Clojure

Functional Programming meets the JVM

Stefan Tilkov | @stilkov | innoQ

JUG Saxony
June 2011

Stefan Tilkov

stefan.tilkov@innoq.com
http://www.innoq.com/blog/st/
@stilkov

http://rest-http.info

→ 2., aktualisierte und erweiterte Auflage

Stefan Tilkov

REST und HTTP

Einsatz der Architektur des Web für Integrationsszenarien

dpunkt.verlag

SoftwareArchitekTOUR

Michael Stal - Christian Weyer -
Markus Völter - Stefan Tilkov

http://heise.de/developer/podcast

innoQ

http://www.innoq.com

Clojure

A practical Lisp variant for the JVM

Functional programming

Dynamic Typing

Full-featured macro system

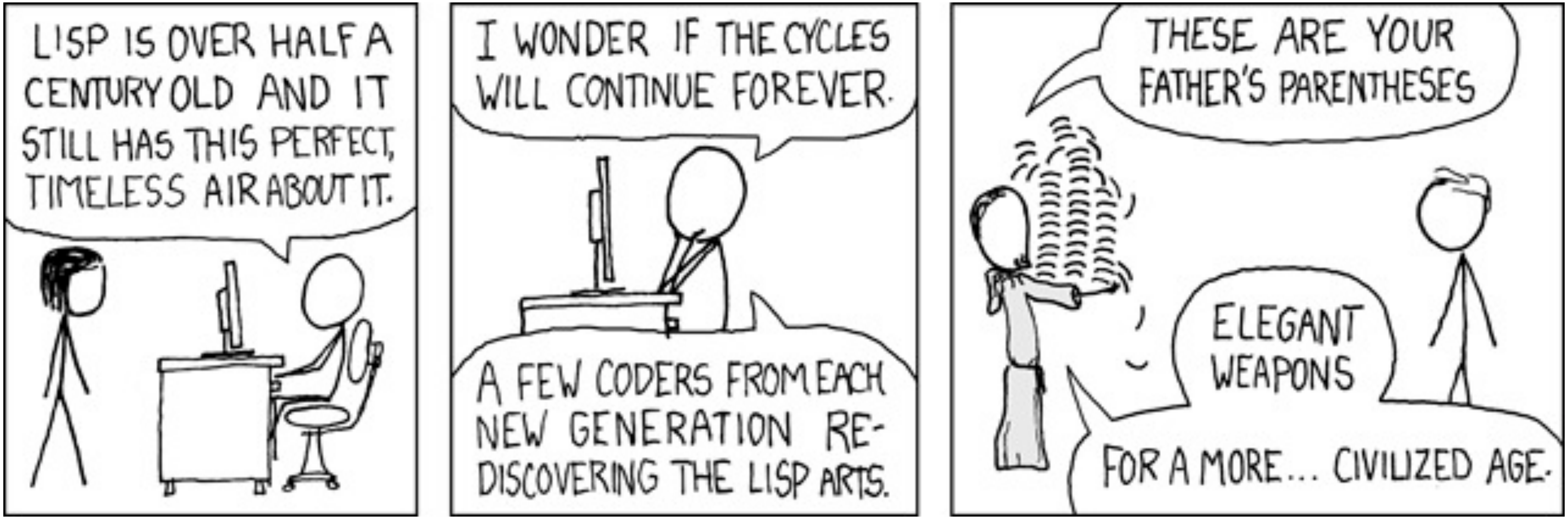
Concurrent programming support

Bi-directional Java interop

Immutable persistent data structures

Lisp??

Lots of irritating silly parentheses?


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

http://xkcd.com/297/

http://www.flickr.com/photos/nicolasrolland/3063007013/

Rich Hickey

http://www.tbray.org/ongoing/When/200x/2008/09/25/-big/R0010774.jpg.html

Intro

Clojure Environment

eclipse

NetBeans IDE

IntelliJIDEA

Clojuresque (Gradle)

maven

Leiningen

<APACHE ANT>

Data structures

Numbers	2 3 4 0.234 3/5 -2398989892820093093090292321
Strings	"Hello" "World"
Characters	\a \b \c
Keywords	:first :last
Symbols	a b c
Regexps	#"Ch.*se"
Lists	(a b c) ((:first :last "Str" 3) (a b))
Vectors	[2 4 6 9 23] [2 4 6 [8 9] [10 11] 9 23]
Maps	{:de "Deutschland", :fr "France"}
Sets	#{"Bread" "Cheese" "Wine"}

Syntax

“You’ve just seen it” – Rich Hickey

Syntax

```
(def my-set #{:a :b :c :c :c}) ;; #{:a :b :c}
(def v [2 4 6 9 23])
(v 0) ;; 2
(v 2) ;; 6
```

```
(def people {:pg "Phillip", :st "Stefan"})
(people :st) ;; "Stefan"
(:pg people) ;; "Phillip"
(:xyz people) ;; nil
```

```
(+ 2 2) ;; 4
(+ 2 3 5 4) ;; 14
(class (/ 4 3)) ;; clojure.lang.Ratio
(* (/ 4 3) 3) ;; 4
```

```
(format "Hello, %s # %d" "world" 1)
```

