Functional Programming with Clojure

Michiel Borkent
@borkdude

May 3rd 2017

Agenda

- Functional Programming
- FP with Clojure
- Demo with full stack Clojure

Part 1: Functional Programming

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.

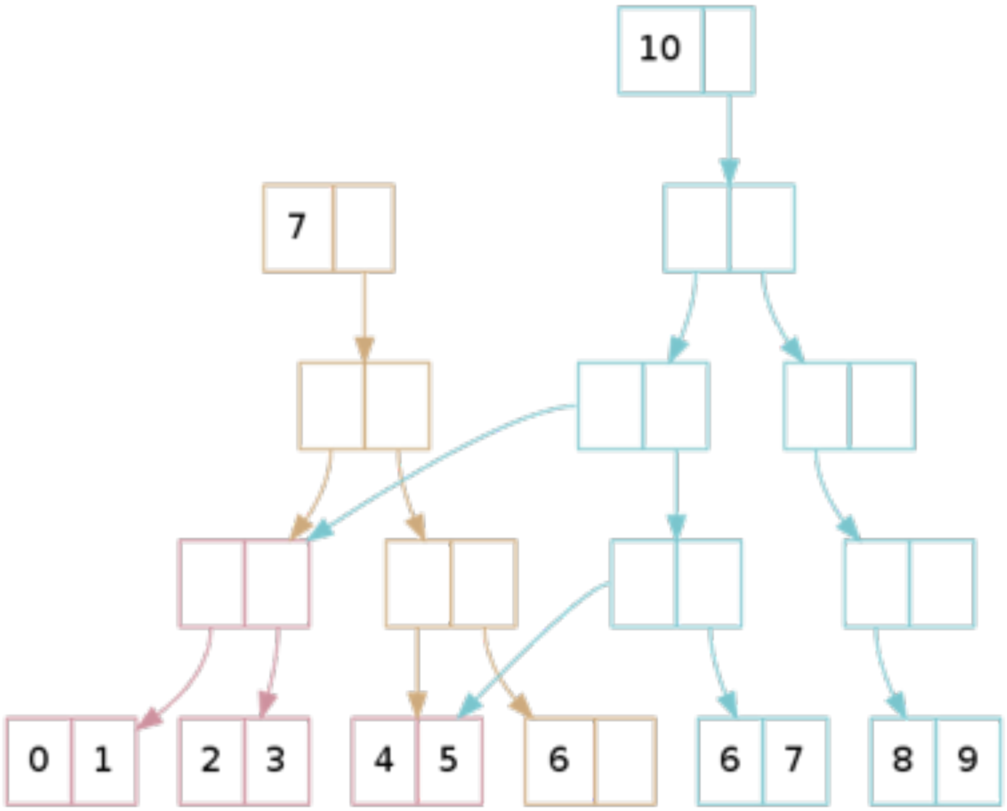
```
def addZero(l: List[Int]) = 0 :: l

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

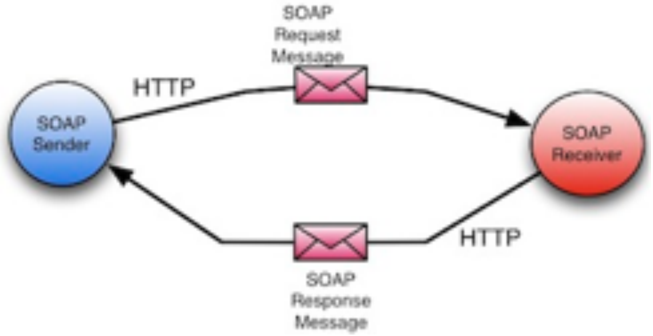


source: http://hypirion.com/musings/understanding-persistent-vector-pt-1

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 parallellisation 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/HU-2017-05/blob/master/codecode

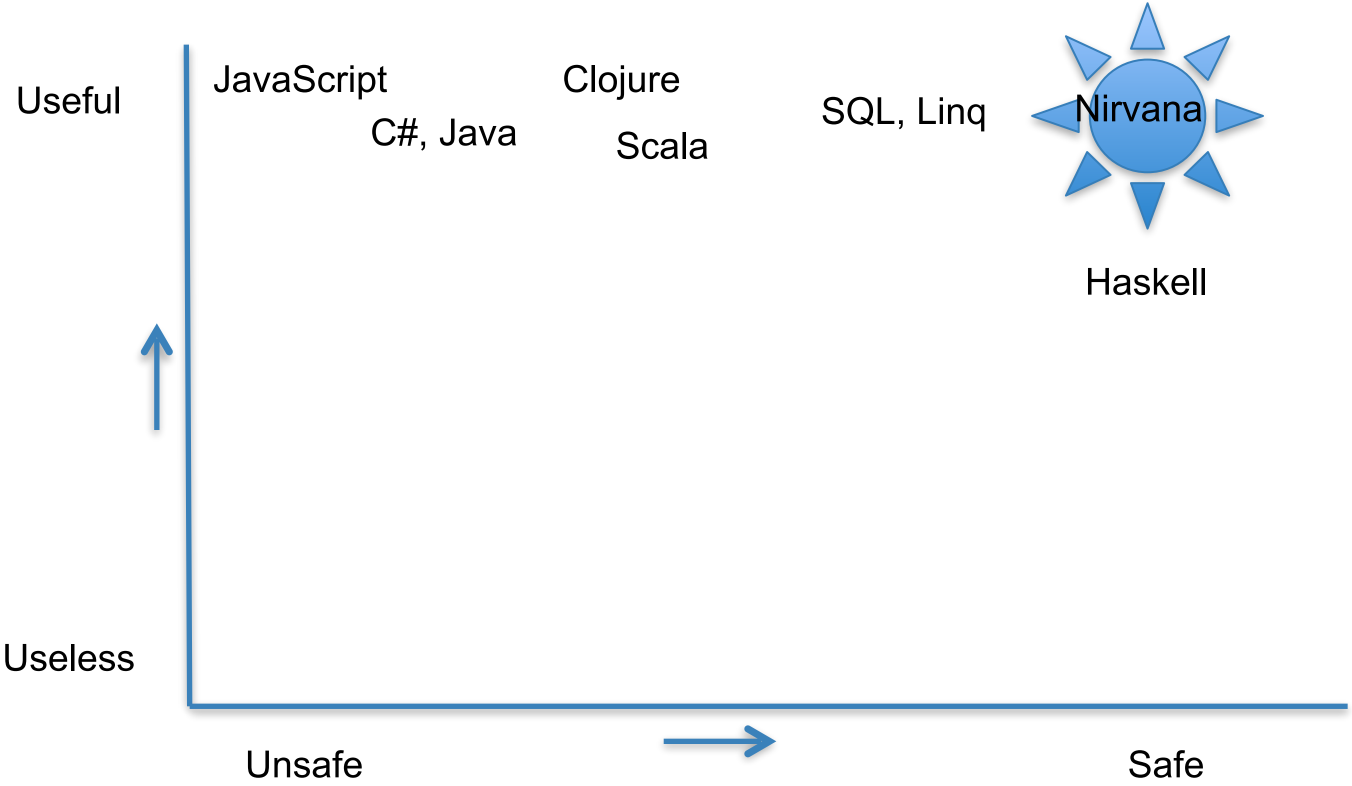
Less trivial example in Haskell and Clojure

- Generate HTML from a list of natural numbers

https://github.com/borkdude/HU-2017-05/blob/master/codecode

Degrees of FP

Simon Peyton Jones, FP researcher



Useful

JavaScript

C#, Java

Clojure

Scala

SQL, Linq

Nirvana

Haskell

Useless

Unsafe

Safe

Part 2: Clojure

Clojure

- Designed by Rich Hickey in 2007
- Frustration with Java, C++
- Deliver the same functionality faster
- Without giving up operational requirements

Non-goals

- Easy to learn by Java etc. developers
- Experiment in language design

Programs are (mostly) about data

- Language should not get in the way of data
- Good support for data literals
- Data transformations
- Data is immutable
- OO is not great for working with data
- Big part of program can be built using plain data

Clojure

- dynamic language
- lisp
- REPL
- functional programming
- immutable data structures
- strong concurrency support
- embraces host platform (JVM, CLR, browser, Node)
- EDN
- spec

Clojure in industry

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

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

https://clojure.org/community/success_stories

https://clojure.org/community/community_stories

Clojure philosophy

"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures." —Alan Perlis

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)  ->  (f x)
```

