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
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 ]

{ \vert x \vert \quad x.\text{odd}? }
\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
1 & 3 & 3 &
\end{bmatrix}
\]
\[ \{ 1, 2, 3, 4 \}. \]

\[
\text{select } \{ |x| \ x\text{.odd?} \}
\]
[ 1, 3 ]
\[ [1, 2, 3, 4] \]
\[
\{ |x| \quad x \text{.odd?} \}
\]
\[ [1, 2, 3, 4]. \]
partition \{ |x| \ x.odd? \}
\[
[ [ 1, 3 ], [ 2, 4 ] ]
\]
\[ \{ 1, 2, 3, 4 \} \]
\[ \{ |x| \ x \ast x \} \]
\[ [1, 2, 3, 4] \]
\[ \downarrow \downarrow \downarrow \downarrow \]
\[ [1, 4, 9, 16] \]
[1, 2, 3, 4].
map { |x| x * x }
\[
\begin{array}{c}
[1, 2, 3, 4] \\
\downarrow^2 \quad \downarrow^2 \quad \downarrow^2 \quad \downarrow^2 \\
[1, 4, 9, 16]
\end{array}
\]
[ 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 }
\[
[1, 2, 3, 4].
\text{inject } \{ |x,y| x + y \}
\]

\textbf{fold} the list with \texttt{+}
\{ |x| \quad x.\text{odd?} \ \}\}

\{ |x| \quad x \ast x \ \}\}

\{ |x,y| \quad 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] }`

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

`make a list of each item`
in inject \{ |x,y| \text{merge}(x,y) \}

\[ \text{merge the sorted lists, pairwise} \]

\[ [1, 2, 3, 4, 5, 6, 7, 8] \]
\[
\begin{array}{c}
[8, 4, 1, 6, 7, 2, 5, 3] \\
\map \{ |x| [x] \} \\
\inj \{ |x, y| \text{merge}(x, y) \}
\end{array}
\]

\[
\downarrow \quad \text{map/reduce}
\]

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

Thursday, November 18, 2010
Implications for Parallelism

\[
\text{merge}(a,b) == \text{merge}(b,a) \\
\&\& \\
\text{merge}(a, \text{merge}(b,c)) \\
== \\
\text{merge}(\text{merge}(a, b), c)
\]

*merges can be done independently*
really getting it
class Proc
  def self.compose(f, g)
    lambda { |*args| f[g[*args]] }
  end
end

Thursday, November 18, 2010
class Proc
  def self.compose(f, g)
    lambda { |*args| f[g[*args]] }
  end
end
class Proc
  def self.compose(f, g)
    lambda { |*args| f[g[*args]] }
  end
end
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...

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

Um, what about 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>
```scala
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://www.cs.uni.edu/~wallingf/patterns/recursion.html


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

http://sicpticlojure.com/