

Functional Programming In Practice

Michiel Borkent

[@borkdude](https://twitter.com/borkdude)

Hogeschool Utrecht

May 9th 2016



simacan

Agenda

- Functional Programming
- Haskell, Scala, Clojure
- Full stack example using Clojure

First, what is FP?

- First class functions
- Pure functions
- Immutable values
- No or few side effects

First class functions

Functions can be created on the fly

```
scala> val f = (x: String) => "Hello " + x  
f: String => String = <function1>
```

```
scala> f("World")  
res10: String = Hello World
```

First class functions

Functions can be passed around as values

```
List("Clojure", "Scala", "Haskell").map(f)  
res: List>Hello Clojure, Hello Scala, Hello Haskell)
```

Pure functions

Function with same input always yields the output:

$f(x) == y$, always

Not a pure function

```
scala> val f = (i: Int) => Math.random() * i  
f: Int => Double = <function1>
```

```
scala> f(1)  
res14: Double = 0.13536266885499726
```

```
scala> f(1)  
res15: Double = 0.4086671423543593
```

A pure function?

```
val add = (x: Int, y: Int) => x + y  
add(1,2) // 3
```

A pure function?

```
class MutableInt(var i: Int) {  
    override def toString = i.toString  
}  
  
val add = (x: MutableInt, y: MutableInt): MutableInt =>  
    new MutableInt(x.i + y.i)  
  
val x = new MutableInt(1)  
val y = new MutableInt(2)  
  
add(x,y) // MutableInt = 3
```

A pure function? No!

You cannot build pure functions with mutable objects.

```
add(x,y) // MutableInt = 3
```

```
x.i = 2
```

```
add(x,y) // MutableInt = 4
```

`add(x,y)` does not always yield the same result!

This is why we need **immutable values** in Functional Programming.

A pure function?

List(1,2,3) is immutable.

```
def addZero(l: List[Int]) = 0 :: l
```

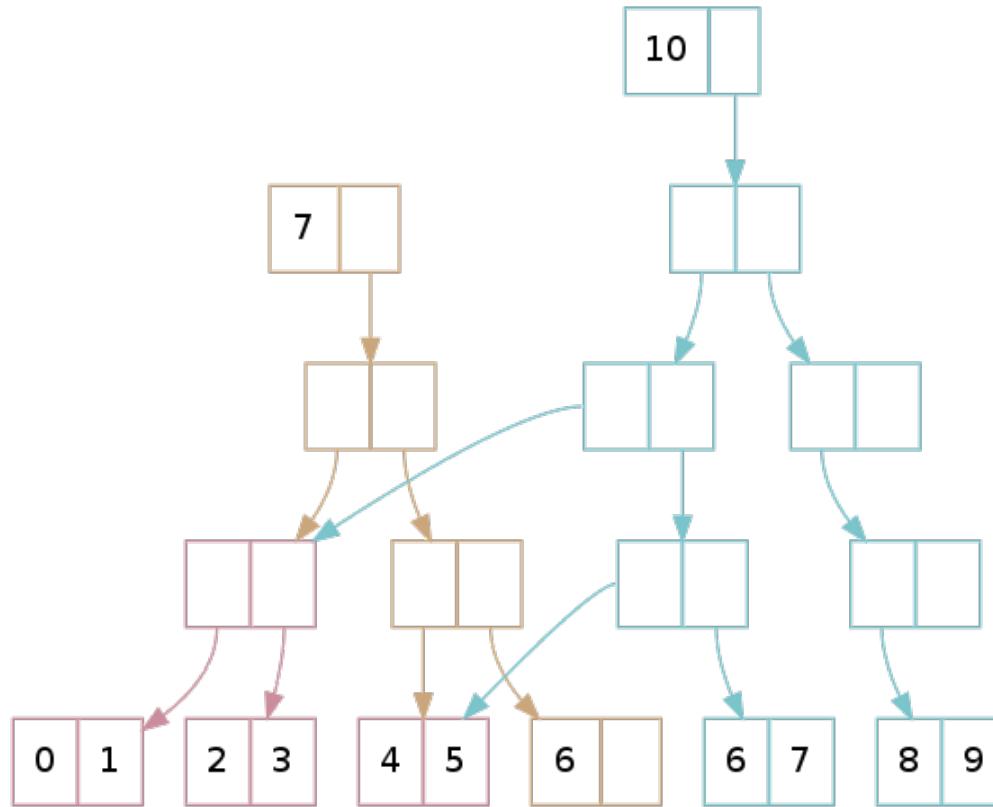
```
addZero(List(1,2,3)) // List(0, 1, 2, 3)
```

