clojure

http://github.com/stuarthalloway/clojure-presentations

stuart halloway

http://thinkrelevance.com

clojure's four elevators

java interop

lisp

functional

state

1. java interop

java new

java	`new Widget("foo")`
clojure	`(new Widget "foo")`
clojure sugar	`(Widget. "red")`

access static members

java	`Math.PI`
clojure	`(. Math PI)`
clojure sugar	`Math/PI`

access instance members

java	`rnd.nextInt()`
clojure	`(. rnd nextInt)`
clojure sugar	`(.nextInt rnd)`

chaining access

java	`person.getAddress().getZipCode()`
clojure	`(. (. person getAddress) getZipCode)`
clojure sugar	`(.. person getAddress getZipCode)`

parenthesis count

java	( ) ( ) ( ) ( )
clojure	( ) ( ) ( )

atomic data types

type	example	java equivalent
string	"foo"	String
character	\f	Character
regex	#"fo*"	Pattern
a. p. integer	42	Integer/Long/BigInteger
double	3.14159	Double
a.p. double	3.14159M	BigDecimal
boolean	TRUE	Boolean
nil	nil	null
symbol	foo, +	N/A
keyword	:foo, ::foo	N/A

example:
refactor apache
commons isBlank

initial implementation

```java
public class StringUtils {
  public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (int i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- type decls

```
public class StringUtils {
  public isBlank(str) {
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- class

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  for (i = 0; i < strLen; i++) {
    if ((Character.isWhitespace(str.charAt(i)) == false)) {
      return false;
    }
  }
  return true;
}
```

+ higher-order function

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  every (ch in str) {
    Character.isWhitespace(ch);
  }
  return true;
}
```

- corner cases

```
public isBlank(str) {
  every (ch in str) {
    Character.isWhitespace(ch);
  }
}
```

lispify

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

clojure is a better java than java

2. lisp

what makes lisp different

feature	industry norm	cool kids	clojure
conditionals	✔	✔	✔
variables	✔	✔	✔
garbage collection	✔	✔	✔
recursion	✔	✔	✔
function type		✔	✔
symbol type		✔	✔
whole language available		✔	✔
everything’s an expression			✔
homoiconicity			✔