Syntax

```
(format "Hello, %s # %d" "world" 1)
; "Hello, World # 1"

(apply format ["Hello, %s # %d" "world" 1])


; (a 2 3)
(quote (a 2 3)) ;; (a 2 3)
'(a 2 3) ;; (a 2 3)


; Evaluation
(eval '(format "Hello, %s" "World"))
(eval (read-string "(+ 2 2)"))
```

Functions

```
(fn [x] (format "The value is %s\n" x))
;; user$eval__1706$fn__1707@390b755d
```

```
((fn [x] (format "The value is %s\n" x)) "Hello")
;; "The value is Hello"
```

```
(def testfn (fn [x] (format "The value is %s\n" x)))
(testfn "Hello")
```

```
(defn testfn [x] (format "The value is %s\n" x))
(testfn "Hello")
```

Functions

```
(defn even [x] (= 0 (rem x 2)))
(even 4) ;; true

(def even-alias even)
(even-alias 2) ;; true


(defn every-even? [l] (every? even l))
(every-even? '(2 4 6 8 9)) ;; false
(every? #(= 0 (rem % 2)) '(2 4 6 8 9)) ;; false
```

Closures

```
(defn make-counter [initial-value]
  (let [current-value (atom initial-value)]
    (fn []
      (swap! current-value inc))))
```

```
(def counter1 (make-counter 0))
(counter1) ;; 1
(counter1) ;; 2
```

```
(def counter2 (make-counter 17))
(counter1) ;; 3
(counter2) ;; 18
(counter1) ;; 4
(counter2) ;; 19
```

Recursion

```
(defn reduce-1 [f val coll]
  (if (empty? coll) val
    (reduce-1 f (f val (first coll)) (rest coll))))
```

```
(reduce-1 + 0 [1 2 3 4]) ;; 10
(reduce-1 + 0 (range 5)) ;; 10
(reduce-1 + 0 (range 50)) ;; 1225
(reduce-1 + 0 (range 50000)) ;; java.lang.StackOverflowError
```

Recursion

```
(defn reduce-2 [f val coll]
  (if (empty? coll) val
    (reduce-1 f (f val (first coll)) (rest coll)))))
```

```
(reduce-2 + 0 [1 2 3 4]) ;; 10
(reduce-2 + 0 (range 5)) ;; 10
(reduce-2 + 0 (range 50)) ;; 1225
(reduce-2 + 0 (range 50000)) ;; 1249975000
```

Example

```
(ns sample.grep
  "A simple complete Clojure program."
  (:use [clojure.contrib.io :only [read-lines]])
  (:gen-class))

(defn numbered-lines [lines]
  (map vector (iterate inc 0) lines))

(defn grep-in-file [pattern file]
  {file (filter #(re-find pattern (second %)) (numbered-lines (read-lines file)))})

(defn grep-in-files [pattern files]
  (apply merge (map #(grep-in-file pattern %) files)))

(defn print-matches [matches]
  (doseq [[fname submatches] matches, [line-no, match] submatches]
    (println (str fname ":" line-no ":" match))))

(defn -main [pattern & files]
  (if (or (nil? pattern) (empty? files))
    (println "Usage: grep <pattern> <file...>")
    (do
      (println (format "grep started with pattern %s and file(s) %s"
                         pattern (apply str (interpose ", " files))))
      (print-matches (grep-in-files (re-pattern pattern) files))
      (println "Done."))))
```

Macros

```
(def *debug* true)

(defn log [msg]
  (if *debug* (printf "%s: %s\n" (java.util.Date.) msg)))

(log "Hello, World")
Tue Apr 27 19:06:43 CEST 2010: Hello, World

(log (format "Hello, World %d" (* 9 9))))
Tue Apr 27 19:06:45 CEST 2010: Hello, World 81
```

Macros

```
(def *debug* true)

(defmacro log [body]
  `(if *debug* (printf "%s: %s\n" (java.util.Date.) ~body)))

(log "Hello, World")
Tue Apr 27 19:06:43 CEST 2010: Hello, World

(macroexpand '(log "Hello, World"))

(if user/*debug*
  (printf "%s: %s\n" (java.util.Date.) "Hello, World"))


(macroexpand '(log (format "Hello, World %d" (* 9 9))))

(if *debug*
  (printf "%s: %s\n" (java.util.Date.)
          (format "Hello, World %d" (* 9 9))))
```

Macros

```
(binding [*debug* false]
  (log "Hello, World"))
```

```
(defmacro with-debug [body]
  `(binding [*debug* true]
     ~body))
```

```
(with-debug
  (log "Hello, World")
  (log "Clojure rocks"))
```

```
Tue Apr 27 19:22:35 CEST 2010: Hello, World
Tue Apr 27 19:22:35 CEST 2010: Clojure rocks
```

Macros

```
(defmacro with-debug [body]
  `(binding [*debug* true]
     ~body))

(macroexpand '(binding [*debug* true]
  (log "Hello, World")))


(let*
[]
(clojure.core/push-thread-bindings (clojure.core/hash-map
(var *debug*) true))
(try
  (log "Hello, World")
  (finally (clojure.core/pop-thread-bindings))))
```