Immutable data structures

`Int`, `String`, etc are already immutable

```
0 :: List(1,2,3)    // List(0,1,2,3)  
Vector(1,2,3) :+ 4 // Vector(1,2,3,4)  
Set(1,2,3) + 4    // Set(1,2,3,4)
```

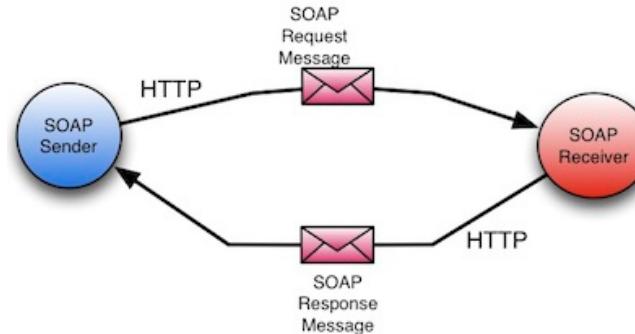
Efficient by re-using structure internally



source: <http://hypirion.com/musings/understanding-persistent-vector-pt-1>

Systems and immutability

- Each system receives a message and/or sends a message
- Mutating a message does not affect other system
- In traditional OO references lead to uncontrolled mutation, also called [spooky action at a distance](#)
- You can protect yourself by using Value Objects or DTOs, but takes work
- Immutable data structures solve this problem



Side effects

- State modification
- Observable action

Examples:

- Modifying a variable
- Writing to a file

Side effects

- Pure functions have no side effects
- Side effects are difficult to test (without mocks)
- Pure FP languages make side effects explicit
- FP languages isolate/minimize side effects

Where are the side effects?

```
class Program extends App {  
    var x: Int = 1  
    def mutateX = {  
        x = (Math.random * 100).toInt  
    }  
    mutateX  
    println(x)  
}
```

Where are the side effects?

```
class Program extends App {  
    var x: Int = 1  
    def mutateX = {  
        x = (Math.random * 100).toInt  
    }  
    mutateX  
    println(x)  
}
```

Why Functional Programming?

- Makes codes easier to reason about
- Easier to test
- More expressive, less code = less bugs
- Better suited for parallelisation and concurrency