http://www.paulgraham.com/diff.html

regular code

```
foo.bar(x,y,z);

foo.bar x y z
```

special forms

imports

scopes

protection

metadata

control flow

anything using a keyword

outside lisp, special forms

look different

may have special semantics unavailable to you

prevent reuse

in a lisp, special forms

look just like anything else

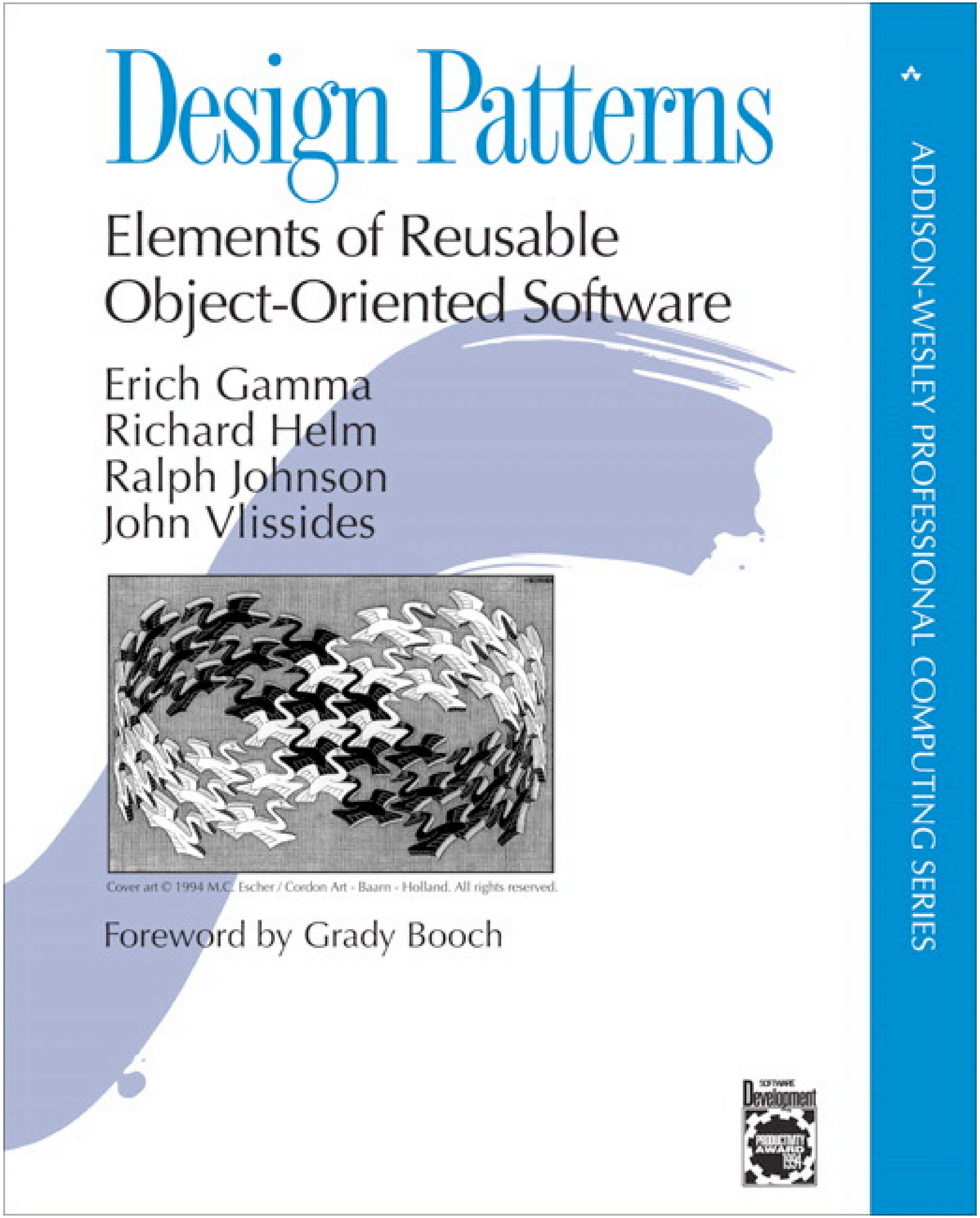
may have special semantics **available** to you

can be augmented with macros

all forms created equal

form	syntax	example
function	list	`(println "hello")`
operator	list	`(+ 1 2)`
method call	list	`(.trim " hello ")`
import	list	`(require 'mylib)`
metadata	list	`(with-meta obj m)`
control flow	list	`(when valid? (proceed))`
scope	list	`(dosync (alter ...))`

who cares?



