Why Functional Programming Matters

An Introduction

Andreas Pauley

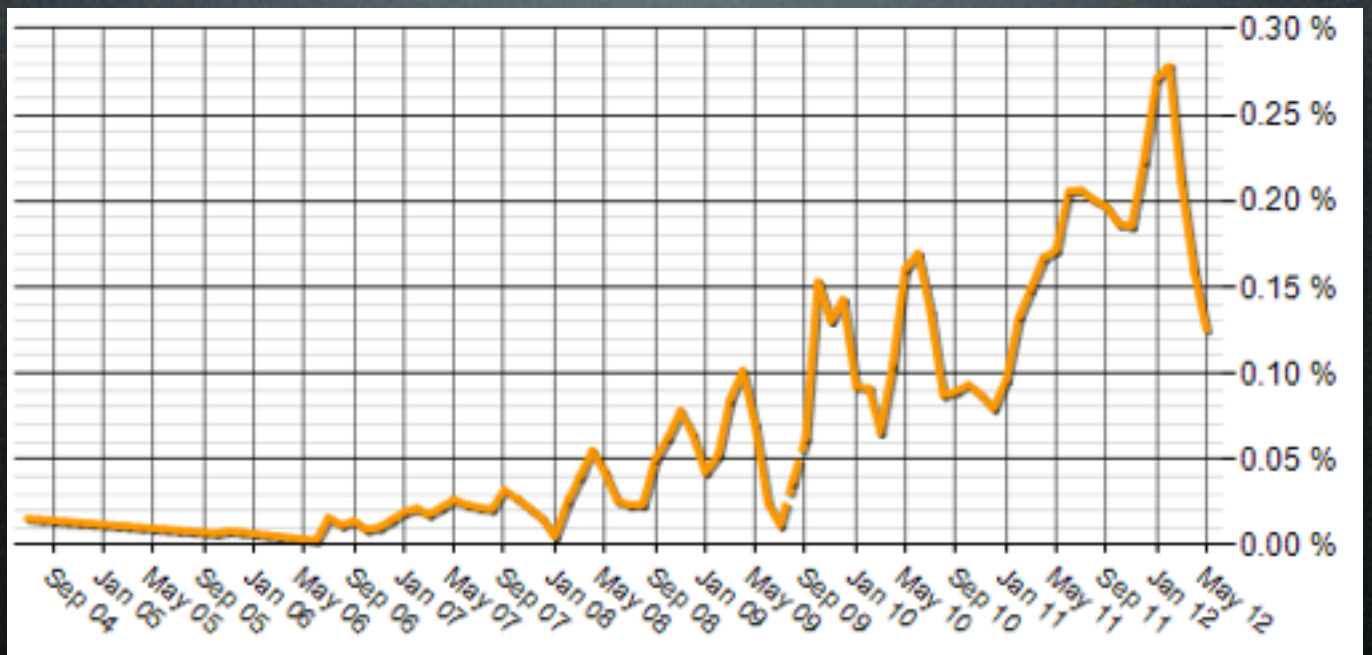
Lambda Luminaries

http://www.meetup.com/lambda-luminaries/

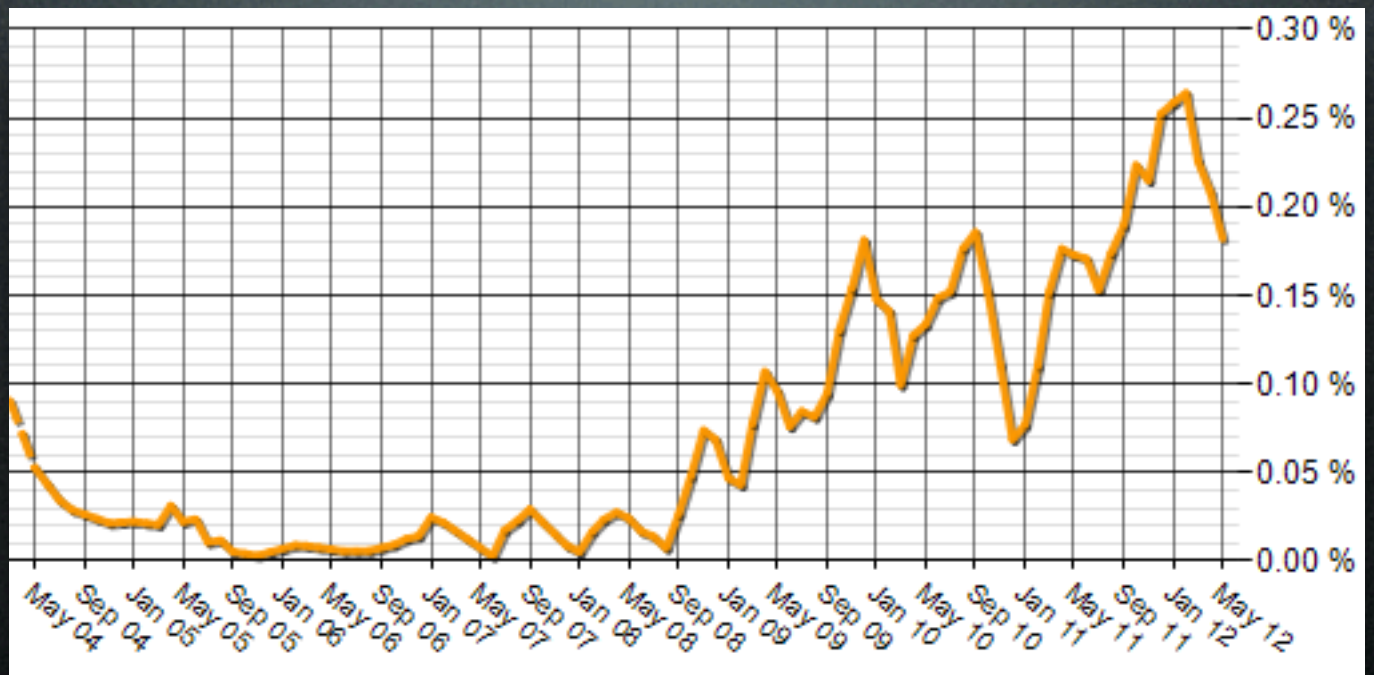
“No matter what language you work in, programming in a functional style provides benefits.
You should do it whenever it is convenient, and you should think hard about the decision when it isn’t convenient.”
John Carmack (ID Software)