Lots of other cool stuff

Persistent data structures

Sequences

Support for concurrent programming

Destructuring

List comprehensions

Metadata

Optiional type information

Multimethods

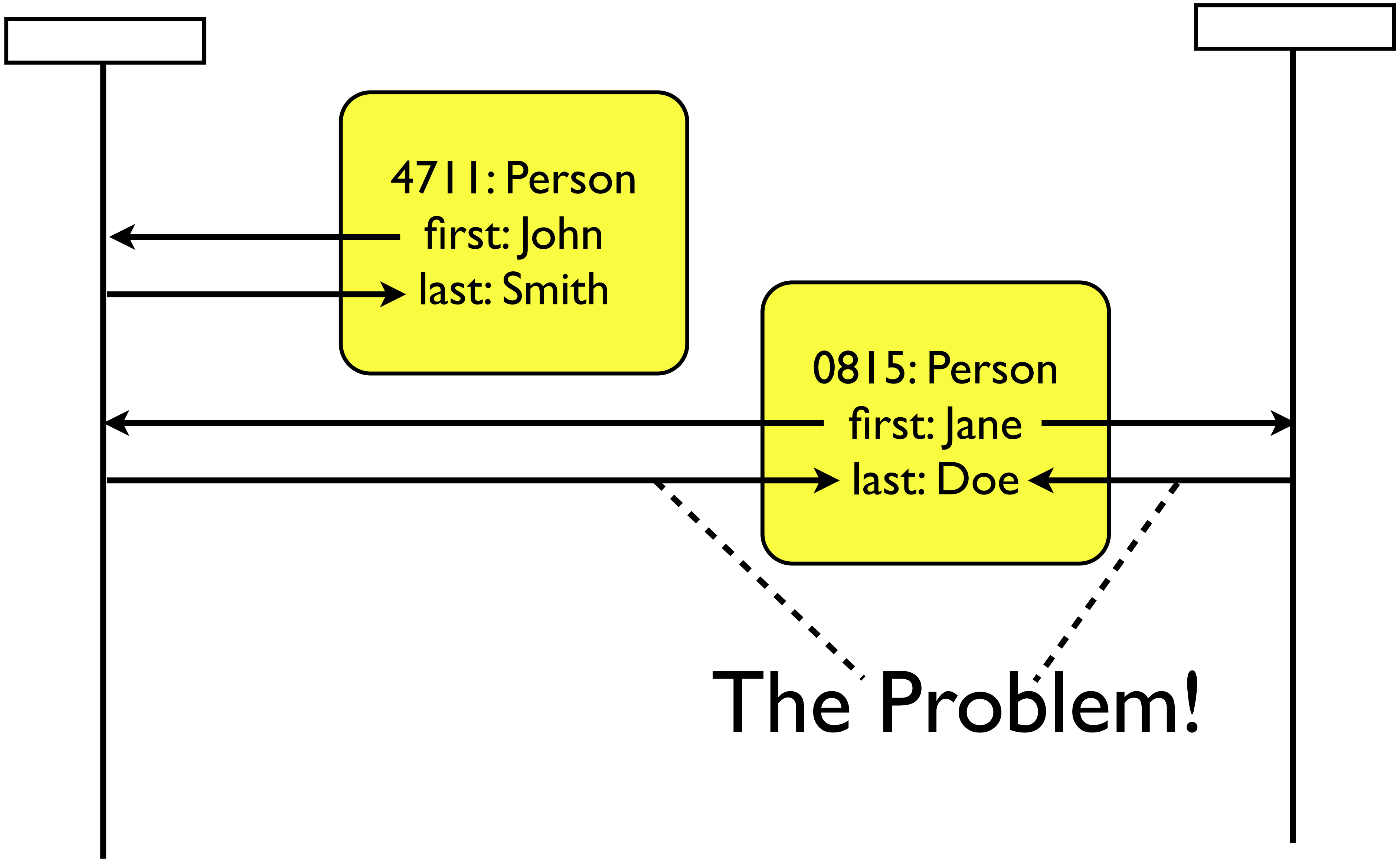
Pre & Post Conditions

Records/Protocols

Extensive core and contrib libraries

…

State



4711: Person
first: John
last: Smith

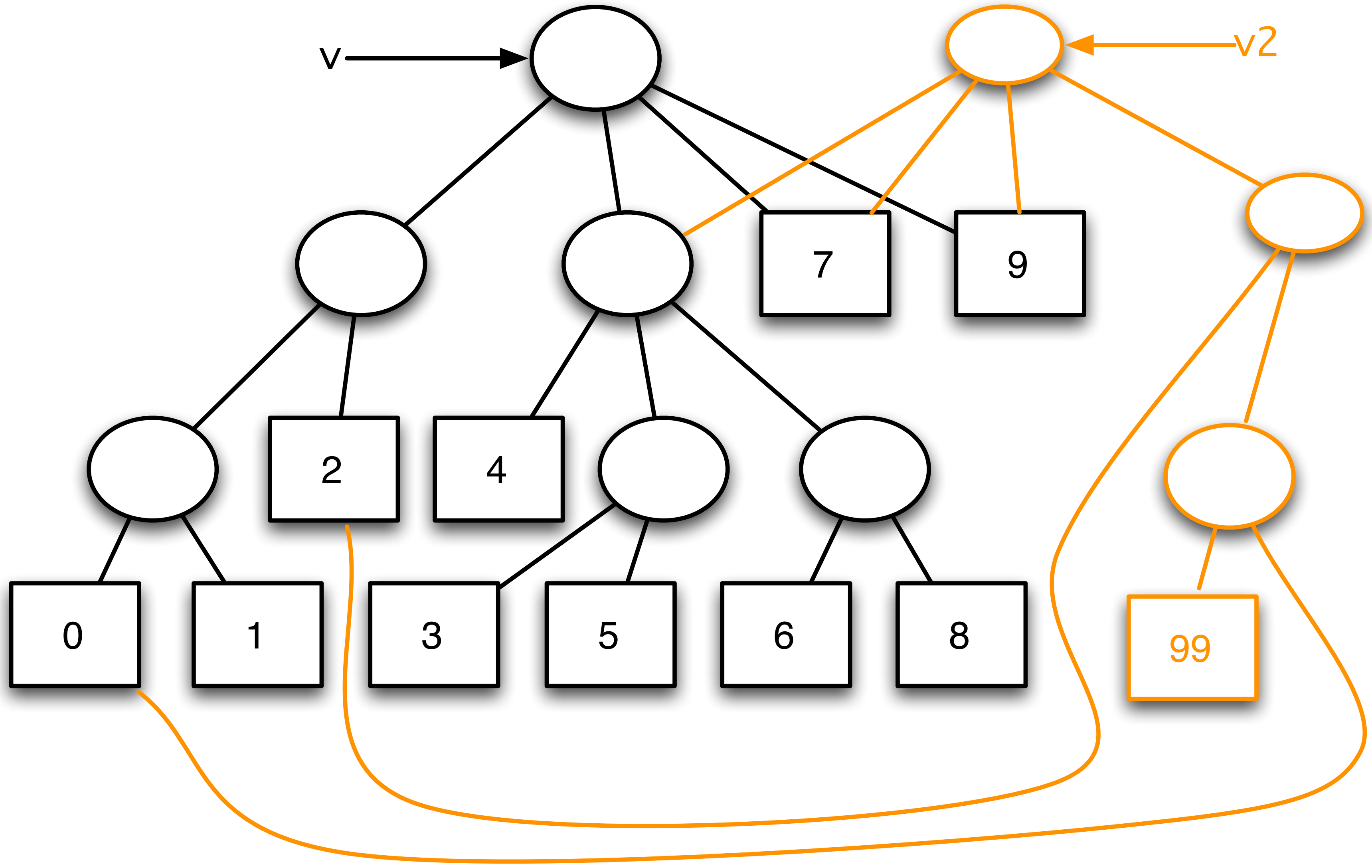
0815: Person
first: Jane
last: Doe

The Problem!

Immutability

```
user> (def v (vec (range 10)))
#'user/v
user> v
[0 1 2 3 4 5 6 7 8 9]
user> (assoc v 1 99)
[0 99 2 3 4 5 6 7 8 9]
 user> v
[0 1 2 3 4 5 6 7 8 9]
user> (def v2 (assoc v 1 99))
#'user/v2
user> v2
[0 99 2 3 4 5 6 7 8 9]
```

```
user> (def v (vec (range 10)))
user> (def v2 (assoc v 1 99))
```



Persistent Data Structures

Pure functional programming model

Efficient implementation

Structural sharing

Thread-safe

Iteration-safe

Based on Bit-partioned hash tries

“Transient” data structures if needed

Performance Guarantees

	hash-map	sorted-map	hash-set	sorted-set	vector	queue	list	lazy seq
conj	near-constant	logarithmic	near-constant	logarithmic	constant (tail)	constant (tail)	constant (head)	constant (head)
assoc	near-constant	logarithmic	-	-	near-constant	-	-	-
dissoc	near-constant	logarithmic	-	-	-	-	-	-
disj	-	-	near-constant	logarithmic	-	-	-	-