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

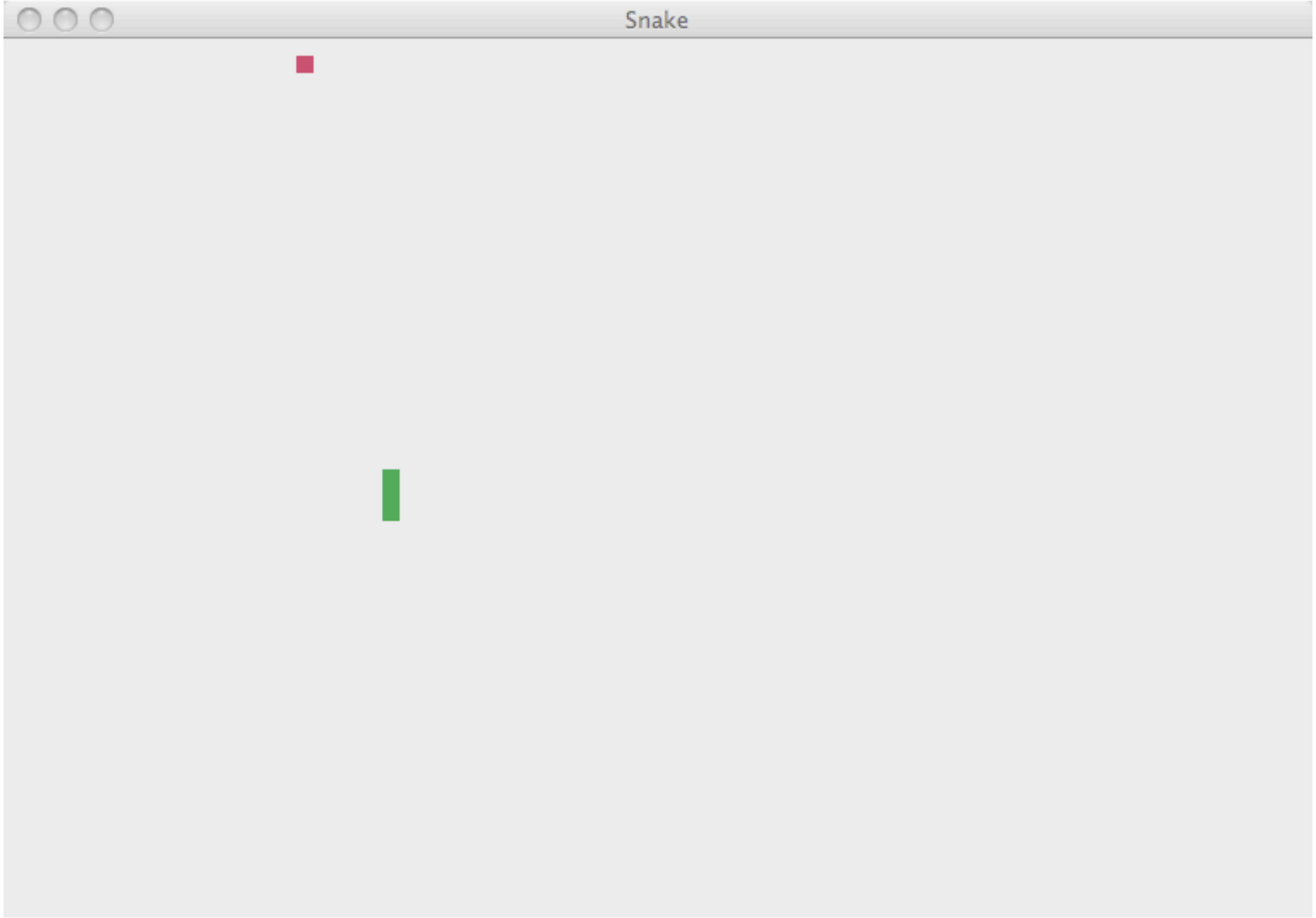
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