http://www.altdevblogaday.com/2012/04/26/functional-programming-in-c/

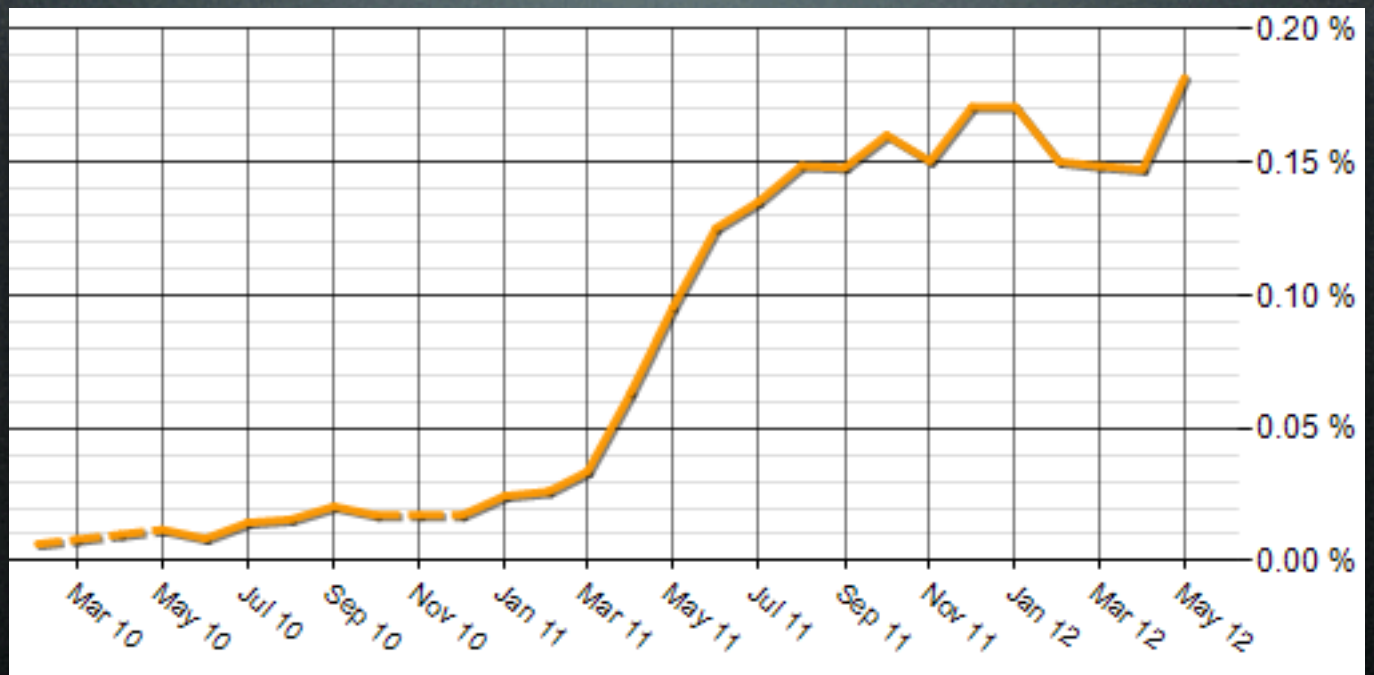
Interest in FP is Growing



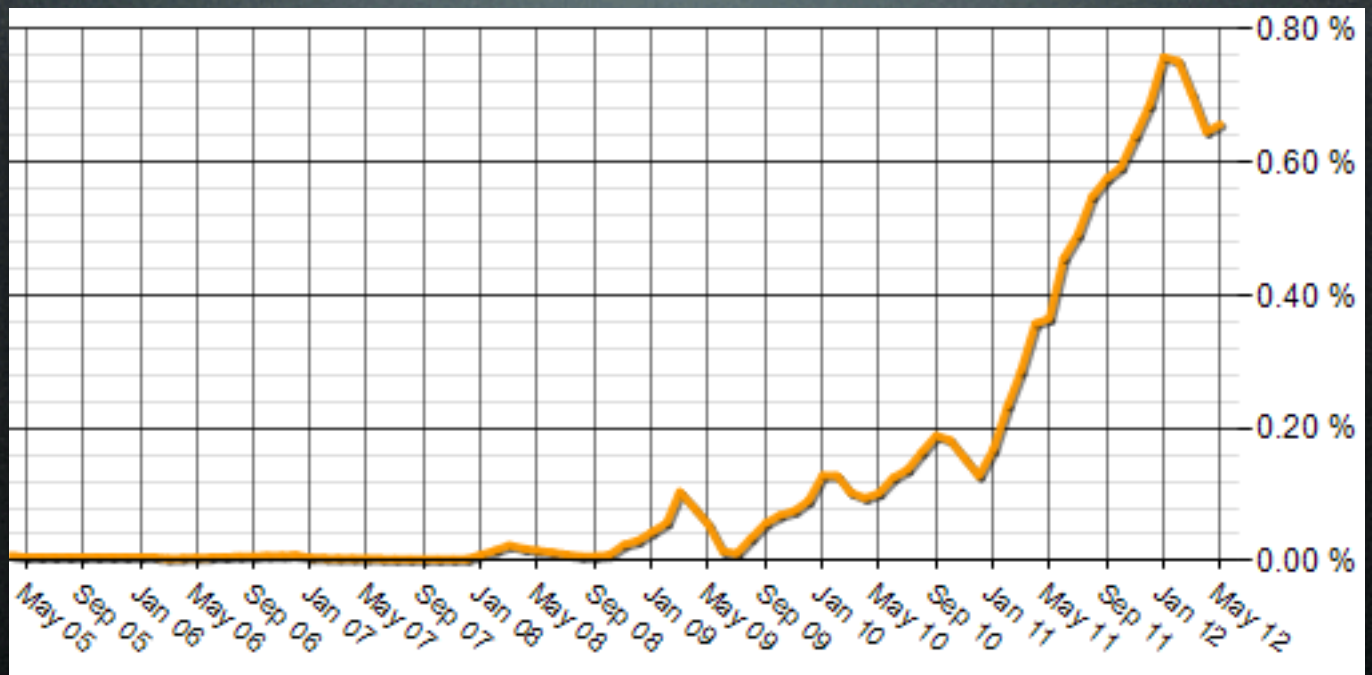
Haskell Demand Trend



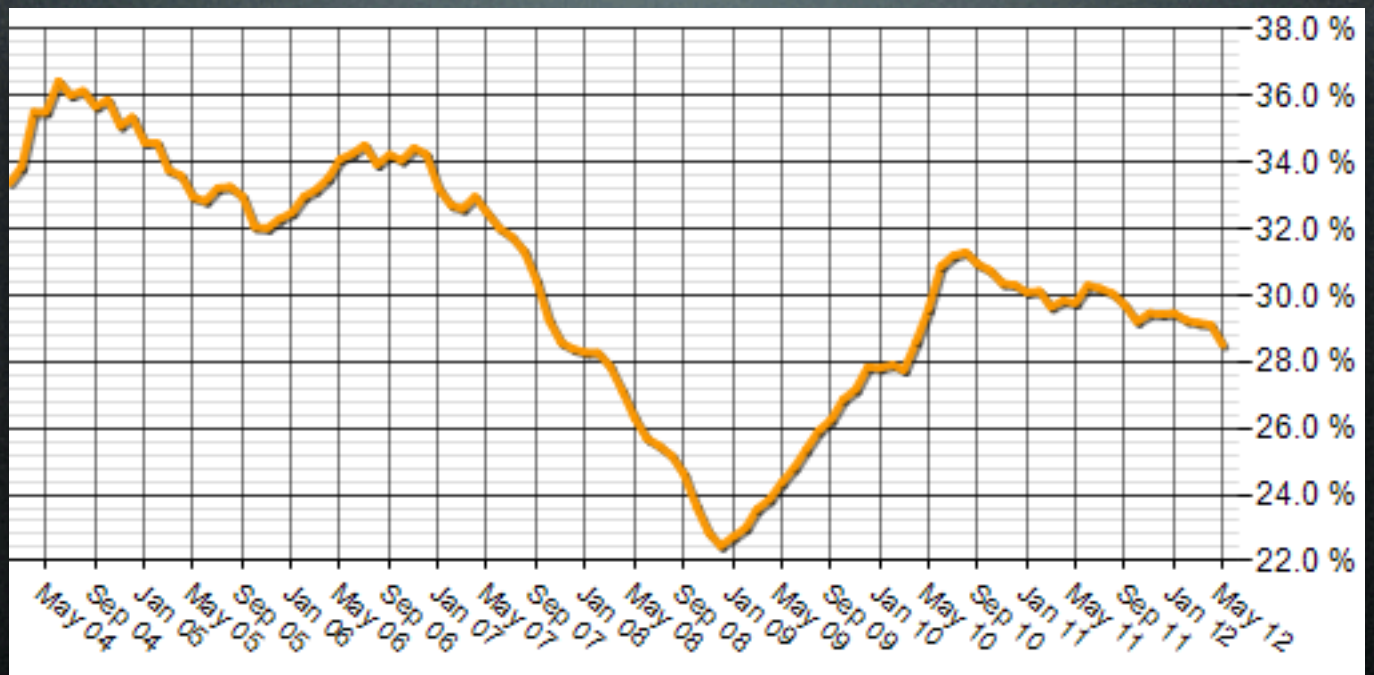
Erlang Demand Trend



