Don't Drive on the Railroad Tracks

Eugene Wallingford
University of Northern Iowa
November 17, 2010

Two Claims

In the small, you know this.
It is no big deal.

In the large, this is different.
It changes how you think
about problems and data.

you know this

```
def addSalesTax( price )
    price * 1.07
end
```

```
def addSalesTax( price )
    price = price * 1.07
end
```

```
def addSalesTax( price )
    price = price * 1.07
end
```

```
def addSalesTax( price )
    tax   = price * 0.07
    price = price + tax
end
```

```
def addSalesTax( price )
    tax   = price * 0.07
    price = price + tax
end
```

side effects

side ~~effects~~


© PUBLICITY PICTURE


FESTIVAL

```
def addSalesTax( price )
    price * 1.07
end
```

```
sort -m access01-ips access02-ips  \
    | uniq -d                       \
    | wc -l
```

```
wc("-l",
   uniq("-d",
       sort("-m",
            access01-ips,
            access02-ips)))
```

[1 , 2 , 3 , 4]
["a","b","c","d"]

[1 , 2 , 3 , 4]

["a" , "b" , "c" , "d"]

```
[ 1 , 2 , 3 , 4 ].
  zip( ["a","b","c","d"] )
```

```
[[1, "a"],
 [2, "b"],
 [3, "c"],
 [4, "d"]]
```

```
[ 1 , 2 , 3 , 4 ]
{ |x|  x.odd? }
```

$$
\begin{array}{l}
[\ \boxed{1}\ ,\ 2\ ,\ \boxed{3}\ ,\ 4\ ] \\
[\ 1\ ,\ 3\ ]
\end{array}
$$

```
[ 1 , 2 , 3 , 4 ].
  select { |x| x.odd? }
```

[1, 3]

```
[ 1 , 2 , 3 , 4 ]
  { |x|  x.odd? }
```

[1 , 2 , 3 , 4]

[1 , 2 , 3 , 4]

```
[ 1 , 2 , 3 , 4 ].
  partition { |x| x.odd? }
```

```
[ [ 1, 3 ], [ 2, 4 ] ]
```

```
[ 1 , 2 , 3 , 4 ]
  { |x| x * x }
```

[1 , 2 , 3 , 4]

↓ ↓ ↓ ↓

[1 , 4 , 9 , 16]

```
[ 1 , 2 , 3 , 4 ].
  map { |x| x * x }
```

$$\begin{array}{ccccccccc}
[& 1 & , & 2 & , & 3 & , & 4 &] \\
 & \big\downarrow \text{^2} & & \big\downarrow \text{^2} & & \big\downarrow \text{^2} & & \big\downarrow \text{^2} & \\
[& 1 & , & 4 & , & 9 & , & 16 &]
\end{array}$$

```
[ 1 , 2 , 3 , 4 ]

   1 + 2 + 3 + 4 => 10
```

$$1 + 2 + 3 + 4$$

$$==$$

$$((1 + 2) + 3) + 4$$

```
[ 1 , 2 , 3 , 4 ].
  inject { |x,y| x + y }
```

```
[ 1 , 2 , 3 , 4 ].
  inject { |x,y| x + y }
```

fold the list with **+**

```
{ |x|   x.odd? }

{ |x|   x * x  }

{ |x,y| x + y }
```

functions

are

first-class values

```
# Python
for item in iterable_collection:
  # do something with item


# Ruby
set.each do |item|
  # do something with item
end
```

next steps

implies

recursion

over

persistent
data structures

```
number ::= 0
         | 1 + number
```

```
list ::= empty
       | item + list
```

```
tree ::= empty
       | item + tree + tree
```

induction

implies

recursion

what

versus

how

```
number ::= 0
         | 1 + number
```

```
if n = 0
   do something
else
   solve for 1
   solve for n-1
   combine
```

```
number ::= 0
         | 1 + number
```

```
number ::= 0
         | 1 + number
```

decrease and conquer

```
number ::= 0
         | 1 + number
```

sequential

```
number ::= 0
         | number/2
              +
           number/2
```

```
number ::= 0
         | number/2
               +
           number/2
```

divide and conquer

```
number ::= 0
         | number/2
               +
           number/2
```

parallel

```
tree ::= empty
       | item + tree + tree
```

divide and conquer

parallel

MapReduce

map an operator
over each item

reduce (fold)
the resulting list

[8, 4, 1, 6, 7, 2, 5, 3]

