Functional Ruby

Rob Biedenharn
Gaslight Software

gaslightsoftware

Functional Ruby

Rob Biedenharn
Gaslight Software

If you can read this, then we're ready to go!

Functional Ruby

Rob Biedenharn
Gaslight Software

If you can read this, then we're ready to go!

Functional Ruby

gaslightsoftware

Functional Ruby

- Ruby is … “A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.”

— From http://www.ruby-lang.org/en/

gaslightsoftware

Functional Ruby

- Ruby is … “A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.”
  — From http://www.ruby-lang.org/en/
- Ruby is definitely an Object-Oriented language and *not* a Functional Language.

gaslightsoftware

Functional Ruby

- Ruby is … “A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.”
  — From http://www.ruby-lang.org/en/
- Ruby is definitely an Object-Oriented language and *not* a Functional Language.
- *But …*

gaslightsoftware

Functional Ruby

- Ruby is a language of careful balance. Its creator, Yukihiro "matz" Matsumoto, blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that **balanced functional programming with imperative programming**. *(emphasis added)*
- He has often said that he is "trying to make Ruby natural, not simple," in a way that mirrors life.

Building on this, he adds:

—*Ruby is simple in appearance, but is very complex inside, just like our human body.*

- Matz, speaking on the Ruby-Talk mailing list, May 12th, 2000.

Functional Ruby

- The Haskell site begins with a definition: "What is functional programming?" which begins:
  - In functional programming, programs are executed by evaluating *expressions*, in contrast with imperative programming where programs are composed of *statements* which change global state when executed. Functional programming typically avoids using mutable state.
- It goes on, but if we stop there (for a moment), then Ruby starts looking a bit functional, too.

All statements are expressions

- All “statements” in Ruby are really just expressions that have a value.

```
irb1.9.2> foo = 42
#1.9.2 => 42
irb1.9.2> if foo > 20
            "too high"
          elsif foo > 10
            "need my toes"
          else
            "that's easy"
          end
#1.9.2 => "too high"
```

All statements are expressions

```
irb1.9.2> f = lambda {|foo|
                if foo > 20
                  "too high"
                elsif foo > 10
                  "need my toes"
                else
                  "that's easy"
                end
              }
#1.9.2 => #<Proc:0x136fa0@(irb):10 (lambda)>
irb1.9.2> f[9]
#1.9.2 => "that's easy"
irb1.9.2> f.call(15)
#1.9.2 => "need my toes"
irb1.9.2> foo
#1.9.2 => 42
```

gaslightsoftware

All statements are expressions

- For methods, the “value” is just the last expression that was evaluated unless there is a **`return`** statement.

```
def answer
  42
end
```

```
def answer
  return 42
end
```

- These have the same value when called, but Ruby is actually optimized such that the implicit value is *slightly* more performant.

gaslightsoftware

Functional Ruby

- "Higher-order functions (HOFs) are functions that take other functions as their arguments. A basic example of a HOF is **`map`** which takes a function and a list as its arguments, applies the function to all elements of the list, and returns the list of its results."
—Haskell site

Higher-order Functions

- Ruby has a construct called a **block** which acts as a closure and is “passed” to a method.

```
irb1.9.2> list = [ 8, 17, 29 ]
#1.9.2 => [8, 17, 29]
```

```
irb1.9.2> list.map {|element| element * 2 }
#1.9.2 => [16, 34, 58]
```

Higher-order Functions

- With a little bit of syntax, that could be any lambda (even our little *f* from before):

```
irb1.9.2> list
#1.9.2 => [8, 17, 29]
```

```
irb1.9.2> list.map &f
#1.9.2 => ["that's easy", "need my toes", "too high"]
```

gaslightsoftware

Higher-order Functions

- Like **`map`** (aka, **`collect`**) which applies a function to a list to obtain a new list, there is **`reduce`** (aka, **`inject`**) which applies a function across a list to obtain a single value.

```
irb1.9.2> list
#1.9.2 => [8, 17, 29]
irb1.9.2> list.reduce {|sum, element| sum + element}
#1.9.2 => 54

irb1.9.2> list.reduce(&:+)
#1.9.2 => 54
```

gaslightsoftware

Higher-order Functions

Higher-order Functions

- Oh, can we use our own favorite “function” *ƒ* ?