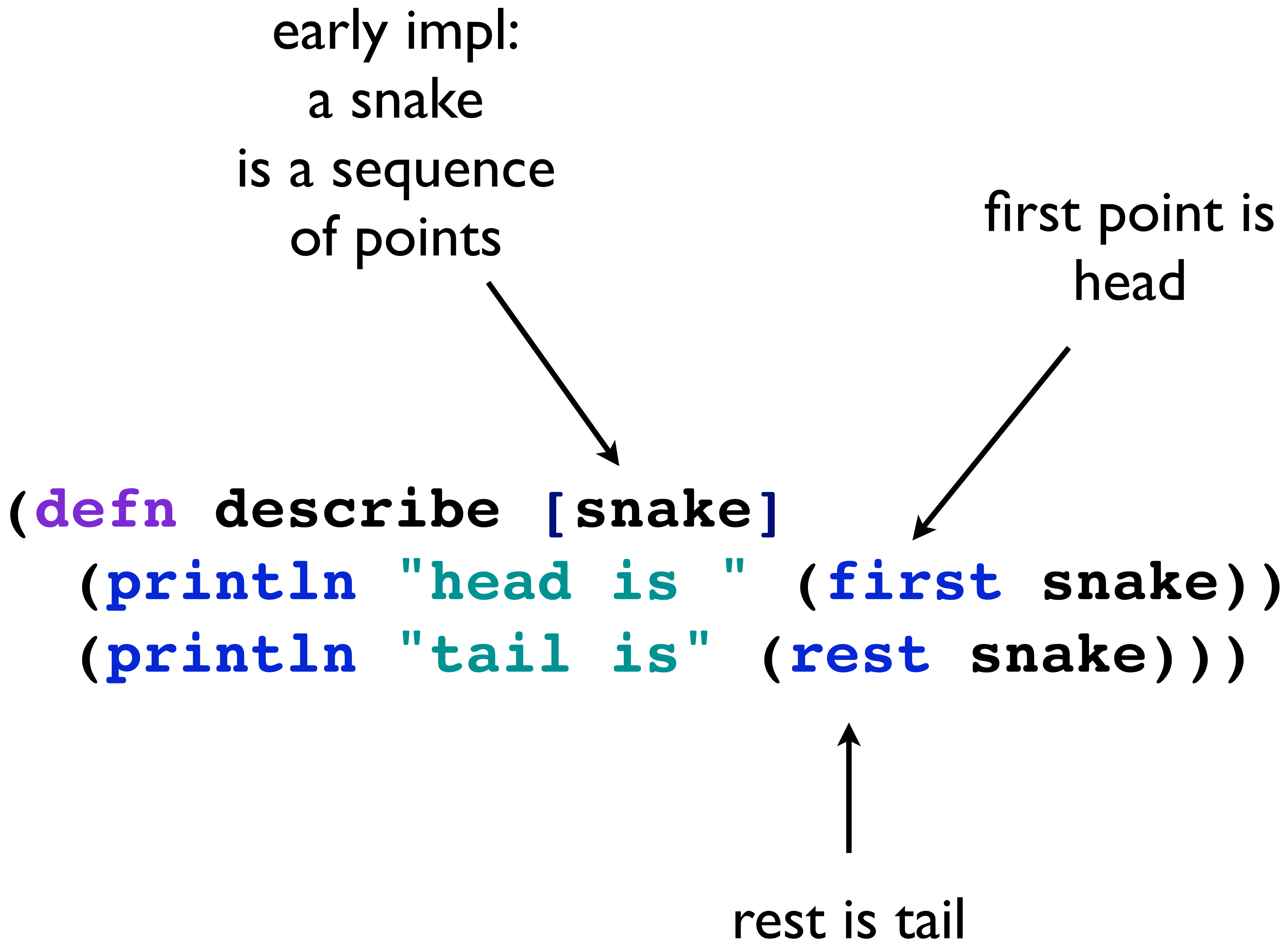
SOFTWARE Development PRODUCTIVITY AWARD 1994

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES clojure is turning the tide in a fifty-year struggle against bloat

game break!