[1, 2, 3, 4, 5, 6, 7, 8]

```
[ 8, 4, 1, 6, 7, 2, 5, 3 ].
  map { |x| [x] }
```

make a list of each item

```
[[8], [4], [1], [6],
 [7], [2], [5], [3]]
```

```
[[8], [4], [1], [6],
 [7], [2], [5], [3]].
    inject { |x,y| merge(x,y) }
```

merge the sorted lists, pairwise

```
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

```
[ 8, 4, 1, 6, 7, 2, 5, 3 ]
  .map    { |x| [x] }
  .inject { |x,y| merge(x,y) }
```

$\downarrow$ *map/reduce*

```
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

Implications for Parallelism

```
merge(a,b) == merge(b,a)

&&

merge(a, merge(b,c))
==
merge(merge(a, b),c)
```

merges can be done ***independently***


FESTIVAL BANQU

really getting it

```ruby
class Proc
  def self.compose(f, g)
    lambda { |*args| f[g[*args]] }
  end
end
```

```ruby
class Proc
  def self.compose(f, g)
    lambda { |*args| f[g[*args]] }
  end
end
```

```ruby
class Proc
  def self.compose(f, g)
    lambda { |*args| f[g[*args]] }
  end
end
```

```ruby
class Proc
  def self.compose(f, g)
    lambda { |*args| f[g[*args]] }
  end
end
```

combinator

A **combinator** is a function
that takes functions as input
and computes its result
by composing those functions. *

** and nothing else.*
There are no free variables.

combinator is to functional programming

as

framework is to object-oriented programming

combinator is to functional programming

as

framework is to object-oriented programming

the next level of abstraction

A Common Pattern...

```
widget.collection
 .select { |a_table|
    a_table.widgets_column_name =~ regex }
 .map    { |a_table|
    widget.attribute_present?(a_table.widgets_column_name) &&
    { a_table.label
           => widget.send(a_table.widgets_column_name) }
      || {} }
 .inject(&:merge)
```

Combinators in Action

suppose we want to find
the square of the sum
of all the odd numbers
between 1 and 100

```
(1..100)
```

```
(1..100).select(&:odd?)
```

```ruby
(1..100).select(&:odd?).inject(&:+)
```

```
lambda { |x| x * x }.call(
  (1..100).select(&:odd?).inject(&:+))
```

```
lambda { |x| x * x }.call(
  (1..100).select(&:odd?).inject(&:+))
```

A **permuting combinator**
composes two functions
in reverse order.

Instead of f(g(x), we want g(f(x)).

```
(1..100).select(&:odd?).inject(&:+)
  .callWithSelf(lambda { |x| x * x })
```

```
(1..100).select(&:odd?).inject(&:+)
  .into         (lambda { |x| x * x })
```

```
(1..100)
  .select(&:odd?)
  .inject(&:+)
  .into(lambda { |x| x * x })
```

```ruby
class Object
  def into expr = nil
    expr.nil? ? yield(self) : expr.to_proc.call(self)
  end
end
```

Um, what about Scala?

```scala
case class Thrush[A](x: A) {
  def into[B](g: A => B): B = {
    g(x)
  }
}
```

```
Thrush((1 to 100)
  .filter(_ % 2 != 0)
  .foldLeft(0)(_ + _))
  .into((x: Int) => x * x)
```

```
accounts
   .filter(_ belongsTo "John S.")
   .map(_.calculateInterest)
   .filter(_ > threshold)
   .foldLeft(0)(_ + _)
   .into {x: Int =>
      updateBooks journalize
                  (Ledger.INTEREST, x)
    }
```

more?