nth	-	-	-	-	near-constant	linear	linear	linear
get	near-constant	logarithmic	near-constant	logarithmic	near-constant	-	-	-
pop	-	-	-	-	constant (tail)	constant (head)	constant (head)	constant (head)
peek	-	-	-	-	constant (tail)	constant (head)	constant (head)	constant (head)
count	constant	constant	constant	constant	constant	constant	constant	linear

Sequences

Standard API for everything sequencable

Collections

Strings

Native Java arrays

java.lang.Iterable

Anything that supports first, rest, cons

Sequences

Standard API for everything sequencable

“Lazy” sequences

```
(def n (iterate (fn [x] (+ x 1)) 0))
(def fives (map #(* 5 %) n))
(take 10 fives)
```

Sequences

Standard API for everything sequencable

“Lazy” sequences

Extensive library

apply
butlast
concat
cons
cycle
distinct
doall
dorun
doseq
drop
drop-last
drop-while
empty?
every?
ffirst
file-seq
filter
first
fnext
for
interleave
interpose
into
into-array
iterate
iterator-seq
keys
last
lazy-cat
lazy-seq
line-seq
map
mapcat
next
nfirst
nnext
not-any?
not-empty
not-every?
nth
nthnext
partition
pmap
range
re-seq
reduce
remove
repeat
repeatedly
replace
replicate
rest
resultset-seq
reverse
rseq
rsubseq
second
seq
seq?
seque
set
some
sort
sort-by
split-at
split-with
subseq
take
take-nth
take-while
to-array-2d
tree-seq
vals
vec
when-first
xml-seq
zipmap
…

