster"
 => true
> x = Hamster.hash :x => 1, :y => 2
 => {:x => 1, :y => 2}
> y = x
 => {:x => 1, :y => 2}
> x = x.put :z, 3
 => {:x => 1, :y => 2, :z => 3}
> y
 => {:x => 1, :y => 2}
```

Requires Ruby 1.8.7 or greater. Not used throughout Ruby, so the value of this is limited.

Immutable Objects: Scala

- Functional / Object-Oriented Hyrbid
- JVM
- Immutable objects

Clojure is not object-oriented, only functional. Scala is a good language to look at for inspiration in this category. Scala is a mix of functional and object-oriented. Has built in support for immutable objects.

Example

```
class Point
  attr_reader :x, :y
  def initialize(x, y)
    move_to(x, y)
  end
  def move_to(x, y)
    @x = x
    @y = y
  end
  def to_s
    "(#{x}, #{y})"
  end
end
```

```
p = Point.new(1,1)
puts p # => (1, 1)
p.move_to(2, 3)
puts p # => (2, 3)
```

Notice that you can’t manipulate x and y directly because Point has no setters for those attributes. Well, actually you can manipulate them with instance_variable_set.

What happens now?

```
p = Point.new(1,1)
foo p
puts p # => ?????
```

Immutable Object

```
class Point(x: Int, y: Int) {
  val x = x
  val y = y
  def moveTo(x: Int, y: Int): Point = {
    new Point(x, y)
  }
  override def toString(): String = {
    "(" + x + ", " + y + ")"
  }
}
```

You can do this in other languages like Java and Ruby, but the fact that it is emphasized in Scala by being built into the language makes it’s usage more idiomatic. Not as efficient as persistent data structures, but may be good enough for your use case.

Immutable Point

```
class Point
  attr_reader :x, :y
  def initialize(x, y)
    @x = x
    @y = y
  end
  def move_to(x, y)
    self.class.new(x, y)
  end
  def to_s
    "(#{x}, #{y})"
  end
end
```

```
p = Point.new(1,1)
puts p # => (1, 1)
p = p.move_to(2, 3)
puts p # => (2, 3)
```

That idea applied to Ruby

Now what happens?

```
p = Point.new(1,1)
foo p
puts p # => ?????
```

Not really enforced by Ruby, but since move_to doesn’t change the p, it’s less likely to be mutated

Recursion

```
def sum(n, acc=0)
  n <= 1 ? n + acc : sum(n-1, n+acc)
end
```

```
> (0..10).map(&(method(:sum)))
 => [0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
> sum(10000)
SystemStackError: stack level too deep
```

Recursive Execution

```
def sum(n, acc=0)
  if n <= 1
    puts caller
    n + acc
  else
    sum(n-1, n+acc)
  end
end

p sum(5)
```

```
sum.rb:6:in `sum'
sum.rb:6:in `sum'
sum.rb:6:in `sum'
sum.rb:6:in `sum'
sum.rb:10:in `<main>'
15
```

Only Ruby 1.9 includes the recursive calls in “caller”, even though they are obviously in there in Ruby 1.8

Tail-Call Optimization: Erlang

```
-module(sum).
-export([sum/1]).

sum(N) -> sum(N,0).

sum(0, Acc) -> 0 + Acc;
sum(1, Acc) -> 1 + Acc;
sum(N, Acc) -> sum(N-1, N+Acc).
```

```
1> l(sum).
{module,sum}
2> lists:map(fun sum:sum/1, [0,1,2,3,4,5,6,7,8,9,10]).
[0,1,3,6,10,15,21,28,36,45,55]
3> sum:sum(10000).
50005000
```

Erlang is a functional language with it's own VM, not based on JVM like Clojure.

The designers of Erlang put Tail-Call Optimization into the language because they wanted to emphasize programming in a recursive style

Pure Functions: Beyond Immutability

- Pure functions are side-effect free
- Take values as input and compute a value
- Do not make changes to the state of the system
- Do not depend on time, randomness, file system, network connections, etc.

Is foo a pure function?

```
(def x [1 2 3])
(foo x)
```

There is no way to tell. You would have to look at the definition of foo and all of the functions that it calls.

Haskell

```
Prelude> let square x = x * x
Prelude> :t square
square :: (Num a) => a -> a
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

Haskell is a functional language. Unlike Ruby, Clojure and Erlang, it is a statically typed language. It uses it's type system to enforce some constraints. Pure functions are built into the language, so there is a clear emphasis on using pure functions. Using pure functions is strongly encouraged in all functional languages, but is enforced in Haskell. Side-effects are a necessary evil in programming, but it's nice to know where they are

Dealing with Side Effects in Ruby

- Create as many pure functions as possible
- Make it obvious where side-effects occur

Summary

- Use blocks in your own code to build abstractions
- Pure functions are easier to reason about and to test
- Learning functional languages like Clojure, Erlang and Haskell will make you a better Ruby programmer

Questions?