functional design patterns

Structural Recursion

functional design patterns

Structural Recursion
Interface Procedure

functional design patterns

Structural Recursion
Interface Procedure
Mutual Recursion

functional design patterns

Structural Recursion
Interface Procedure
Mutual Recursion
Accumulator Passing

functional design patterns

Structural Recursion
Interface Procedure
Mutual Recursion
Accumulator Passing
Local Procedure

functional design patterns

Structural Recursion
Interface Procedure
Mutual Recursion
Accumulator Passing
Local Procedure
Program Derivation

functional design patterns

Structural Recursion
Interface Procedure
Mutual Recursion
Accumulator Passing
Local Procedure
Program Derivation

Tail-Recursive State Machine
Continuation Passing
Control Abstraction

Isn’t all this recursion so inefficient as to be impractical?

This is the 21st century.

garbage collection

tail-call elimination

```
def foo(...) = {
  if (n is base case)
     return some value
  else
     foo(...)
}
```

<Scheme indulgence>

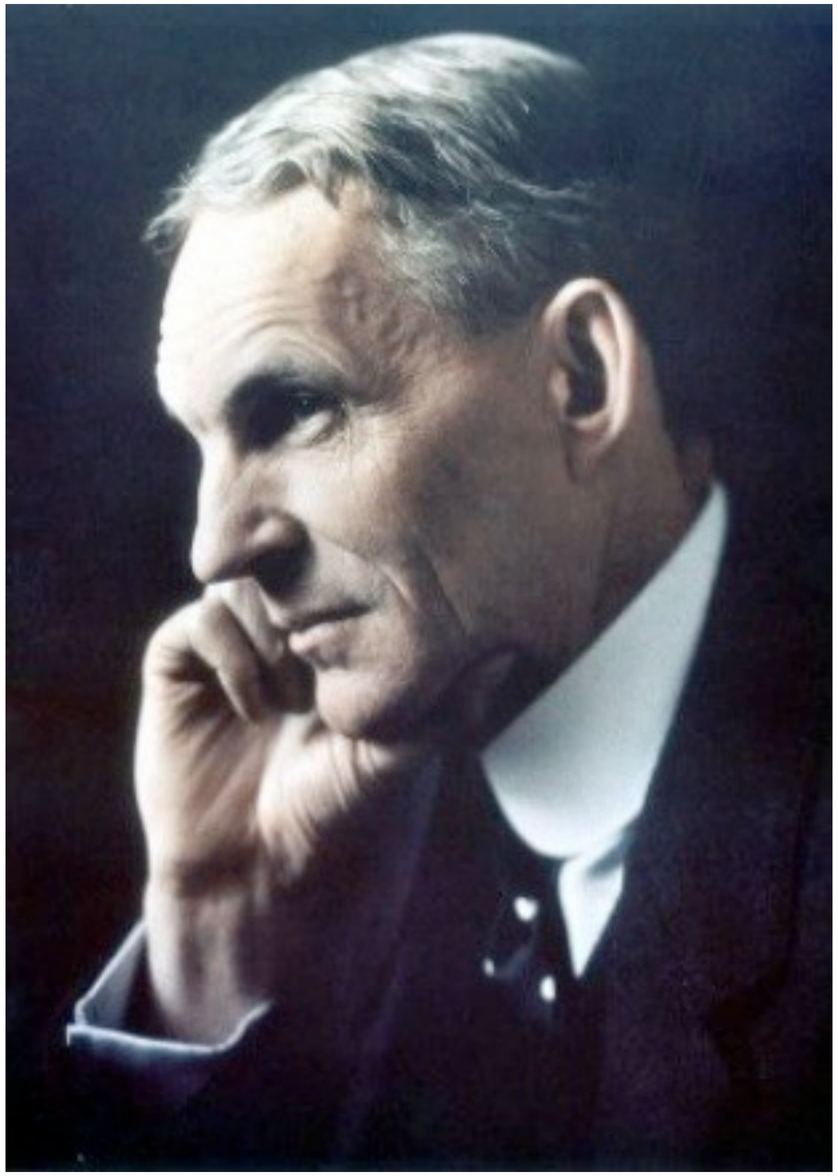
```
def factorial(n: Int) = {
 def loop(n: Int, acc: Int): Int =
   if (n <= 0)
      acc
   else
      loop(n - 1, acc * n)

 loop(n, 1)
}
```