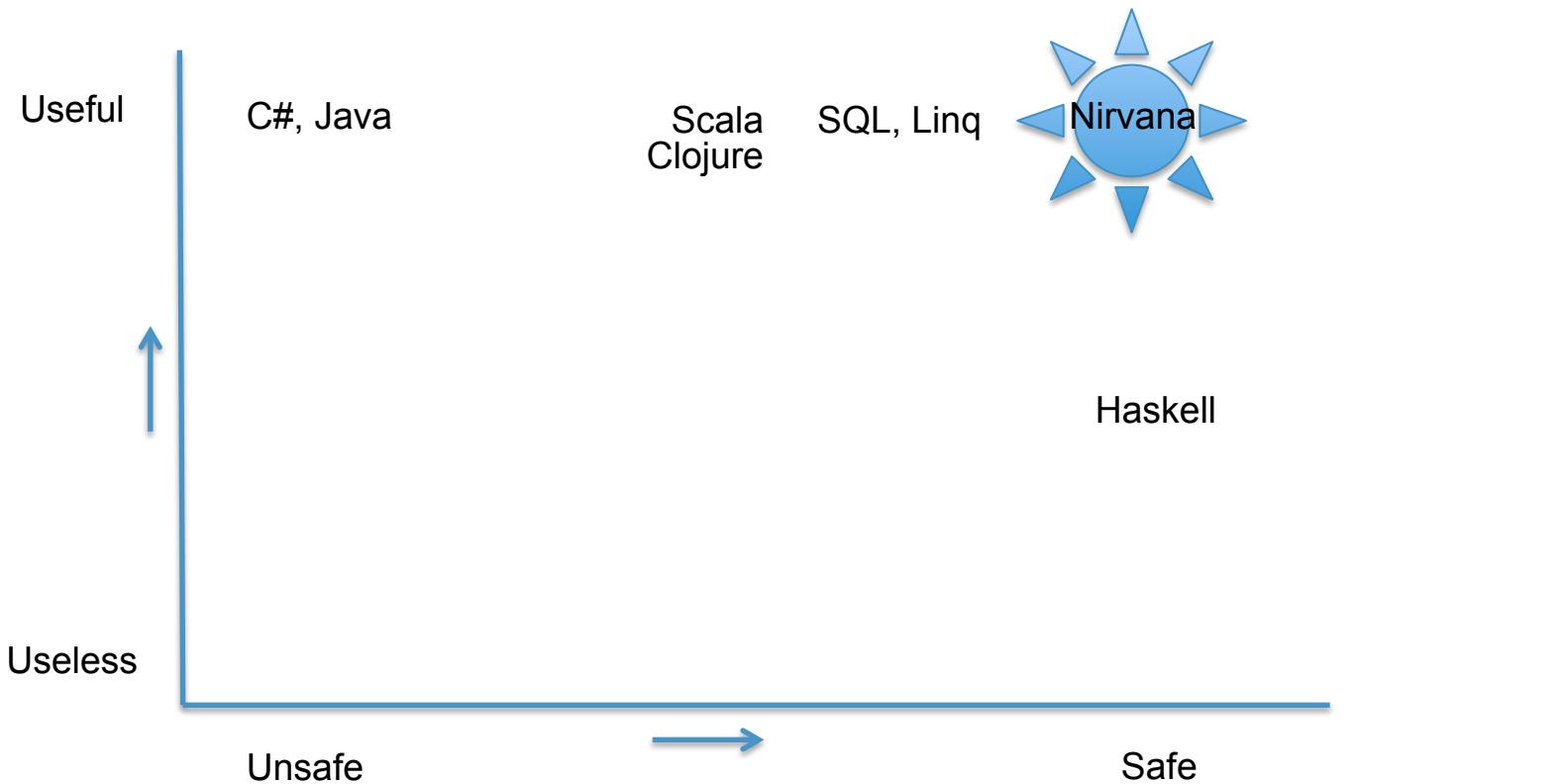
Examples in Haskell, Scala and Clojure

- Define a new type Person
- Create a list of Persons
- Count total length of first names with length greater than 4

<https://github.com/borkdude/fp-hu-may-2016/blob/master/code>

Degrees of FP

[Simon Peyton Jones, FP researcher](#)



