Purely functional programming in Ruby

Shugo Maeda

2012-09-15T10:00:00 - 2012-09-15T10:30:00

Who am I?

ラベル

- 前田修吾
- Rubyコミッタ
- ネットワーク応用通信研究所取締役
- Rubyアソシエーション事務局長


I love fishing


But, no fish today :-(


広告
Programmers wanted!
http://www.netlab.jp/recruit/
※画像はイメージです

What is functional programming?

このプレゼンテーションでの定義

- 副作用を用いないプログラミング
- 純粋な関数プログラミング
  = まったく副作用を用いないプログラミング

純粋な関数プログラミングを部分的に使う

- Rubyでは、汚いことは普通にオブジェクト指向の流儀でやればいいでしょ

What are side-effects?

変数の再代入

```
x = 0; 3.times { x += 1 }
```

オブジェクトの状態変更

```
x = "foo"; x.upcase!
```

入出力

```
puts "hello, world"
```

Why functional programming?

生産性のため？

テストしやすい？

並列プログラミングのため？

何かかっこよさそうだから

- 努力と報酬が相関すると人は努力しなくなる

Is Ruby functional?

関数型っぽい要素

- ブロック・lambda
- ifなどが文ではなく式

でもやっぱり命令型言語

- 変数の再代入
  - 定数ですら…
- 基本的なデータ構造（String, Array, Hash）がmutable

Pseudo functional programming in Ruby

よいinject

```
a.inject(0) { |x, y| x + y }
```

わるいinject

```
a.inject({}) { |h, (k, v)| h[k] = v; h }
```

ふつうのmap

```
a.map { |i| i * 2 }
```

Persistent data structures

短命データ構造

- 変更を加えると前のデータが失われるデータ構造
- RubyのArrayやHash
- 前のデータを取っておきたい時はコピーが必要

永続データ構造

- 変更前のバージョンのデータが保存されるデータ構造
- HaskellのListやMap

関数プログラミングだと自然に永続データ構造になる

- 一度作ったデータは壊せない
- 参照しなくなったデータはGCで回収される

Is map functional?