Clojure Demand Trend



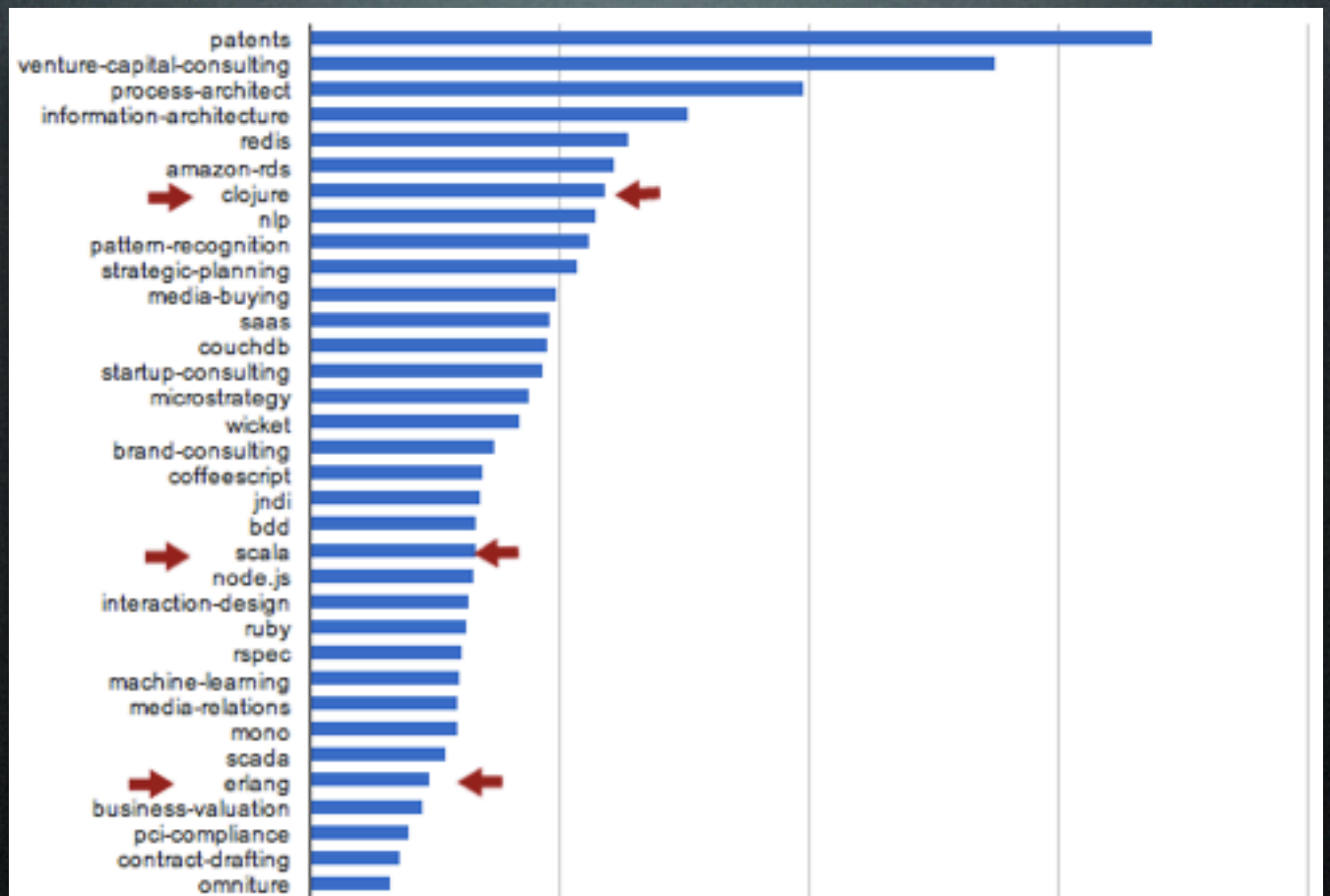
Scala Demand Trend



Java Demand Trend

http://www.itjobswatch.co.uk/

Charts providing 3-month moving totals of permanent jobs in the UK citing a programming language as a proportion of the total demand within the Programming Languages category



High Wage Skills

Some interesting companies are using Functional Programming

- Twitter - Scala, Clojure
- Prismatic - Clojure
- Google - MapReduce
- CouchDB - Erlang
- Riak - Erlang
- More?

So what exactly is Functional Programming?