Sample Code:
http://github.com/stuarthalloway/programming-clojure

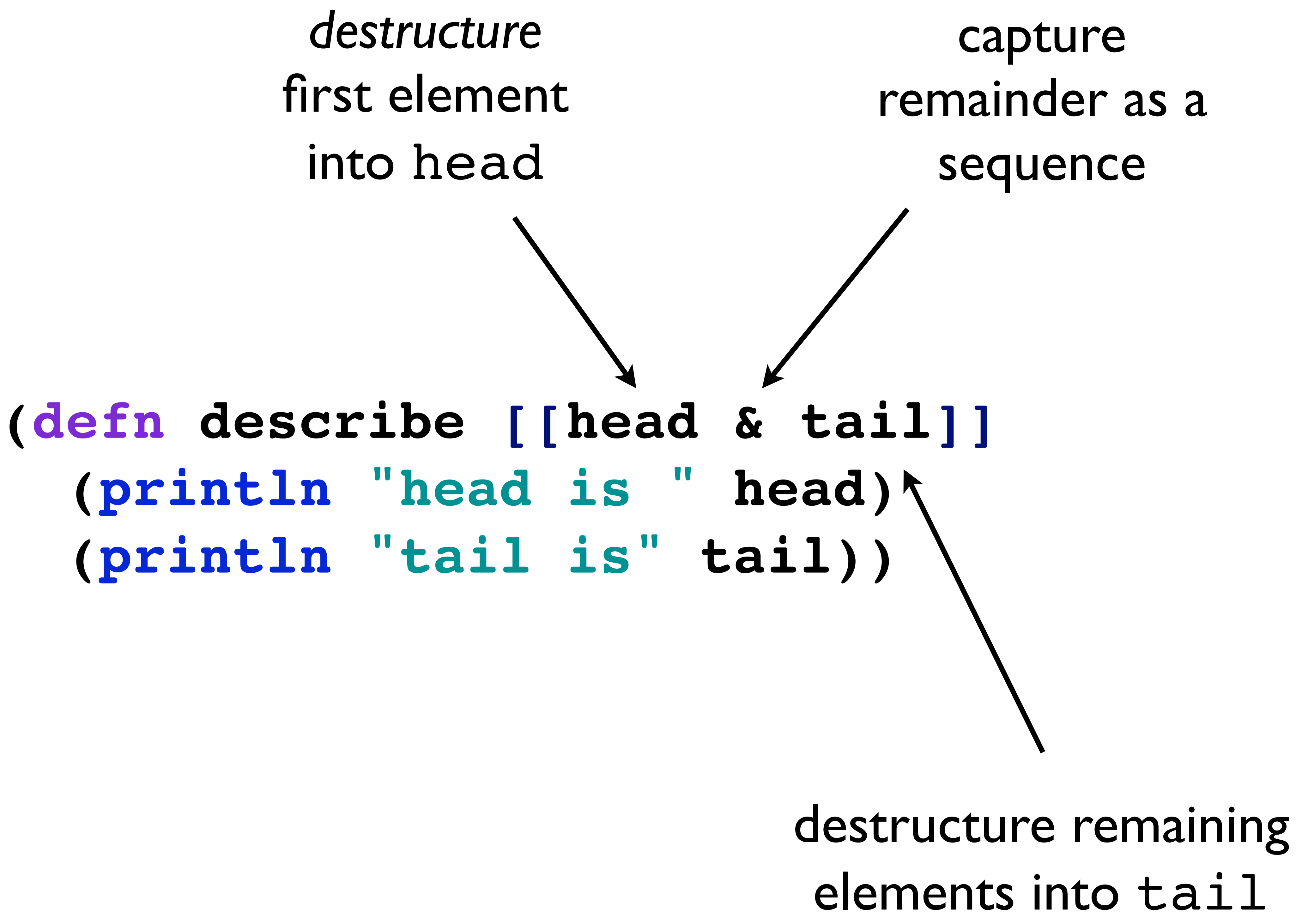


early impl:
a snake
is a sequence
of points

first point is
head

```
(defn describe [snake]
  (println "head is " (first snake))
  (println "tail is" (rest snake)))
```

rest is tail



destructure first element into `head`

capture remainder as a sequence

```clojure
(defn describe [[head & tail]]
  (println "head is " head)
  (println "tail is" tail))
```

destructure remaining elements into `tail`

snake is more than location

```
(defn create-snake []
  {:body (list [1 1])
   :dir [1 0]
   :type :snake
   :color (Color. 15 160 70)})
```

2. nested destructure
to pull head and tail from the
:body value

```
(defn describe [{[head & tail] :body}]
  (println "head is " head)
  (println "tail is" tail))
```

1. destructure map,
looking up the `:tail`

losing the game

```
(defn lose? [{[head & tail] :body}]
  (includes? tail head))
```