ぱっと見は関数型に見える

```
a.map { |i| i * 2 }
```

でも実装は命令型

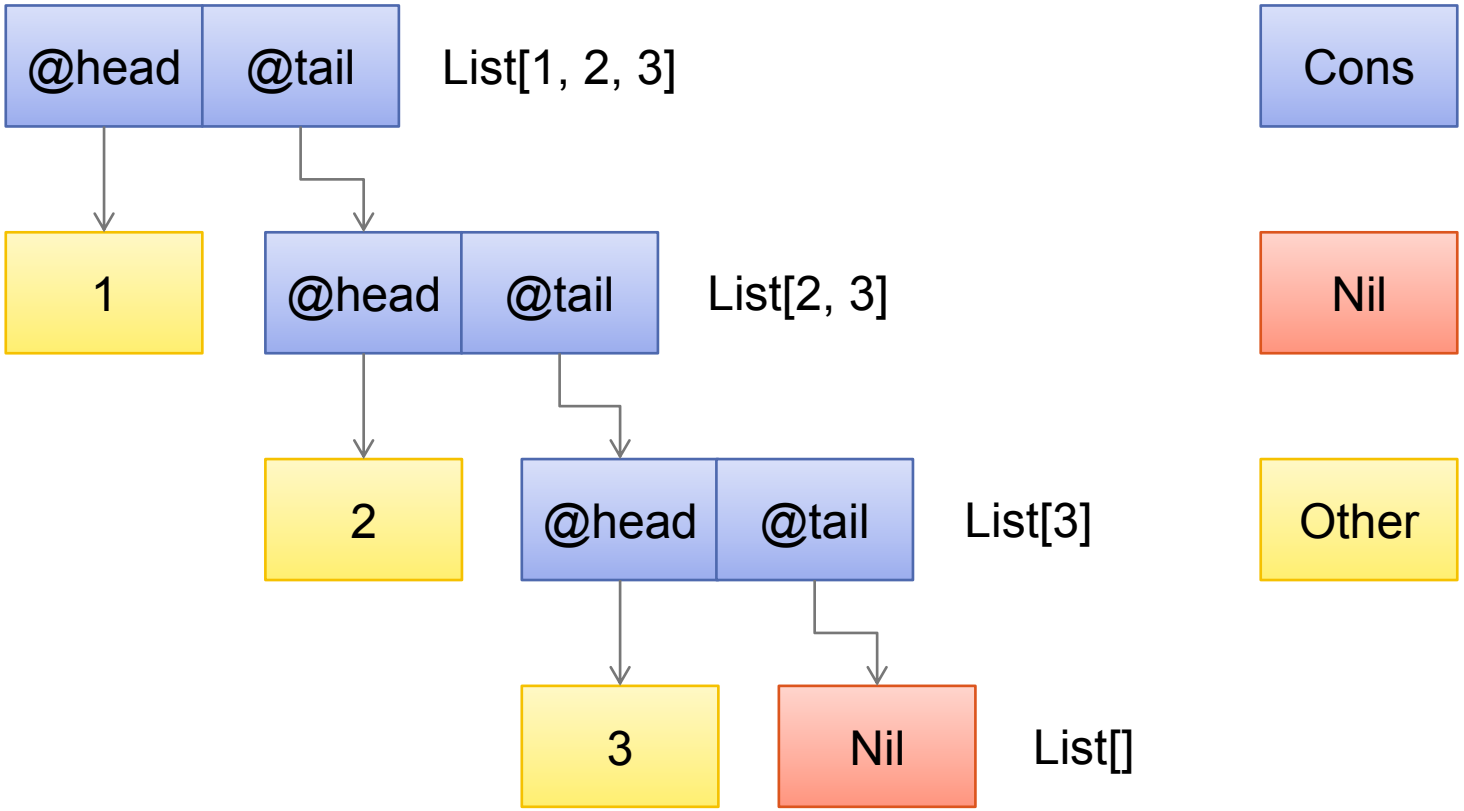
```
def map
  ary = []
  each { |i| ary.push(i) }
  return ary
end
```

Implementing lists

永続的データ構造としてリストを実装する

```
xs = List[1, 2, 3]
ys = xs.cons(0) #=> List[0, 1, 2, 3]
xs #=> List[1, 2, 3]
```

Representation of lists



Definition of classes

```
class List
end
class Cons < List
  attr_reader :head, :tail
  def initialize(head, tail)
    @head = head
    @tail = tail
  end
end
Nil = List.new
def Nil.head; raise "list is empty"; end
def Nil.tail; raise "list is empty"; end
```

Constructor

```
class List
  def self.[](*args)
    args.reverse_each.inject(Nil) { |x, y|
      Cons.new(y, x)
    }
  end
end
```

Don’t use the ‘if’ keyword

“Don’t use the ‘else’ keyword”?

- Perfecting OO's Small Classes and Short Methods
- オブジェクト指向というより命令型では?
- Haskellでは逆にelseを省略できない

本物のオブジェクト指向プログラマはifを使わない

- というのはちょっとおおげさ

Dynamic dispatching

実行時に呼び出されるメソッドが決まる

- オブジェクト指向の常套手段

関数型言語ではパターンマッチを使う

- 詳しくは次の講演で

データの種類が増えた時に対応しやすい

- if式だと元の定義の分岐を増やす必要がある

パターンマッチよりもある意味で限定的

- 再帰的なデータに深くマッチさせるのは難しい
    - ダブルディスパッチである程度可能?
- Rubyだとダックタイピングなのでより柔軟な面も

Example of dynamic dispatching

```
class Cons < List
  def empty?
    false
  end
end
def Nil.empty?
  true
end
List[].empty?  #=> true
List[1].epmty? #=> false
```

Don’t use the ‘while’ keyword

while式は副作用に依存する

```
class List
 def length
    result = 0
    xs = self
    while !xs.empty?
      result += 1
      xs = xs.tail
    end
    return result
  end
end
```