Functional Programming is a list of things you CAN'T do

You can't vary your variables

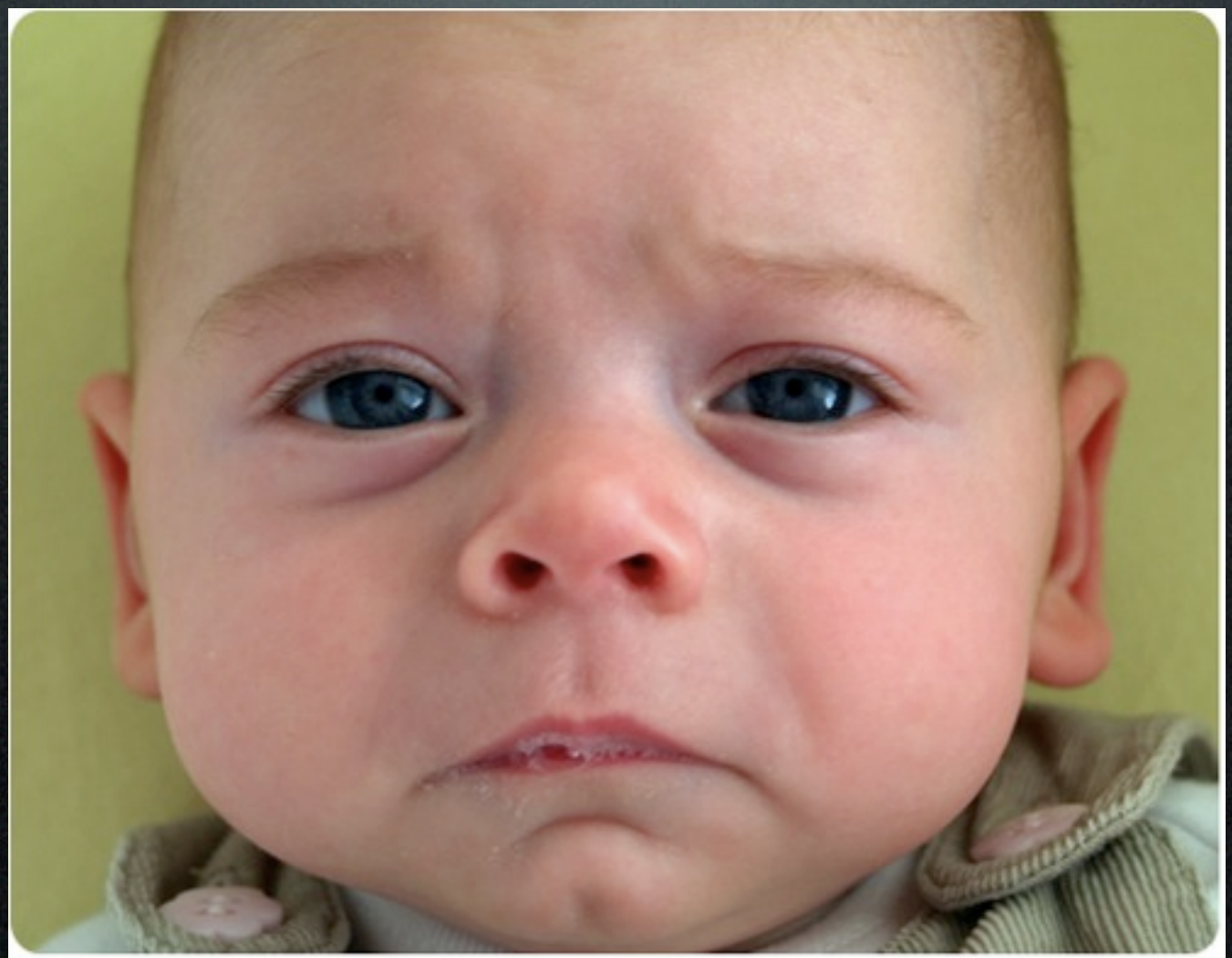
You can't mutate or change your state

No while/for loops,
sorry

You can't have side-effects

You can't control the order of execution

How can anyone program like this?



This is like
“you can’t have
GOTO’s”

We need a better definition

What about:

"Functional programming is so called because a program consists entirely of functions."

John Hughes in Why Functional Programming Matters

OK... So what exactly is a function?

$$f(x) = 2x + 3$$

$$f(x) = 2x + 3$$

The value of x will not change inside the function body.
No x = 1; x = 2

$$f(x) = 2x + 3$$

Same input, same output. Every time. (Referential Transparency)

$$g(x) = f(x) + 1$$

Functions can call other functions

k = 42

Constant values are just functions with no input parameters

$$h(x) = f(g(x))$$

Functions can be combined

$$h(f, g, x) = f(x) + g(2)$$

Functions can take functions as input and/or return functions as the result

A functional program consists entirely of functions

```
main(args) =
  result = call_some_function(args)
  call_another_function(result)
```

Object-Oriented vs Functional Programming

“OO makes code understandable by encapsulating moving parts.
FP makes code understandable by minimizing moving parts.”
Michael Feathers @mfeathers

Overlapping OO and FP principles

Overlapping OO and FP principles

- DRY (don't repeat yourself)
- Separation of Concerns
- Do the simplest thing that can possibly work
- High Cohesion, Low Coupling
- More?