Learning FP

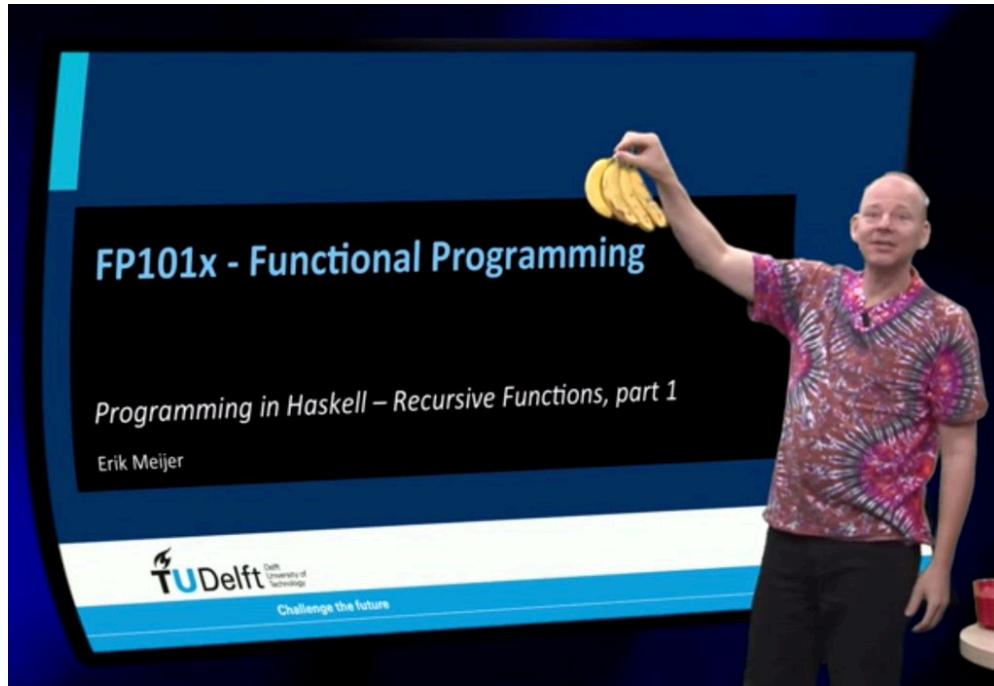
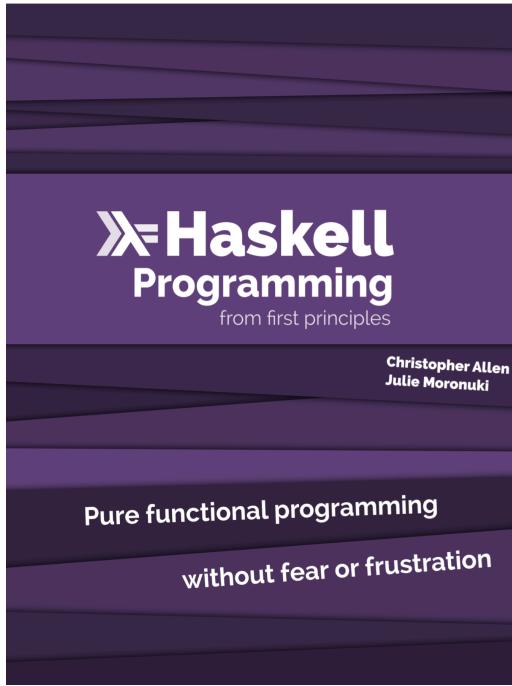
Choose a language that encourages FP

Haskell	Pure, lazy, statically typed
Clojure	Not pure. FP + Lisp on JVM, dynamically typed
Scala	Not pure. FP + (immutable) OO on JVM, statically typed

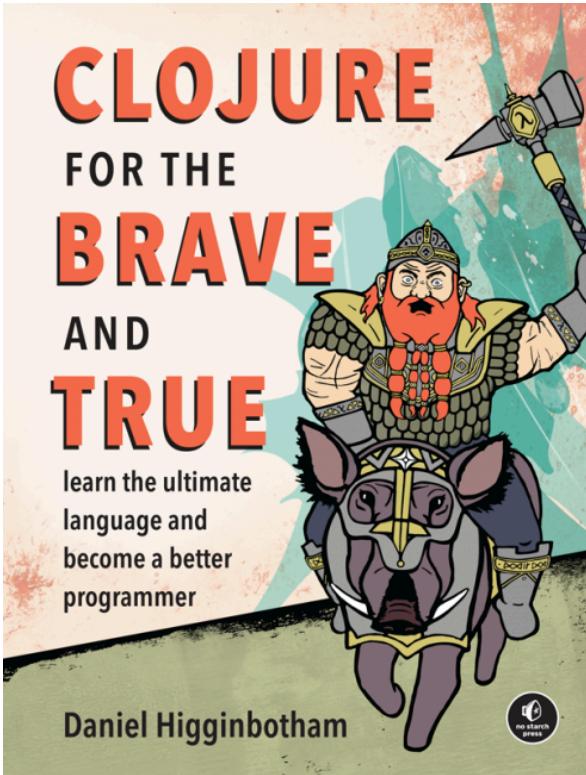
Apply the concepts in your favorite language