Concurrency Support

Core Ideas

Everything immutable

Shared state for reading

No changes to shared state

Isolated threads

Re-use of platform facilities

Java Integration

(java.util.concurrent.Callable)

def & binding

```
(def some-var 10)

(binding [some-var 30]
  (println some-var)) ;; 30
```

```
(def some-var 10)
(println some-var) ;; 10

(binding [some-var some-var]
  (println some-var) ;; 10
  (set! some-var 30)
  (println some-var)) ;; 30
```

Atoms

```
(def a (atom "Initial Value"))
(println @a) ;; "Initial Value"

(swap! a #(apply str (reverse %)))
(println @a) ;; "eulaV laitinI"

(swap! a #(apply str (reverse %)))
(println @a) ;; "Initial Value"
```

Atoms

```
(defn run-thread-fn [f]
  (.start (new Thread f)))

(defn add-list-item [coll-atom x]
  (swap! coll-atom #(conj % x)))

(def int-list (atom ())) ;; ()
(run-thread-fn #(add-list-item int-list 5)) ;; (5)
(run-thread-fn #(add-list-item int-list 3)) ;; (3 5)
(run-thread-fn #(add-list-item int-list 1)) ;; (1 3 5)


(def int-list (atom ())) ;; ()
(let [run-fn (fn [x] (run-thread-fn #(add-list-item int-list x)))]
  (doall (map run-fn (range 100))))
;; (98 97 96 ... 0)
```

Refs

```
(defn make-account
  [balance owner]
  {:balance balance, :owner owner})


(defn withdraw [account amount]
  (assoc account :balance (- (account :balance) amount)))


(defn deposit [account amount]
  (assoc account :balance (+ (account :balance) amount)))
```

Refs

```
(defn transfer
  [from to amount]
  (dosync
   (alter from withdraw amount)
   (alter to deposit amount)))

(defn init-accounts []
  (def acc1 (ref (make-account 1000 "alice")))
  (def acc2 (ref (make-account 1000 "bob")))
  (def acc3 (ref (make-account 1000 "charles"))))
```

Refs

```
(init-accounts)
```

```
                  acc1: {:balance 1000, :owner "alice"}
                  acc2: {:balance 1000, :owner "bob"}
                  acc3: {:balance 1000, :owner "charles"}
```

```
(do
  (run-thread-fn #(transfer acc1 acc2 100))
  (transfer acc3 acc1 400))
```

```
                  acc1: {:balance 1300, :owner "alice"}
                  acc2: {:balance 1100, :owner "bob"}
                  acc3: {:balance 600, :owner "charles"}
```

Refs

```
acc1: {:balance 1300, :owner "alice"}
acc2: {:balance 1100, :owner "bob"}
acc3: {:balance 600, :owner "charles"}
```

```clojure
(defn slow-transfer
  [from to amount]
  (dosync
   (sleep 1000)
   (alter from withdraw amount)
   (alter to deposit amount)))
```

```clojure
(do
  (run-thread-fn #(slow-transfer acc1 acc2 100))
  (transfer acc3 acc1 400))
```

```
acc1: {:balance 1600, :owner "alice"}
acc2: {:balance 1200, :owner "bob"}
acc3: {:balance 200, :owner "charles"}
```

Software Transactional Memory (STM)

Multi-version concurrency control (MVCC)

Atomic changes to multiple refs

Non-blocking, retry-based

“Read committed”

Can't help with non-pure functions

Works with atoms and agents

```
deref/@  ensure  commute  ref-set  alter  throw
```

Software Transactional Memory

`deref/@`	Reads value of reference, blocks none
`ensure`	Reads value of reference, blocks writers
`commute`	Reads value of reference, blocks none, delayed write, last writer wins
`ref-set`	Changes reference to new value, blocks writers
`alter`	Atomically reads, computes, sets reference value, blocks writers
`throw`	Rolls back transaction

Agents

Asynchronous execution

Run on java.util.concurrent thread pool

```
(let [my-agent (agent 0)
      slow-fn  (fn [x]
                 (sleep 1000)
                 (inc x))]
  (send my-agent slow-fn)
  (println @my-agent)
  (sleep 2000)
  (println @my-agent))
;; 0
;; 1
```

`agent` `send` `send-off` `deref/@` `await` `await-for`

Agents

`agent`	Creates agent with initial value
`send`	Dispatch function to agent for execution
`send-off`	Dispatch long-running function
`deref/@`	Read agent value
`await`	Wait for agent to execute function(s) dispatched from current thread
`await-for`	Same as await, but with timeout

Validators

```
(def some-var 10)

(set-validator! #'some-var #(< % 100))

(def some-var 101) ;; Invalid reference state
;; [Thrown class java.lang.IllegalStateException]

(def some-var)
(defn limit-validator [limit]
  (fn [new-value]
    (if (< new-value limit)
      true
      (throw (Exception.
                (format "Value %d is larger than limit %d"
                         new-value limit))))))
```