Use recursion

```
class Cons < List
  def length
    tail.length + 1
  end
end
def Nil.length
  0
end
List[1, 2, 3].length #=> 0
List[*(1..9999)].length #=> SystemStackError
```

Use tail recursion

```
class List
  def length; _length(0); end
end
class Cons < List
  def _length(n)
    tail._length(n + 1)
  end
end
def Nil._length(n)
  n
end
```

Tail call optimization

```
# 末尾呼び出しの最適化を有効化する設定
# 設定後にコンパイルされたコードにのみ有効
RubyVM::InstructionSequence.compile_option =
{
  trace_instruction: false,
  tailcall_optimization: true
}
require "list" # list.rbでは最適化が有効
List[*(1..9999)].length #=> 9999
```

Ruby 2.0ではデフォルトで有効に?

Can you make this tail recursive?

```
class Cons < List
  def map(&block)
    Cons.new(yield(head), tail.map(&block))
  end
end
def Nil.map
  Nil
end
```

Does this work well?

```
class List
  def map(&block); _map(Nil, &block); end
end
class Cons < List
  def _map(xs, &block)
    tail._map(Cons.new(yield(head), xs),
              &block)
  end
end
def Nil._map(xs); xs; end
```

Correct answer

```
class List
  def map(&block); rev_map(&block).reverse; end
  def reverse; _reverse(Nil); end
  def rev_map(&block); _rev_map(Nil, &block); end
end
class Cons < List
  def _reverse(xs)
    tail._reverse(Cons.new(head, xs))
  end
  def _rev_map(xs, &block)
    tail._rev_map(Cons.new(yield(head), xs), &block)
  end
end
def Nil._reverse(xs); xs; end
def Nil._rev_map(xs); xs; end
```

Time efficiency and Space efficiency

non-tail recursion版では時間効率が良い

- 小さい入力に有利

tail recursion版では空間効率が良い

- 大きい入力に有利

ただし、時間計算量・空間計算量はどちらもO(n)

- tail recursion版はスタックを消費しない代わりに中間リストが必要

Folding

再帰は強力だがわかりにくい

畳み込み

- リストの各要素を一つの値（リストであることもある）に畳み込む
- 再帰を直接使うより用途が限定されるが、わかりやすい
- Rubyだとinject

```
def length
  inject(0) { |x, y| x + 1 }
end
```

foldr

右からの再帰

non-tail recursion

```
class Cons < List
  def foldr(e, &block)
    yield(head, tail.foldr(e, &block))
  end
end
def Nil.foldr(e, &block)
  e
end
```

foldl

左からの再帰

tail recursion

```
class Cons < List
  def foldl(e, &block)
    tail.foldl(yield(e, head), &block)
  end
end
def Nil.foldl(e, &block)
  e
end
```

Right-associative and left-associative operations

foldrは右結合

```
List[1,2,3].foldr(0, &:+)
# 1 + (2 + (3 + 0))
```

foldlは左結合

```
List[1,2,3].foldl(0, &:+)
# ((0 + 1) + 2) + 3
```

map by foldr

```
class List
  def map(&block)
    foldr(Nil) { |x, ys|
      Cons.new(yield(x), ys)
    }
  end
end
```

Cons.newは右結合なのでfoldrを使うのが自然

map by foldl

```
class List
  def map(&block)
    rev_map(&block).reverse
  end
  def reverse
    foldl(Nil) { |xs, y| Cons.new(y, xs) }
  end
  def rev_map(&block)
    foldl(Nil) { |xs, y|
      Cons.new(yield(y), xs)
    }
  end
end
```

Feature #6242 Ruby should support lists

```
I've heard that Ruby is a LISP.

LISP stands for "LISt Processing."

Hence, Ruby should support lists.


I've attached a patch to add the classes
List and Cons, and the cons operator `:::'.
```

Listを組み込みクラスにしようという提案

Example

```
>> S[1,2,3].inject(:+)
=> 6
>> S[1,2,3]
=> S[1, 2, 3]
>> 0 ::: S[1,2,3]
=> S[0, 1, 2, 3]
>> 0 ::: 1 ::: 2 ::: 3 ::: nil # 右結合
=> S[0, 1, 2, 3]
>> S[1,2,3].inject(:+)
=> 6
```