'Think like a fundamentalist, code like a hacker' – Erik Meijer

Haskell resources



Clojure resources



<http://michielborkent.nl/clojurecursus>

INLEIDING FUNCTIONEEL PROGRAMMEREN MET CLOJURE

Auteur: Michiel Borkent

Cursusjaar: 2012-2013

- [studiewijzer](#)
- [dictaat](#)
- [practicum](#)

Scala resources

Functional Programming in Scala



Coursera MOOC

The screenshot shows the Coursera course page for 'Functional Programming Principles in Scala' offered by EPFL. The page includes the EPFL logo, the course title, a brief description, and a preview video thumbnail.

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Functional Programming Principles in Scala

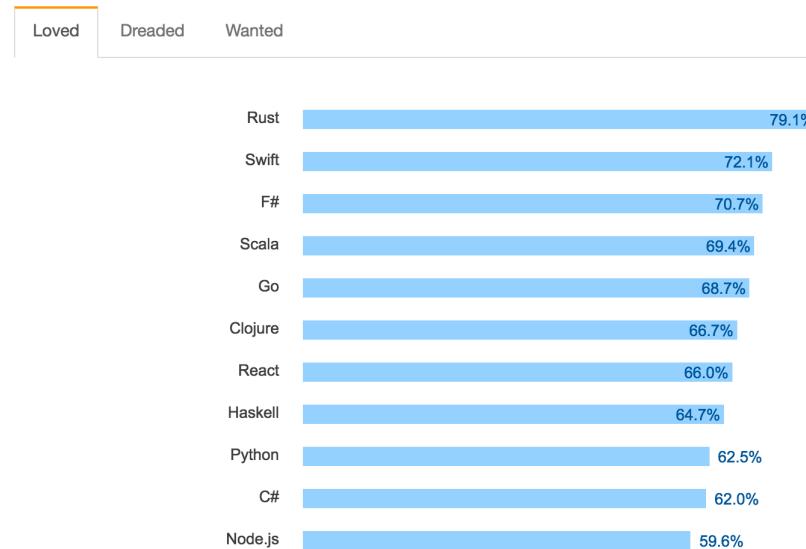
Learn about functional programming, and how it can be effectively combined with object-oriented programming. Gain practice in writing clean functional code, using the Scala programming language.

Watch Intro Video

```
case tree
  case class Branch(left: Tree, right: Tree)
  case class Leaf(x: Int) extends Tree
  val tree1 = Branch(Leaf(1), Leaf(2))
  def sumLeaf(tree: Tree): Int = tree match {
    case Leaf(x) => x
    case Branch(l, r) => sumLeaf(l) + sumLeaf(r)
  }
  sumLeaf(tree1)
```