High Cohesion, Low Coupling

High Cohesion: Related concepts are grouped together
Low Coupling: changes in one part of a program does not
affect other parts.

What happens if we take low coupling to the extreme?

Low Coupling in FP

The state (data) in a program is decoupled from the behaviour.

Some Common Functional Language Features

Recursion

```
doubleAll :: Num a => [a] -> [a]
doubleAll []     = []
doubleAll (x:xs) = x*2 : doubleAll xs
```

Iterative While Loop

```
public static ArrayList<Integer> double_each(ArrayList<Integer> in) {
  ArrayList<Integer> out = new ArrayList<Integer>();
  for(int i : in) {
    out.add(i*2);
  }
  return out;
}
```

Pattern Matching

```
doubleAll :: Num a => [a] -> [a]
doubleAll []     = []
doubleAll (x:xs) = x*2 : doubleAll xs
```

Higher-order Functions

```
def f(x):
  return x + 3

def g(function, x):
  return function(x) * function(x)

print g(f, 7)
```

List Comprehensions

```
Prelude> let s = "Hello, world!"
Prelude> [toUpper c | c <- s]
"HELLO, WORLD!"
Prelude> [x*2 | x <- [1..5]]
[2,4,6,8,10]
```

Advantages of Functional Programming

FP provides some higher levels of abstraction. Example: working with lists

Basic List Operations

- Transform each element of a list into something else, returning a new list of the same size
- Getting the subset of a list that matches some condition
- Going through a list and accumulating all values into a single value

New list from old in Java

```
import java.util.*;

public class Transmogrify {
    public static void main (String[] args) {
        ArrayList<Integer> input = range(1, 10);
        ArrayList<Integer> doubled = double_each(input);
    }
    public static ArrayList<Integer> double_each(ArrayList<Integer> in) {
        ArrayList<Integer> out = new ArrayList<Integer>();
        for(int i : in) {
            out.add(i*2);
        }
        return out;
    }
    public static ArrayList<Integer> range(int start, int end) {
        ArrayList<Integer> k = new ArrayList<Integer>();

        for (int i=start; i<end; i++) {
            k.add(i);
        }
        return k;
    }
}
```

New list from old in Java

```
public static ArrayList<Integer> double_each(ArrayList<Integer> in) {
  ArrayList<Integer> out = new ArrayList<Integer>();
  for(int i : in) {
    out.add(i*2);
  }
  return out;
}
```

Map in Erlang

```
doubleAll(List) ->
  DoubleFun = fun(X) -> X * 2 end,
  lists:map(DoubleFun, List).
```

Map in Haskell

```
doubleAll somelist = map (*2) somelist
```

Map in Haskell

```
doubleAll = map (*2)
```

```
Prelude> doubleAll [2,3,4]
[4,6,8]
```

For Comparison

```
doubleAll :: Num a => [a] -> [a]
doubleAll []     = []
doubleAll (x:xs) = x*2 : doubleAll xs
```

Filter a list in Haskell

```
passed = filter (>60)
```

```
Prelude> passed [49, 58, 76, 82, 88, 90]
[76,82,88,90]
```

Filter a list in Haskell

```
passfail = partition (>60)
```

```
Prelude> let (passed, failed) = passfail [49, 58, 76, 82, 88, 90]
Prelude Data.List> passed
[76,82,88,90]
Prelude Data.List> failed
[49,58]
```

Reduce a list to a single element

```
Prelude> foldl (+) 0 [1,2,3]
6

Prelude> foldl (*) 1 [2,3,4]
24
```

Reduce a list to a single element

```
Prelude> foldl (+) 0 [1,2,3]
6
Prelude> (+) 0 1
1
Prelude> foldl (+) 1 [2,3]
6
Prelude> (+) 1 2
3
Prelude> foldl (+) 3 [3]
6
Prelude> foldl (+) 6 []
6
```

Reduce a list to a single element

```
Prelude> foldl (*) 1 [2,3,4]
24
Prelude> foldl (*) 2 [3,4]
24
Prelude> foldl (*) 6 [4]
24
Prelude> foldl (*) 24 []
24
```