Discussion

> > Hi,shugo
> > What benefits of the proposed lisp change
sample is very similar with native ruby Hash in m
>
> ∧,,∧
> (,,･∀･) You should say "Array", not "Hash"!
> ～(_u,uノ
(ノ⊙益⊙)ノ彡┻━━┻

--
Aaron Patterson
http://tenderlovemaking.com/

Rejected

mame (Yusuke Endoh) が5ヶ月前に更新

- ステータス を *Open* から *Rejected* に変更

> April fools' day ended. Thanks.
>
> --
> Yusuke Endoh <mame@tsg.ne.jp>

immutable

仕方がないのでgemで

```
$ gem install immutable
```

Immutableというモジュールを提供

```
require "immutable"


include Immutable
p List[1,2,3].map(&:to_s) #=> List["1", "2", "3"]
```

Lazy evaluation

必要になるまで式の評価を遅延させる

```
def If(x, y, z)
  if x
    y
  else
    z
  end
end
x = If(1>2, 3+4, 5+6) # 3+4は評価されない
```

Immutable::Promise

評価の遅延(delay)と強制(force)を明示

```
def If(x, y, z)
  if x.force
    y.force
  else
    z.force
  end
end
x = If(Promise.delay{1>2},
       Promise.delay{3+4}, Promise.delay{5+6})
```

lambda/callと何が違うの？

Memoization

何回forceしても一回しか評価されない

- 効率のため
- 評価する式に副作用がなければ結果は同じ

Promise.lazy

Promise.lazy { x }はPromise.delay { x.force }と同じ

再帰してもスタックが溢れない点が異なる

```
def loop
  Promise.lazy { loop }
  # Promise.delay { loop.force }だと
  # SystemStackErrorが発生
end
loop.force
```

SRFI-45参照

How to make lazy methods

コンストラクタをPromise.delayでくるむ

値を取り出すところではforceする

メソッド本体をPromise.lazyでくるむ

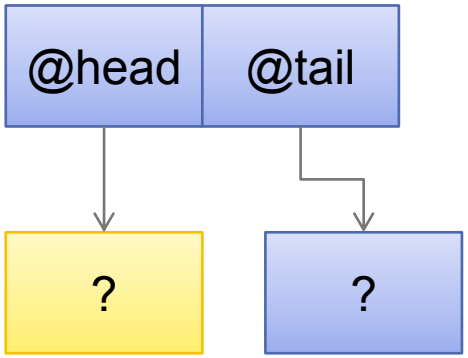
```
def stream_filter(s, &block)
  Promise.lazy {
    xs = s.force
    if xs.empty?
      Promise.delay { List[] }
    else
      ...
```