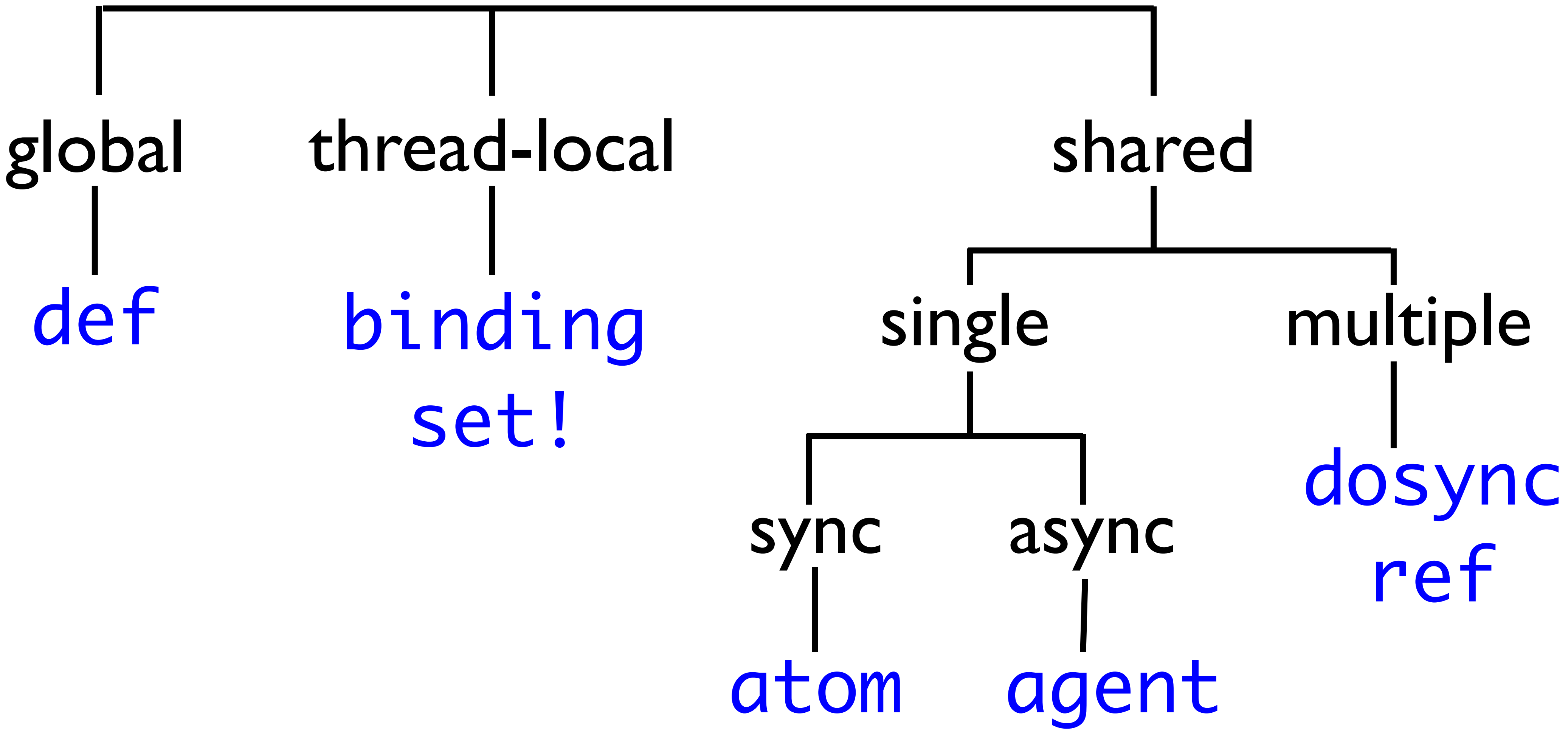
```
(set-validator! #'some-var (limit-validator 100))
(def some-var 101)
;; Value 101 is larger than limit 100
;; [Thrown class java.lang.Exception]
```

Watchers

```
(def *a* (atom 0))
(def *events* (atom ()))

(defn log-event
  [coll s]
  (swap! coll conj s))

(log-event *events* "some event") ;; ("some event")
(log-event *events* "yet another event") ;; ("yet another event" "some event")

(defn log-value-change
  [key ref old new]
  (if (= key :log)
    (log-event *events* (format "value of %s changed from %d to %d" ref old new))))

(log-value-change :log 'x 0 1)
;; ("value of x changed from 0 to 1" "yet another event" "some event")
(add-watch a :log log-value-change)
(swap! a inc) ;; 1

(deref *events*)
;; ("value of clojure.lang.Atom@59829c6b changed from 0 to 1"
;;  "value of x changed from 0 to 1" "yet another event" "some event")
```

Futures & Promises

```
user> (doc future)
-------------------------
clojure.core/future
([& body])
Macro
  Takes a body of expressions and yields a future object that will
  invoke the body in another thread, and will cache the result and
  return it on all subsequent calls to deref/@. If the computation has
  not yet finished, calls to deref/@ will block.

user> (doc promise)
-------------------------
clojure.core/promise
([])
  Alpha - subject to change.
  Returns a promise object that can be read with deref/@, and set,
  once only, with deliver. Calls to deref/@ prior to delivery will
  block. All subsequent derefs will return the same delivered value
  without blocking.
```



global

def

thread-local

binding

set!