Higher-order Functions

- Oh, can we use our own favorite "function" *ƒ* ?
- Well, not quite:

Higher-order Functions

- Oh, can we use our own favorite “function” *f* ?
- Well, not quite:

```
irb1.9.2> list.reduce &f
ArgumentError: wrong number of arguments (2 for 1)
   from (irb):11:in `block in irb_binding'
   from (irb):31:in `each'
   from (irb):31:in `reduce'
   from (irb):31
   from /Users/rab/.rvm/rubies/ruby-1.9.2-p180/bin/
irb:16:in `<main>'
```

gaslightsoftware

Higher-order Functions

- But, we could do this:

```
irb1.9.2> list.map(&f).reduce(&:+)
#1.9.2 => "that's easyneed my toestoo high"
```

Higher-order Functions

- But, we could do this:

```
irb1.9.2> list.map(&f).reduce(&:+)
#1.9.2 => "that's easyneed my toestoo high"
```

- Did you catch that?

gaslightsoftware

Higher-order Functions

- But, we could do this:

```
irb1.9.2> list.map(&f).reduce(&:+)
#1.9.2 => "that's easyneed my toestoo high"
```

- Did you catch that?

`list.reduce(&:+)` used '**+**' to add numbers

gaslightsoftware

Higher-order Functions

- But, we could do this:

```
irb1.9.2> list.map(&f).reduce(&:+)
#1.9.2 => "that's easyneed my toestoo high"
```

- Did you catch that?

`list.reduce(&:+)` used '**+**' to add numbers

`list.map(&f).reduce(&:+)` used '**+**' to concatenate strings

gaslightsoftware

Iteration, Recursion, Tail-recursion

gaslightsoftware

Iteration, Recursion, Tail-recursion

- Most iteration is accomplished with an object's **`each`** method which is the key to all the features from the **`Enumerable`** module.
  - Since Ruby 1.9.2, most enumerations called without a block return an **`Enumerator`** instance.

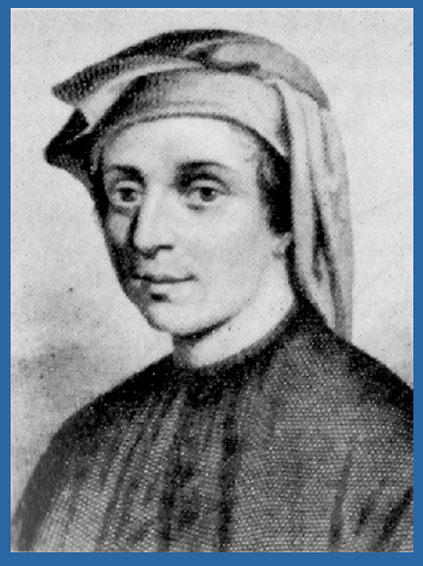
Iteration, Recursion, Tail-recursion

- Most iteration is accomplished with an object's **`each`** method which is the key to all the features from the **`Enumerable`** module.
  - Since Ruby 1.9.2, most enumerations called without a block return an **`Enumerator`** instance.
- What better to illustrate these techniques than…

gaslightsoftware

Iteration, Recursion, Tail-recursion

- Most iteration is accomplished with an object's **`each`** method which is the key to all the features from the **`Enumerable`** module.
  - Since Ruby 1.9.2, most enumerations called without a block return an **`Enumerator`** instance.
- What better to illustrate these techniques than…
- **The Fibonacci Sequence**



gaslightsoftware

Iteration, Recursion, Tail-recursion

```
irb1.9.2> class Integer
            def fib_i
              a, b = 0, 1
              self.times do
                a, b = b, a+b
              end
              a
            end
          end
```

```
irb1.9.2> (0..10).map {|number| number.fib_i }
#1.9.2 => [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

gaslightsoftware

Iteration, Recursion, Tail-recursion