Developers love FP

II. Most Loved, Dreaded, and Wanted



Source: <http://stackoverflow.com/research/developer-survey-2016>

Companies want FP devs

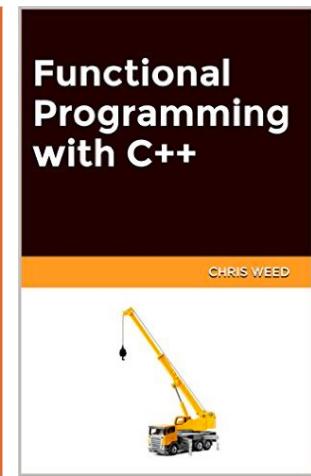
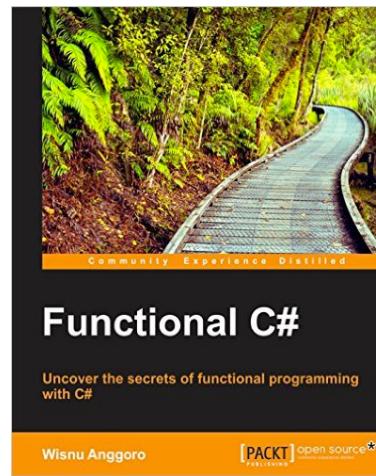
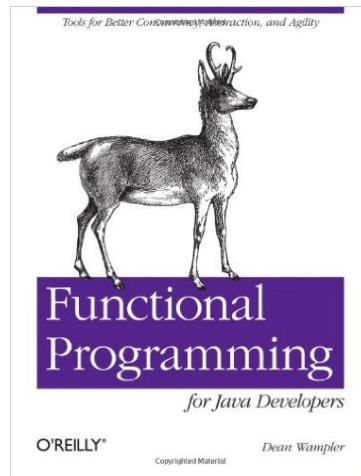
V. Top Paying Tech