Immutable::Stream

遅延リスト

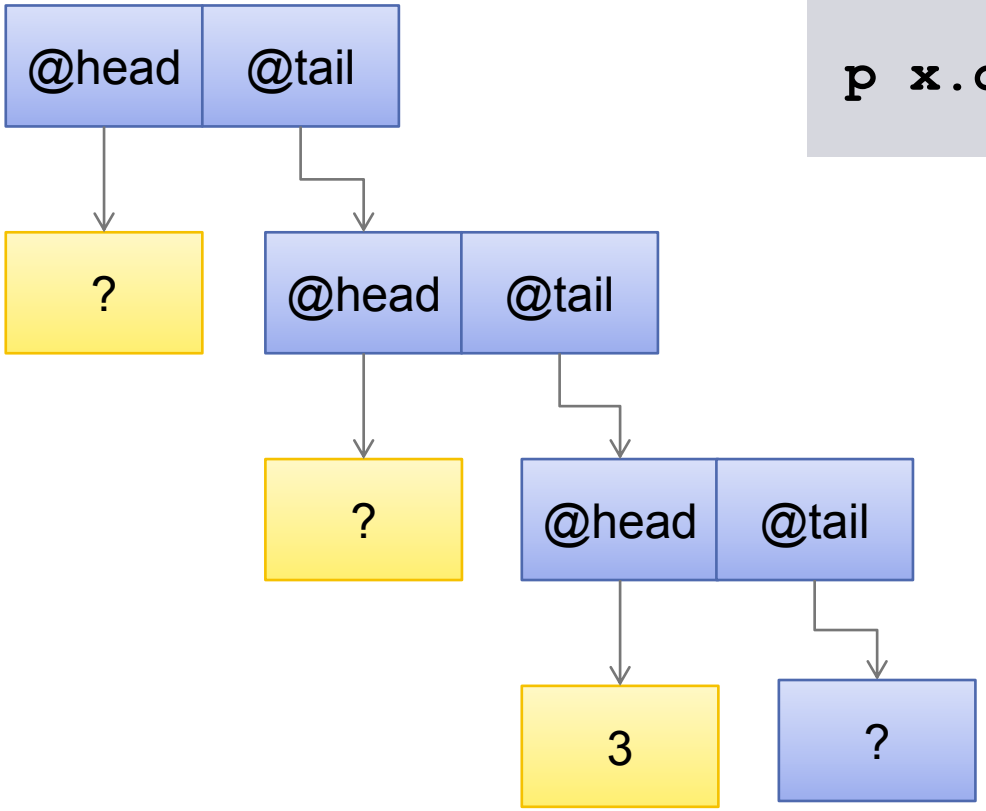
要素の値が必要になるまで要素の生成が遅延される

Lazy evaluation with streams




```
x = Stream.from(1)
```

Lazy evaluation with streams




```
x = Stream.from(1)
p x.drop(2).head #=> 2
```

Example from Project Euler

Problem 2

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

Answer

```
fib = Stream.cons(->{1},
        ->{Stream.cons(->{2},
             ->{fib.zip_with(fib.tail,
                             &:+)})})
p fib.filter(&:even?).take_while { |n|
  n <= 4_000_000
}.foldl(0, &:+)
```

Stream.cons(->{...}, ->{...})のように->{}が必要なのがちょっと汚い

- マクロがあれば…

Answer by unfolding

```
fib = Stream.unfoldr([1, 2]) { |x, y|
  [x, [y, x + y]]
}
p fib.filter(&:even?).take_while { |n|
  n <= 4_000_000
}.foldl(0, &:+)
```

Other data strutures

Immutable::Map

Immutable::Queue

Immutable::Deque

- See “Purely Functional Data Structures” by Chris Okasaki

Benchmark

```
SIZE = 10000
TIMES = 10
def run(bm, list, folding_method, msg)
  bm.report(msg) do
    TIMES.times do
      list.map {|i| i * 2}.send(folding_method, 0, &:+)
    end
  end
end
Benchmark.bmbm do |bm|
  run(bm, (1..SIZE).to_a, :inject, "Array")
  run(bm, Immutable::List[*(1..SIZE)], :foldl, "List")
  run(bm, Immutable::Stream.from(1).take(SIZE),
      :foldl, "Stream")
end
```

Benchmark result

```
Rehearsal ----------------------------------------
Array     0.080000   0.010000   0.090000 (  0.074561)
List      0.660000   0.000000   0.660000 (  0.665009)
Stream    5.340000   0.010000   5.350000 (  5.351024)
------------------------------- total: 6.100000sec

              user     system      total        real
Array     0.080000   0.000000   0.080000 (  0.079840)
List      0.590000   0.000000   0.590000 (  0.594458)
Stream    4.570000   0.000000   4.570000 (  4.583689)
```

ListはArrayの7.4倍、StreamはArrayの57.4倍遅い!

- mapなしでfoldlだけだともうちょっと早い

Conclusion

関数プログラミングはかっこいい

でも大抵はArrayやHashを使う方がおすすめ

- 繰り返しdupしてるようなコードはListやMapを使った方がいいかも

Slides

以下のURLで入手可能

- http://shugo.net/tmp/functional_ruby.pdf

Thank you!