```
irb1.9.2> class Integer
            def fib_r
              if self < 2
                self
              else
                (self - 2).fib_r + (self - 1).fib_r
              end
            end
          end
```

```
irb1.9.2> (0..10).map {|number| number.fib_r }
#1.9.2 => [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

gaslightsoftware

Iteration, Recursion, Tail-recursion

```
irb1.9.2> class Integer
            def fib_t
              fib_t_helper(self, 0, 1)
            end

            def fib_t_helper(n, a, b)
              if n.zero?
                a
              else
                fib_t_helper(n-1, b, a+b)
              end
            end
          end
irb1.9.2> (0..10).map {|number| number.fib_t }
#1.9.2 => [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Iteration, Recursion, Tail-recursion

Why does it matter?

```
require 'benchmark'
include Benchmark

bm(14) do |x|
  class Integer
    …
  end
  x.report("iteration 1000") {1000.times {|i| i.fib_i } }
  x.report("recursion 30  ") {  30.times {|i| i.fib_r } }
  x.report("recursion 40  ") {  40.times {|i| i.fib_r } }
  x.report("tail-rec. 1000") {1000.times {|i| i.fib_t } }
end
```

gaslightsoftware

Iteration, Recursion, Tail-recursion

gaslightsoftware

Iteration, Recursion, Tail-recursion

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000  0.900000   0.010000   0.910000 (  0.907310)
```

Iteration, Recursion, Tail-recursion

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000   0.900000   0.010000   0.910000 (  0.907310)
recursion 30     1.360000   0.010000   1.370000 (  1.371345)
```

Iteration, Recursion, Tail-recursion

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000   0.900000   0.010000   0.910000 (  0.907310)
recursion 30     1.360000   0.010000   1.370000 (  1.371345)
recursion 40   167.110000   0.840000 167.950000 (168.156662)
```

Iteration, Recursion, Tail-recursion

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000   0.900000   0.010000   0.910000 (  0.907310)
recursion 30     1.360000   0.010000   1.370000 (  1.371345)
recursion 40   167.110000   0.840000 167.950000 (168.156662)
tail-rec. 1000   0.740000   0.000000   0.740000 (  0.746140)
```

Iteration, Recursion, Tail-recursion Memoization!

- In computing, **memoization** is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously-processed inputs.

  —Wikipedia

- Since pure functional programming means that an expression (or function) with the same input(s) **always** produces the same value.

gaslightsoftware

Iteration, Recursion, Tail-recursion

Memoization!

```
irb1.9.2> class Integer
            @@fibs = [0,1]
            def fib_m
              @@fibs[self] ||= ((self - 2).fib_m +
                                (self - 1).fib_m)
            end
          end
```

```
irb1.9.2> (0..10).map {|number| number.fib_m }
#1.9.2 => [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

gaslightsoftware

Iteration, Recursion, Tail-recursion

Why does it matter?

```
require 'benchmark'
include Benchmark

bm(14) do |x|
  class Integer
    …
  end
  x.report("iteration 1000") {1000.times {|i| i.fib_i } }
  x.report("recursion 30  ") {  30.times {|i| i.fib_r } }
  x.report("recursion 40  ") {  40.times {|i| i.fib_r } }
  x.report("tail-rec. 1000") {1000.times {|i| i.fib_t } }
  x.report("memoized  40  ") {  40.times {|i| i.fib_m } }
  x.report("memoized  1000") {1000.times {|i| i.fib_m } }
end
```