3. functional

data literals

type	properties	example
list	singly-linked, insert at front	`(1 2 3)`
vector	indexed, insert at rear	`[1 2 3]`

map	key/value	`{:a 100`
		` :b 90}`

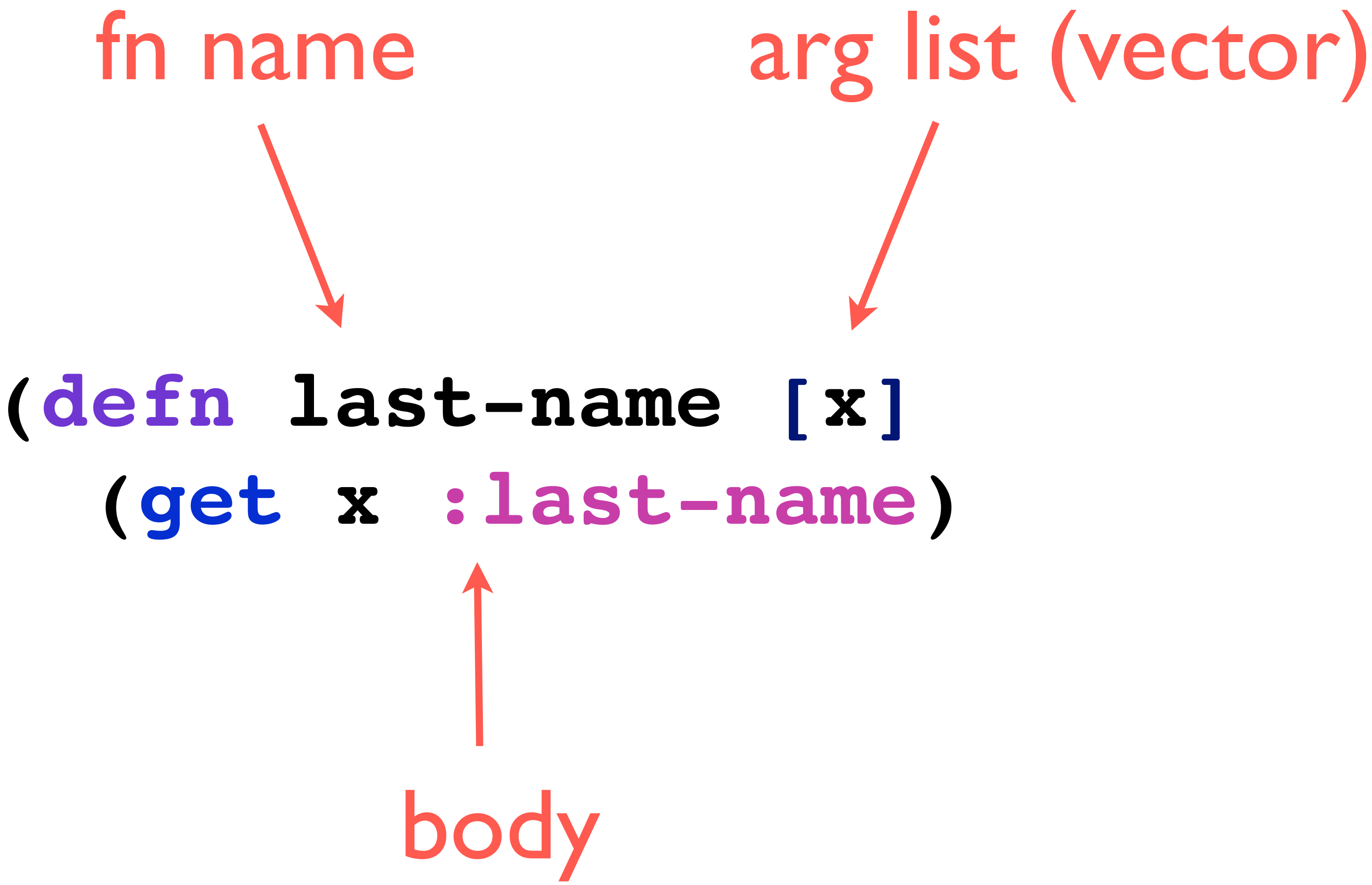
set	key	`#{:a :b}`

higher-order functions

some data

```
lunch-companions
-> ({:fname "Neal", :lname "Ford"}
    {:fname "Stu", :lname "Halloway"}
    {:fname "Dan", :lname "North"})
```