shared

single

sync

atom

async

agent

multiple

dosync

ref

Summary

Built on immutablity from the ground up

Powerful collections

Extensive sequence library

Built-in concurrency primitives

Java Integration

Clojure → Java

```clojure
(new java.lang.String "Hello")
(java.lang.String. "Even quicker")
(java.io.File/separator)
(import '(java.io InputStream File))
(File/separator)
(. System/out println "Hello")
(.println System/out "Hello")

(defn blank? [s] (every? #(Character/isWhitespace %) s))
(blank? "some string") ;; false
(blank? "") ;; true

(every? #(instance? java.util.Collection %)
        '([1 2] '(1 2) #{1 2}))
;; true
```

Clojure ↔ Java

```
(import '(java.util Vector Collections))

(def java-collection (Vector.))
(doto java-collection
  (.add "Gamma")
  (.add "Beta")
  (.add "Alpha"))
;; #<Vector [Gamma, Beta, Alpha]>

(defn make-comparator [compare-fn]
     (proxy [java.util.Comparator] []
       (compare [left right] (compare-fn left right))))


(Collections/sort java-collection
                  (make-comparator #(. %1 compareTo %2)))

;; #<Vector [Alpha, Beta, Gamma]>
```

Clojure ← Java

```
package com.innoq.test;
public interface ClojureInterface {
    String reverse(String s);
}
```

```
(ns com.innoq.test)

(gen-class
  :name    com.innoq.test.ClojureClass
  :implements [com.innoq.test.ClojureInterface]
  :prefix class-)

(defn class-reverse
  [this s]
  (apply str (reverse s)))
```

```
package com.innoq.test;
public class ClojureMain {
    public static void main(String[] args) {
        ClojureInterface cl = new ClojureClass();
        System.out.println("String from Clojure: " + cl.reverse("Hello, World"));
    }
}
```

Core

http://clojure.org/

clojure@googlegroups.com

#clojure freenode

build.clojure.org

http://en.wikibooks.org/wiki/Clojure

http://www.assembla.com/wiki/show/clojure/Getting_Started

http://github.com/relevance/labrepl

Blogs

http://www.bestinclass.dk/index.php/blog/

http://stuartsierra.com/

http://technomancy.us/

http://kotka.de/blog/

http://blog.fogus.me/

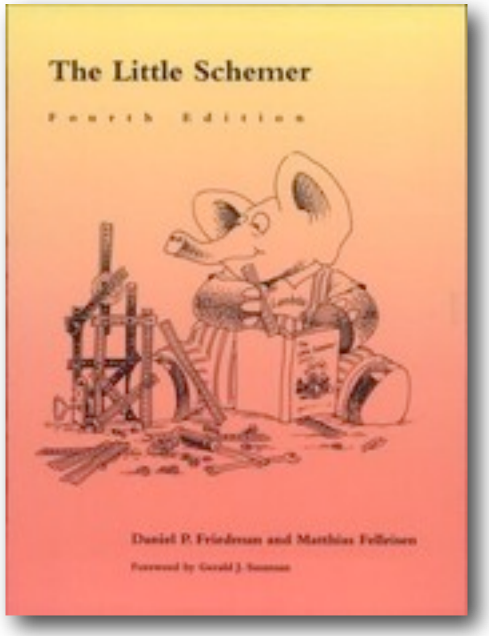
Screencasts

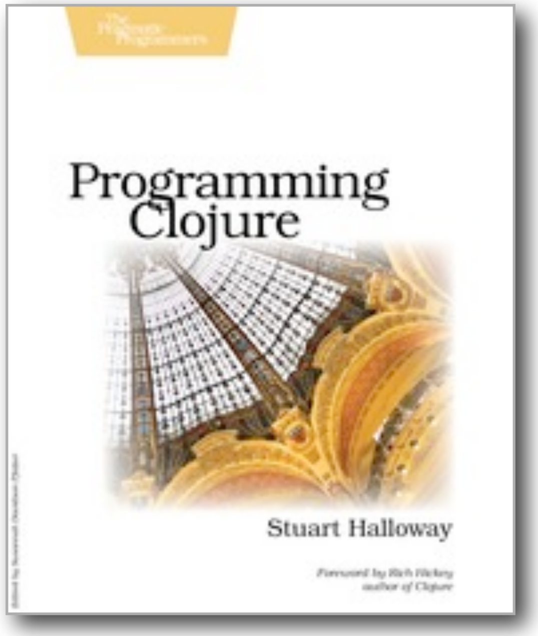
http://technomancy.us/136

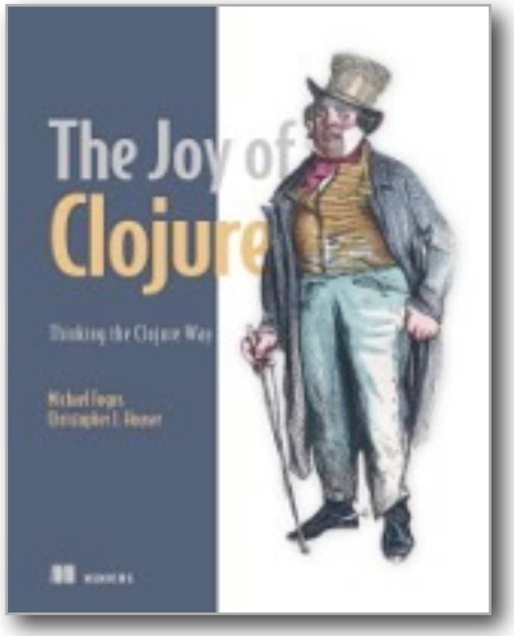
http://peepcode.com/products/functional-programming-with-clojure