gaslightsoftware

Iteration, Recursion, Tail-recursion

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000   0.900000   0.010000   0.910000 (  0.907310)
recursion 30     1.360000   0.010000   1.370000 (  1.371345)
recursion 40   167.110000   0.840000 167.950000 (168.156662)
tail-rec. 1000   0.740000   0.000000   0.740000 (  0.746140)
```

Iteration, Recursion, Tail-recursion Memoization!

Iteration, Recursion, Tail-recursion Memoization!

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000   0.900000   0.010000   0.910000 (  0.907310)
recursion 30     1.360000   0.010000   1.370000 (  1.371345)
recursion 40   167.110000   0.840000 167.950000 (168.156662)
tail-rec. 1000   0.740000   0.000000   0.740000 (  0.746140)
```

Iteration, Recursion, Tail-recursion Memoization!

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000   0.900000   0.010000   0.910000 (  0.907310)
recursion 30     1.360000   0.010000   1.370000 (  1.371345)
recursion 40   167.110000   0.840000 167.950000 (168.156662)
tail-rec. 1000   0.740000   0.000000   0.740000 (  0.746140)
memoized  40     0.000000   0.000000   0.000000 (  0.000153)
```

Iteration, Recursion, Tail-recursion Memoization!

```
irb1.9.2> load 'fib_benchmark.rb'
                      user     system      total        real
iteration 1000   0.900000   0.010000   0.910000 (  0.907310)
recursion 30     1.360000   0.010000   1.370000 (  1.371345)
recursion 40   167.110000   0.840000 167.950000 (168.156662)
tail-rec. 1000   0.740000   0.000000   0.740000 (  0.746140)
memoized  40     0.000000   0.000000   0.000000 (  0.000153)
memoized  1000   0.010000   0.000000   0.010000 (  0.003212)
```

Functional Ruby

- “Infinite Streams”
  —JEG2 site

Infinite Streams

- Like in the Scala example last month, we can define a stream that is lazily evaluated by knowing the next value and having a way to get the rest when it is needed.

```
module LazyStream
  class Node
    def initialize( head, &promise )
      @head, @tail = head, promise
    end
  end
end
```

Infinite Streams

- Then we can define the parts:

```
attr_reader  :head

def tail
  if @tail.is_a? Proc
    if @tail.arity == 1
      @tail.call(head)
    else
      @tail.call
    end
  else
    @tail
  end
end
```

Infinite Streams

- And a way to take an item from the stream:

```
def drop
  result, next_stream = head, tail
  @head, @tail = case next_stream
                 when self.class
                   next_stream.instance_eval {
                     [@head, @tail]
                   }
                 else
                   [next_stream, @tail]
                 end
  result
end
alias_method :tail!, :drop
```

Infinite Streams

- And a way to iterate over the stream, partially.

```
def each!(limit=nil)
  loop do
    break unless limit.nil? || (limit -= 1) > -1
    yield drop
    break if end?
  end
  self
end

def each(limit=nil, &block)
  clone.each!(limit, &block)
  self
end
include Enumerable
```

gaslightsoftware

Infinite Streams

- However, if you wanted to process more than one stream simultaneously, things get a bit messier.
- Fortunately, Ruby 1.9 introduced the **`Fiber`** class:
  - At their most basic, let you create simple generators (much as you could do previously with blocks).
    - Dave Thomas, http://pragdave.blogs.pragprog.com/pragdave/2007/12/pipelines-using.html

Infinite Streams

- Well, the Fibonacci Sequence is like that!

```
irb1.9.2> fib = Fiber.new do
            f1, f2 = 0, 1
            loop do
              Fiber.yield f1
              f1, f2 = f2, f1 + f2
            end

irb1.9.2> 11.times { print "#{fib.resume} " }; puts
0 1 1 2 3 5 8 13 21 34 55
```

gaslightsoftware

Infinite Streams

- We can define a **`Fiber`** that generates even numbers:

```
irb1.9.2> evens = Fiber.new do
            value = 0
            loop do
              Fiber.yield value
              value += 2
            end
          end
```

Infinite Streams

- And a consumer that accepts values from the generator and prints them:

```
irb1.9.2> consumer = Fiber.new do
            10.times do
              next_value = evens.resume
              puts next_value
            end
          end
```

- Technically, the consumer doesn't have to be a Fiber, but it will give us some flexibility.

Infinite Streams

- So, we need to kick off the **`consumer`** with resume:

```
irb1.9.2> consumer.resume
0
2
4
6
8
10
12
14
16
18
#1.9.2 => 10
```

Infinite Streams

- But we have hard-coded the name of the generator **`evens`** inside the **`consumer`**. ;-(
  - Lets's fix that!

```
irb1.9.2> def consumer(source)
            Fiber.new do
              10.times do
                next_value = source.resume
                puts next_value
              end
            end
          end
```

- And assume that we changed **`evens`** to a method, too.

Infinite Streams

- Let's also add another Fiber that is a filter that consumes values from a generator, but only yields values that are a multiple of 3.

```
irb1.9.2> def multiples_of_three(source)
        Fiber.new do
          loop do
            next_value = source.resume
            Fiber.yield next_value if next_value % 3 == 0
          end
        end
      end
```

Infinite Streams

- Of course, Fibers get better in Dave's next post: http://pragdave.blogs.pragprog.com/pragdave/2008/01/pipelines-using.html

  —*Yes, these are from 3½ years ago!*

Any Questions?

- About something I covered too quickly?
- About something I didn't cover at all?
- …

- rab@GaslightSoftware.com
- @rabiedenharn
- http://GaslightSoftware.com/

References

- **Functional Programming**
  Wikipedia (long list of citations)
  http://en.wikipedia.org/wiki/**Functional_programming**
- **Functional programming**
  From the Haskell site
  http://www.haskell.org/haskellwiki/**Functional_programming**

- **From the C2 Wiki:**
  http://c2.com/cgi/wiki?**RubyLanguage**
  http://c2.com/cgi/wiki?**FunctionalProgramming**
  http://c2.com/cgi/wiki?**FunctionalProgrammingLanguage**

References

- **Higher Order Ruby**
  A series of articles by James Edward Gray II
  http://blog.grayproductions.net/categories/higherorder_ruby
  (Note that the list is most-recent-first so start at the bottom.)
- **Pearls of Functional Algorithm Design**
  by Richard Bird
  ISBN-13: 9780521513388
  http://search.barnesandnoble.com/Pearls-of-Functional-Algorithm-Design/Richard-Bird/e/9780521513388#
- **Functional techniques in Ruby**
  Slides by @erockenjew *(no longer a Twitter name)*
  http://www.slideshare.net/erockendude/functional-techniques-in-ruby

gaslightsoftware

References

- **Thinking Functionally In Ruby**
  @tomstuart / tom@experthuman.com
  http://www.slideshare.net/RossC0/thinking-functionally-in-ruby
- **Memoization**
  Examples of memoization in various programming languages
  http://en.wikipedia.org/wiki/**Memoization**

gaslightsoftware

References

- **Structure and Interpretation of Computer Programs**
  - Video Lectures by Hal Abelson and Gerald Jay Sussman

    These twenty video lectures by Hal Abelson and Gerald Jay Sussman are a complete presentation of the course, given in July 1986 for Hewlett-Packard employees, and professionally produced by Hewlett-Packard Television. Note: These lectures follow the first edition (1985) of *Structure and Interpretation of Computer Programs*.
    http://groups.csail.mit.edu/mac/classes/6.001/abelson-sussman-lectures/

  - Companion site to the text and the video lectures.
    http://mitpress.mit.edu/sicp/

- **Higher-Order Procedures (in Ruby)**

  Slides from 13-Dec-2006 based on SICP
  http://www.slideshare.net/jashmenn/higher-order-procedures-in-ruby-15799

gaslightsoftware

Images

- http://en.wikipedia.org/wiki/File:Fibonacci.jpg
  Public Domain