Smalltalk and Ruby

map ⇔ collect

filter ⇔ select

fold ⇔ reduce

Advantages

- Abstractions with higher-order functions

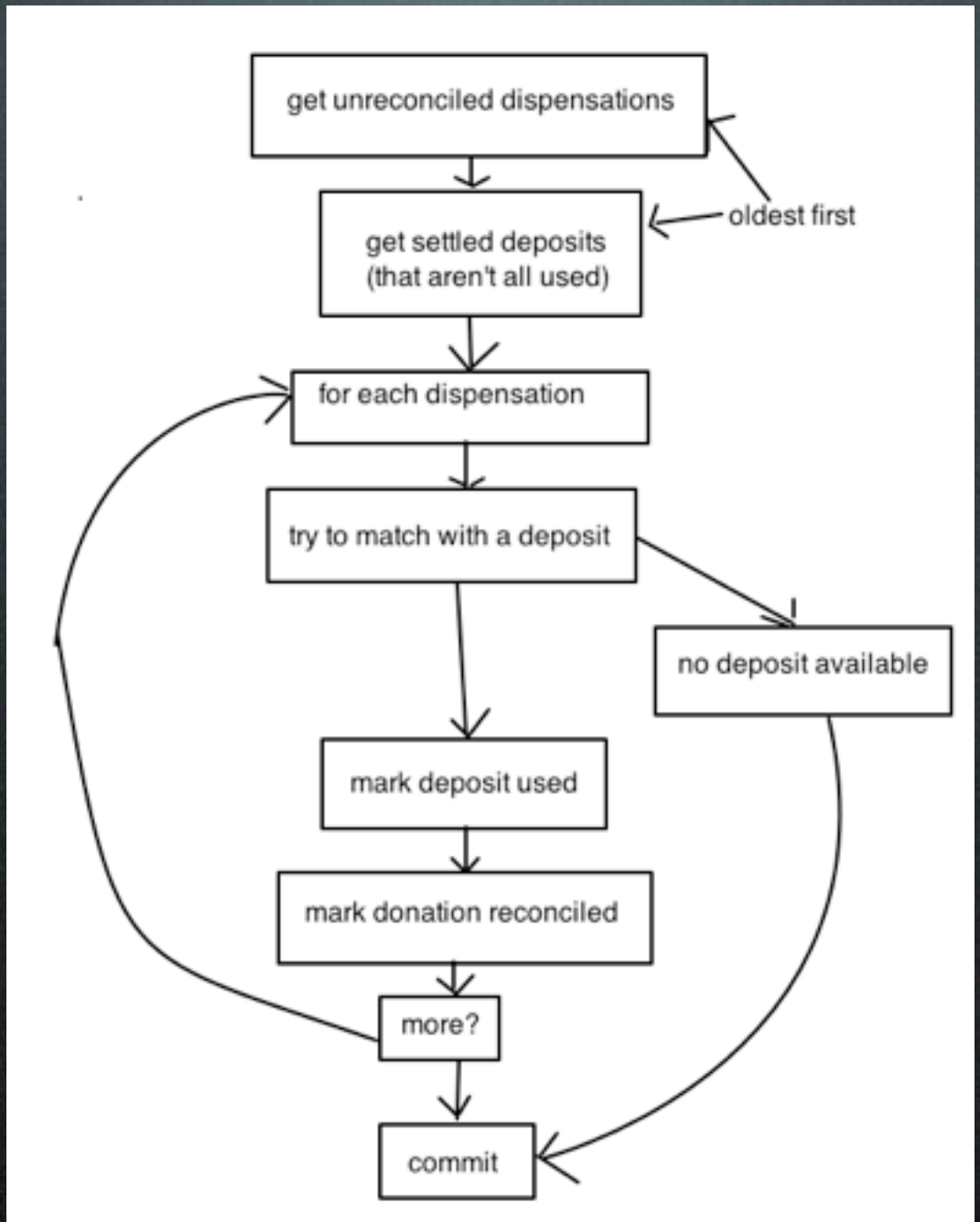
FP will change the way you approach problem solving

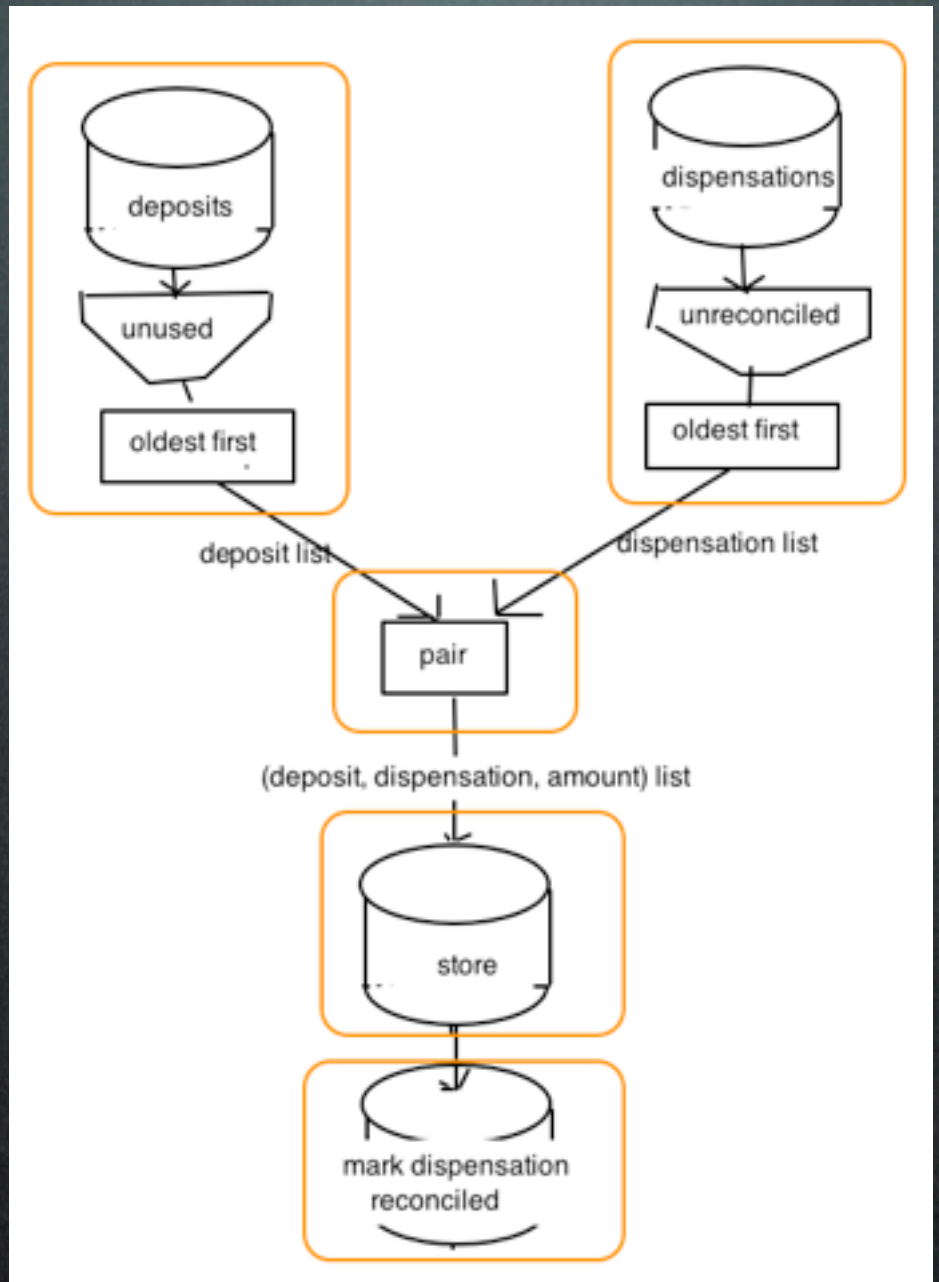
“Functional programming encourages one to think about processing in terms of data streams”