Source: <http://stackoverflow.com/research/developer-survey-2016>

Once you know FP

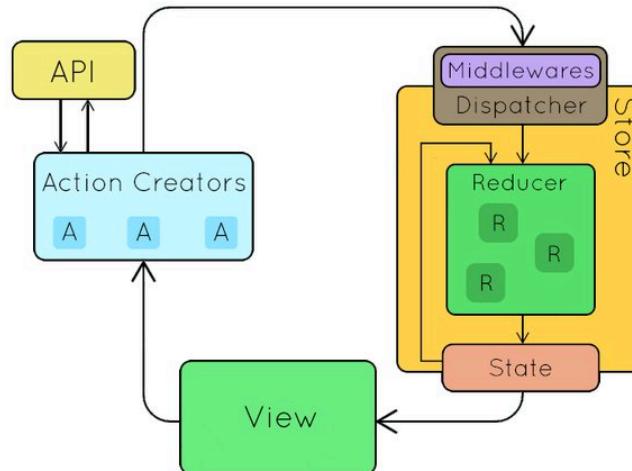
... you can apply and see it everywhere

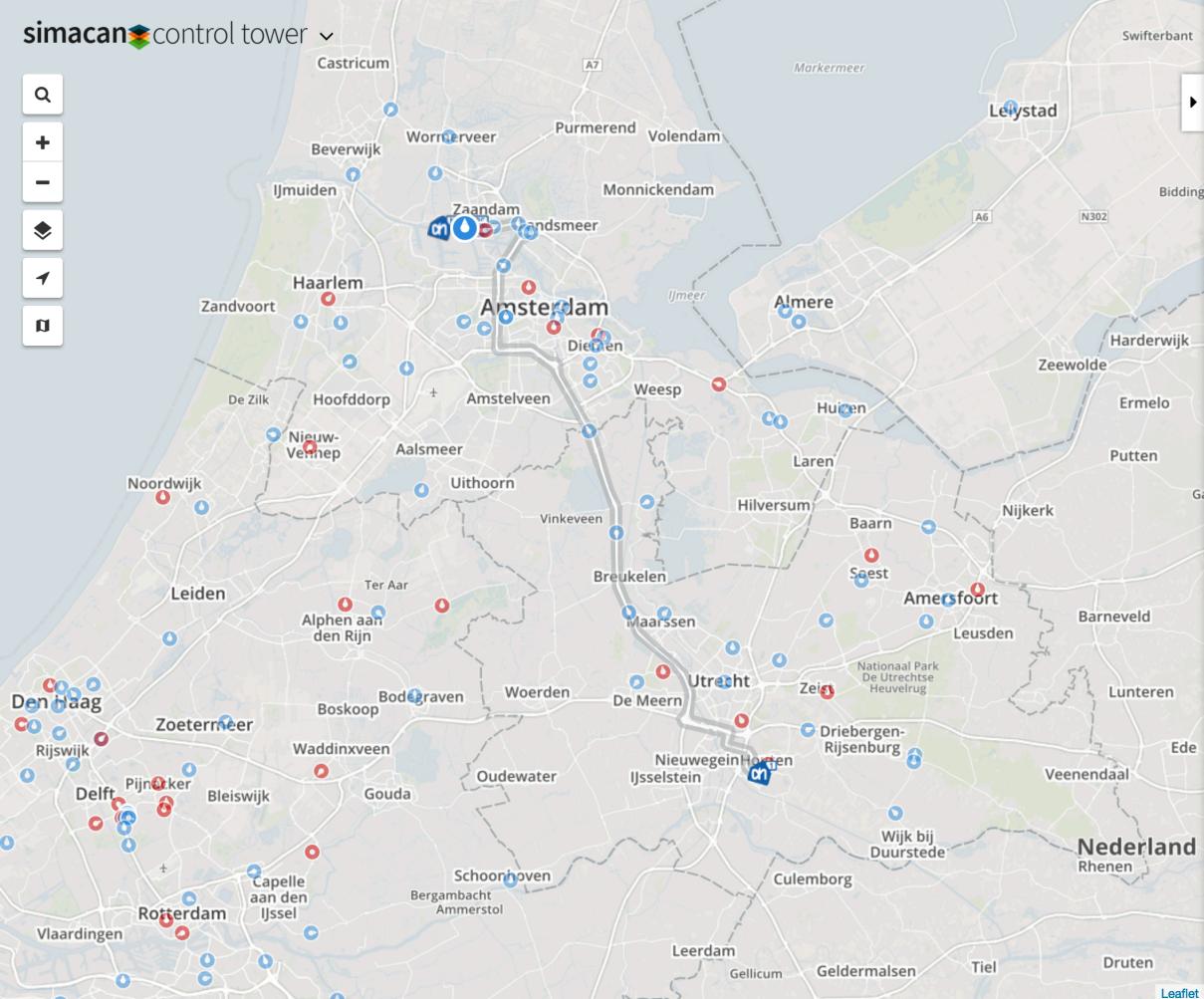


FP at my job



- Microservices in Scala
- UI in JavaScript using React, ImmutableJS and Redux





6-5-2016



AFWIJKINGEN

Deze rit heeft geen afwijkingen

GESCHIEDENIS

Nog geen berichten...

REALISATIE

PLANNING

12:21 - 12:41 DCZ Dock: 110 (HB)

14:14 - 14:50 8559 Houten (HB)

16:05 - 16:25 MCZ

RIT DETAILS

Laatste update:

Telefoonnummer:

Voertuig:

Kenteken:

Trailers:

Toon kruimelpad



Clojure

- dynamic language
- Lisp
- functional programming
- immutable data structures
- strong concurrency support
- embraces host platform (JVM, js)
- EDN

Clojure in industry



<https://www.youtube.com/watch?v=av9Xi6CNqq4>



<https://www.youtube.com/watch?v=iUC7noGU1mQ>

<http://dev.clojure.org/display/community/Clojure+Success+Stories>

Data literals

Keyword:	:a
Vector:	[1 2 3 4]
Hash map:	{:a 1, :b 2}
Set:	#{1 2 3 4}
List:	'(1 2 3 4)

Extensible Data Notation

```
{:key1 "Bar"  
  :key2 [1 2 3]  
  "key3", #{1.0 2.0 \c}  
  :key4, {:foo {:bar {:baz 'hello}}}}
```

```
(pr-str {:foo "bar"})  
(read-string "{\"foo \\\"bar\\\"}")
```