return 'done

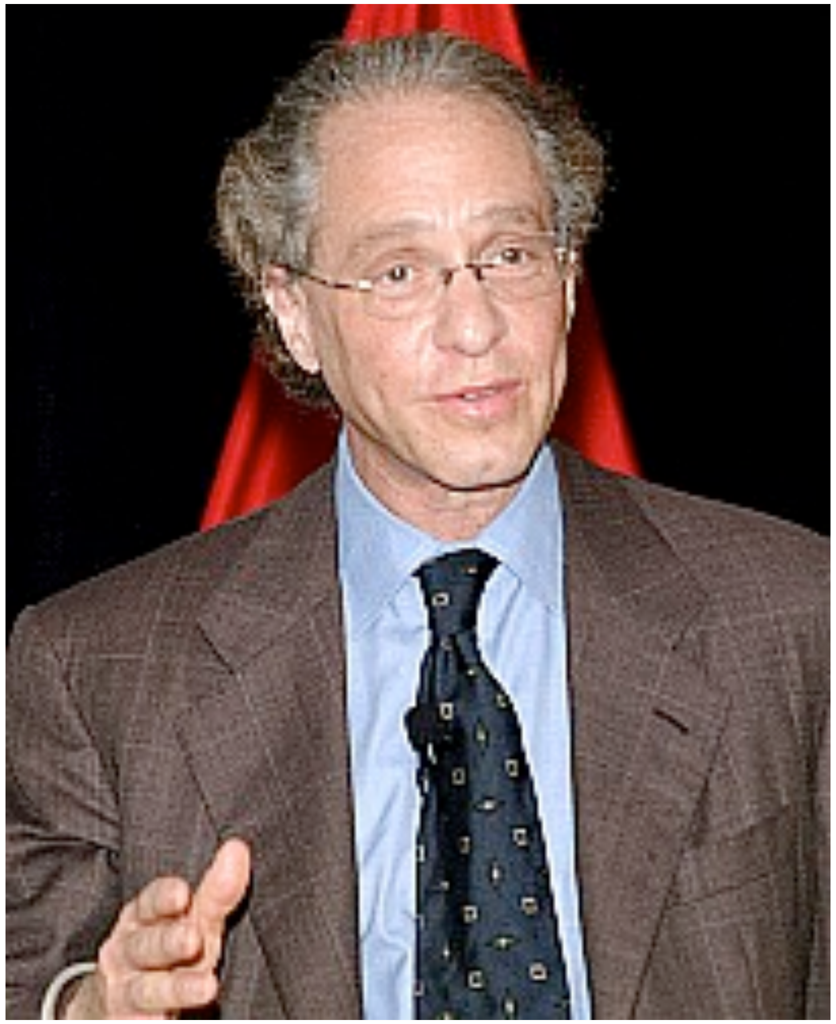
If I had asked people
what they wanted,
they would have said
'faster horses'.

Henry Ford



An invention has to make
sense in the world
in which it is finished,
not the world
in which it was started.

Ray Kurzweil



resources to study

http://www.youtube.com/watch?v=c_5GpBgsang

http://weblog.raganwald.com/2008/01/no-detail-too-small.html

http://debasishg.blogspot.com/2009/09/thrush-combinator-in-scala.html

http://fupeg.blogspot.com/2009/04/tail-recursion-in-scala.html

http://www.cs.uni.edu/~wallingf/patterns/recursion.html

http://www.cs.uni.edu/~wallingf/patterns/envoy.pdf

http://mitpress.mit.edu/sicp/

http://sicpinclojure.com/