JessiTRON

http://jessitron.blogspot.com/2012/06/why-functional-matters-your-white-board.html?m=1

The problem:
Money comes in through deposits. Money goes out through dispensations. Take an ordered list of each and pair them up into Utilizations, which identify which deposits were used to satisfy which dispensations. If any dispensation cannot be satisfied with the provided deposits, do not satisfy the dispensation at all.


get unreconciled dispensations
oldest first
get settled deposits
(that aren't all used)
for each dispensation
try to match with a deposit
no deposit available
mark deposit used
mark donation reconciled
more?
commit


deposits
unused
oldest first
dispensations
unreconciled
oldest first
deposit list
dispensation list
pair
(deposit, dispensation, amount) list
store
mark dispensation
reconciled

Advantages

- Abstractions with higher-order functions
- Changes* your brain (*may include frying)

FP makes concurrency easier

Lock-free concurrency

FP makes reasoning about code easier

Pure functions. Explicit state

FP makes testing code easier

QuickCheck

No Null.
No NullReferenceException

Advantages

- Abstractions with higher-order functions
- Changes* your brain (*may include frying)
- Concurrency
- Reasoning about Code
- Testing
- No Null

Disadvantages of Functional Programming

- Steep learning curve
- Some concepts are very cryptic and abstract
- Tools/IDE's are not as advanced yet
- Order of execution may be difficult to reason about in a lazy language
- More?

Functional Programming is unfamiliar territory for most

“If you want everything to be familiar you will never learn anything new”
Rich Hickey

Now What?

Functional Programming

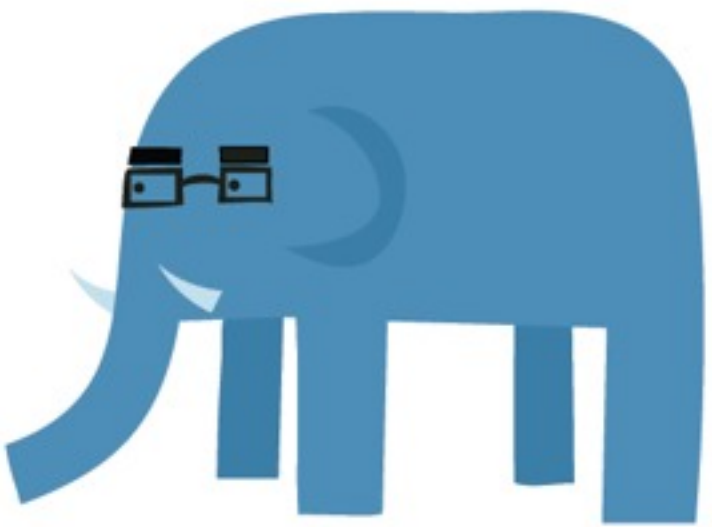
for Java Developers

O'REILLY®

Dean Wampler

Learn You a Haskell for Great Good!

A Beginner's Guide



Miran Lipovača

no starch press

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Joe Armstrong


patternmatched
TECHNOLOGIES

Lambda Luminaries

http://www.meetup.com/lambda-luminaries/

A Small Programming Example: Hollingberries

https://github.com/apauley/HollingBerries

@lambdaluminary
@apauley