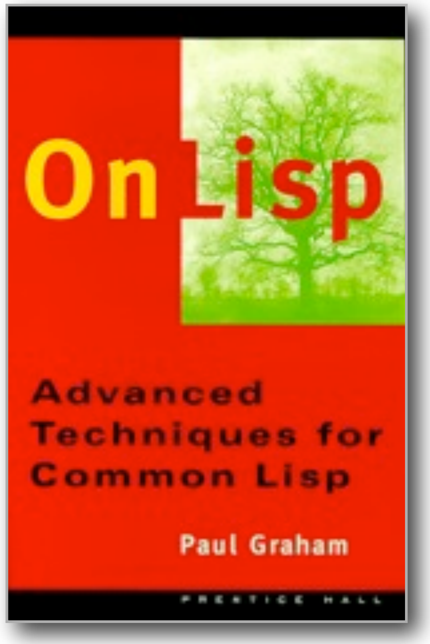
http://vimeo.com/channels/fulldisclojure

Books


The Little Schemer
Fourth Edition
Daniel P. Friedman and Matthias Felleisen
Foreword by Gerald J. Sussman


The Pragmatic Programmers
Programming Clojure
Stuart Halloway
Foreword by Rich Hickey author of Clojure





Auf deutsch ...

Clojure: Ein pragmatisches Lisp für die JVM
Stefan Tilkov, heise Developer Juli 2010
http://bit.ly/ceLkmT

Clojure: Funktional, parallel, genial
Burkhard Neppert, Stefan Tilkov
dreiteilige Artikelserie in JavaSPEKTRUM 02-04/2010
http://bit.ly/caHJ8f

STEFAN KAMPHAUSEN / TIM OLIVER KAISER
Clojure, dpunkt Verlag
http://www.dpunkt.de/buecher/3372.html

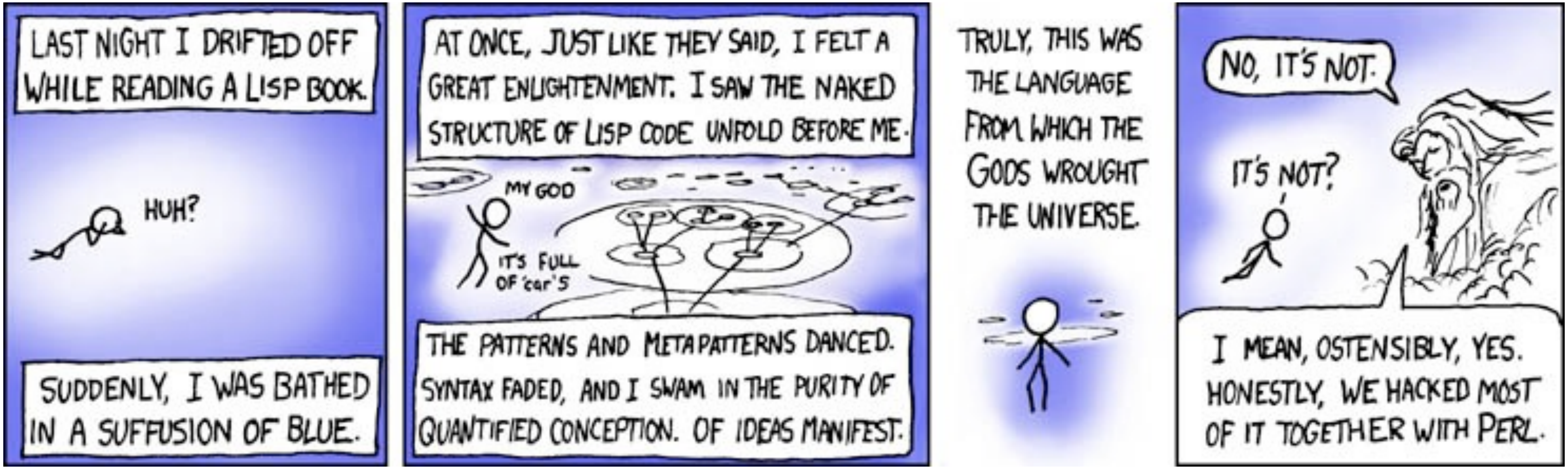
Stefan Kamphausen · Tim Oliver Kaiser

Clojure

- Grundlagen
- Concurrent Programming
- Java

dpunkt.verlag

Q&A


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

http://xkcd.com/224/

Stefan Tilkov
stefan.tilkov@innoq.com
http://www.innoq.com/blog/st/
Twitter: stilkov