Syntax

$f(x) \rightarrow (f\ x)$

Syntax

```
if (...) {  
    ...  
} else {  
    ...  
}  
->  
        (if ...  
            ...  
            ... )
```

Syntax

```
var foo = "bar";
```

```
(def foo "bar")
```

JavaScript - ClojureScript

```
if (bugs.length > 0) {  
    return 'Not ready for release';  
} else {  
    return 'Ready for release';  
}
```

```
(if (pos? (count bugs))  
    "Not ready for release"  
    "Ready for release")
```

JavaScript - ClojureScript

```
var foo = {bar: "baz"};
foo.bar = "baz";
foo["abc"] = 17;
```

```
alert('foo')
new Date().getTime()
new
Date().getTime().toString()
```

```
(def foo (js-obj "bar" "baz"))
(set! (.bar foo) "baz")
(aset foo "abc" 17)
```

```
(js/alert "foo")
(.getTime (js/Date.))
(.. (js/Date.) (getTime)
(toString))
```

Persistent data structures

```
(def v [1 2 3])
(conj v 4) ;; => [1 2 3 4]
(get v 0) ;; => 1
(v 0) ;; => 1
```

Persistent data structures

```
(def m {:foo 1 :bar 2})  
(assoc m :foo 2) ;; => {:foo 2 :bar 2}  
(get m :foo) ;;=> 1  
(m :foo) ;;=> 1  
(:foo m) ;;=> 1  
(dissoc m :foo) ;;=> {:bar 2}
```

Functional programming

```
(def r (->>
            (range 10)      ;; (0 1 2 .. 9)
            (filter odd?)  ;; (1 3 5 7 9)
            (map inc)))    ;; (2 4 6 8 10)
;; r is (2 4 6 8 10)
```

Functional programming

```
;; r is (2 4 6 8 10)
(reduce + r)
;; => 30
(reductions + r)
;; => (2 6 12 20 30)
```

```
var sum = _.reduce(r, function(memo, num){ return memo + num; });
```

Sequence abstraction

Data structures as seqs

```
(first [1 2 3]) ;;=> 1
```

```
(rest [1 2 3]) ;;=> (2 3)
```

General seq functions: `map`, `reduce`, `filter`, ...

```
(distinct [1 1 2 3]) ;;=> (1 2 3)
```

```
(take 2 (range 10)) ;;=> (0 1)
```

See <http://clojure.org/cheatsheet> for more

Sequence abstraction

Most seq functions return lazy sequences:

```
(take 2 (map  
         (fn [n] (js/alert n) n)  
         (range)))
```

side effect

infinite lazy sequence of numbers

Mutable state: atoms

```
(def my-atom (atom 0))
@my-atom ;; 0
(reset! my-atom 1)
(reset! my-atom (inc @my-atom)) ;; bad idiom
(swap! my-atom (fn [old-value]
                  (inc old-value)))
(swap! my-atom inc) ;; same
@my-atom ;; 4
```

Lisp: macros

```
(map inc  
  (filter odd?  
    (range 10)))
```

```
(->> .....  
  (range 10)  
  (filter odd?)  
  (map inc))
```

thread last macro

Lisp: macros

```
(macroexpand  
'(->> (range 10) (filter odd?)))
```

```
;; => (filter odd? (range 10))
```

```
(macroexpand  
'(->> (range 10) (filter odd?) (map inc)))
```

```
;; => (map inc (filter odd? (range 10)))
```

Lisp: macros

JVM Clojure:

```
(defmacro defonce [x init]
  `(when-not (exists? ~x)
    (def ~x ~init)))
```

ClojureScript:

```
(defonce foo 1)
(defonce foo 2) ; no effect
```

notes:

- macros must be written in JVM Clojure
- are expanded at compile time
- generated code gets executes in ClojureScript

Artikel in Java Magazine over Clojure

http://michielborkent.nl/nljug/18_21_Clojure.pdf

Let's hack on an app!

<https://github.com/borkdude/fp-hu-may-2016>