Syntax

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

->

```
(if ...
    ...
    ...)
```

Syntax

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

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

JavaScript - ClojureScript

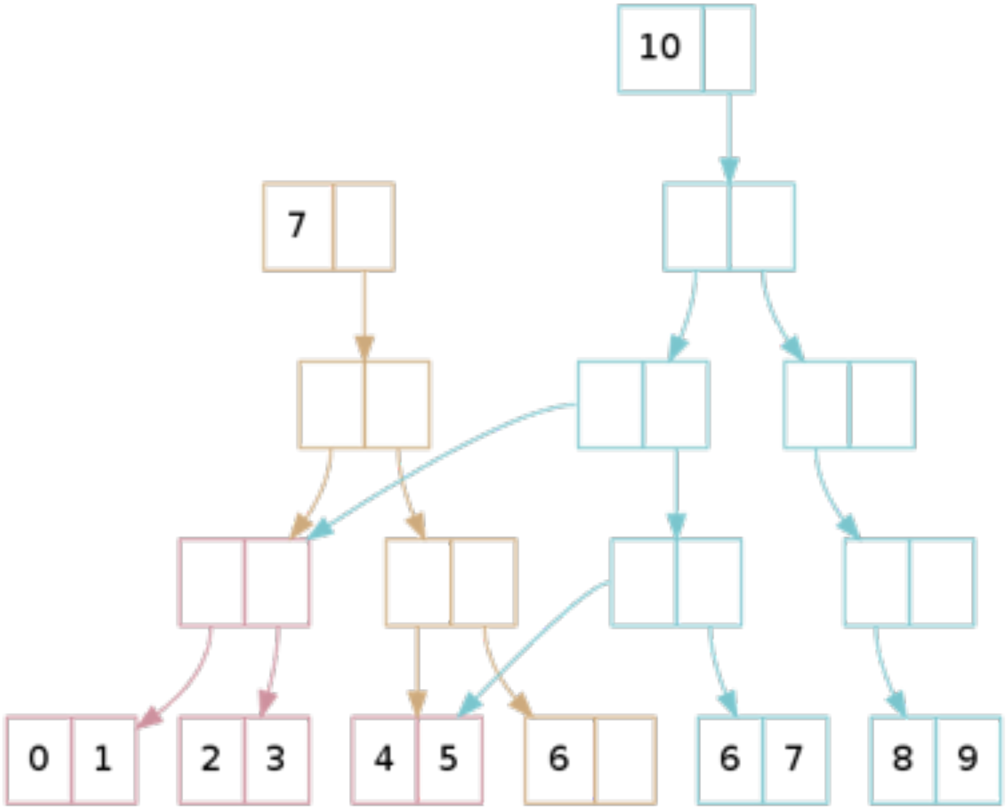
```javascript
if (bugs.length > 0) {
  return 'Not ready for
release';
} else {
  return 'Ready for release';
}
```

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

source: http://himera.herokuapp.com/synonym.html

Persistent data structures

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



source: http://hypirion.com/musings/understanding-persistent-vector-pt-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))
```

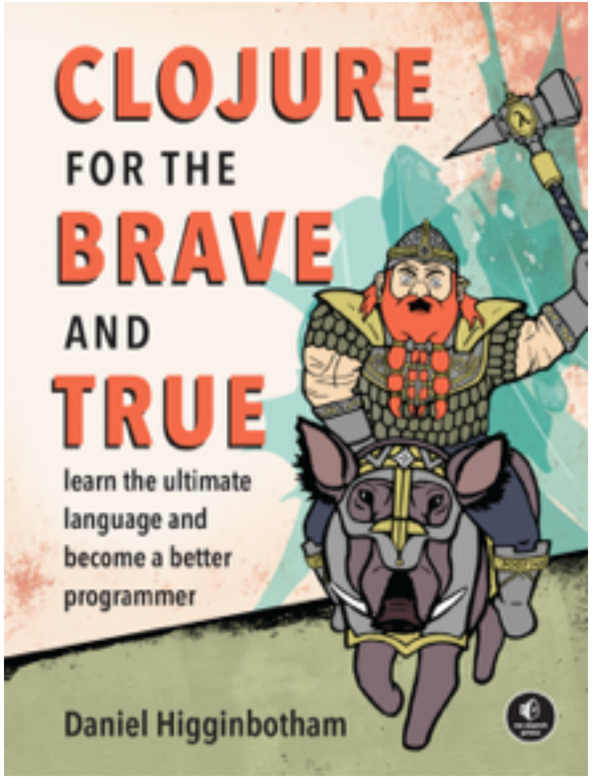
thread last macro

Lisp: macros

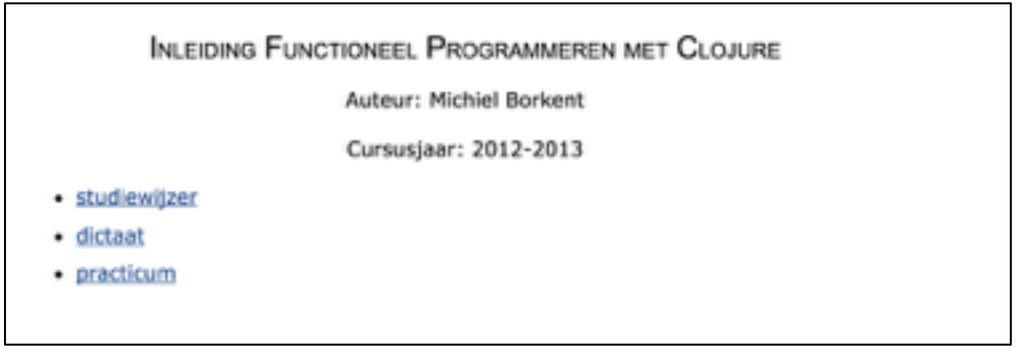
```
(macroexpand
  '(->> (range 10) (filter odd?)))

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

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

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

Part 3: Full Stack Clojure

Clojure resources



http://michielborkent.nl/clojurecursus



INLEIDING FUNCTIONEEL PROGRAMMEREN MET CLOJURE

Auteur: Michiel Borkent

Cursusjaar: 2012-2013

- studiewijzer
- dictaat
- practicum