“getter” function



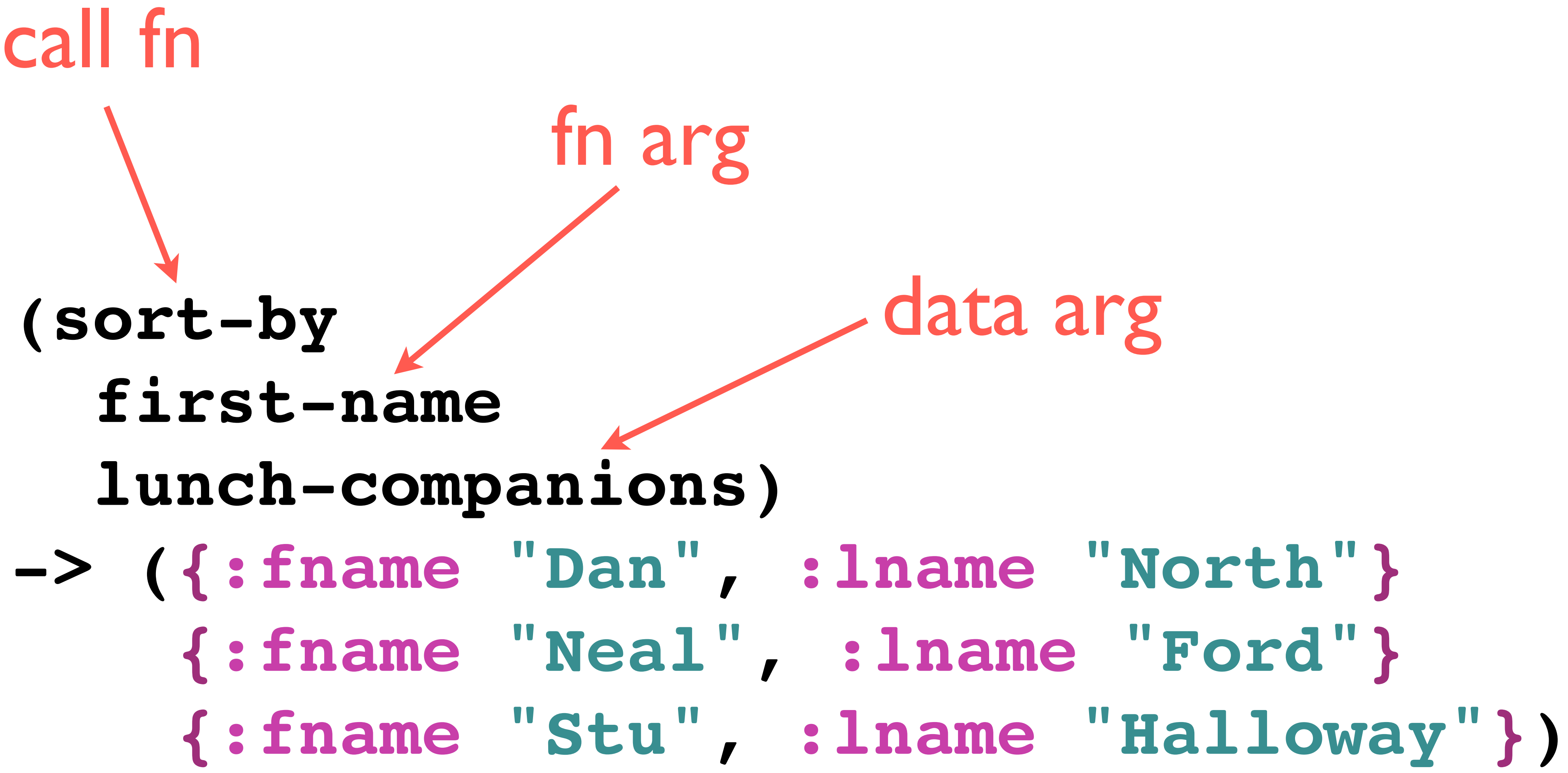
fn name

arg list (vector)

```
(defn last-name [x]
  (get x :last-name))
```

body

pass fn to fn



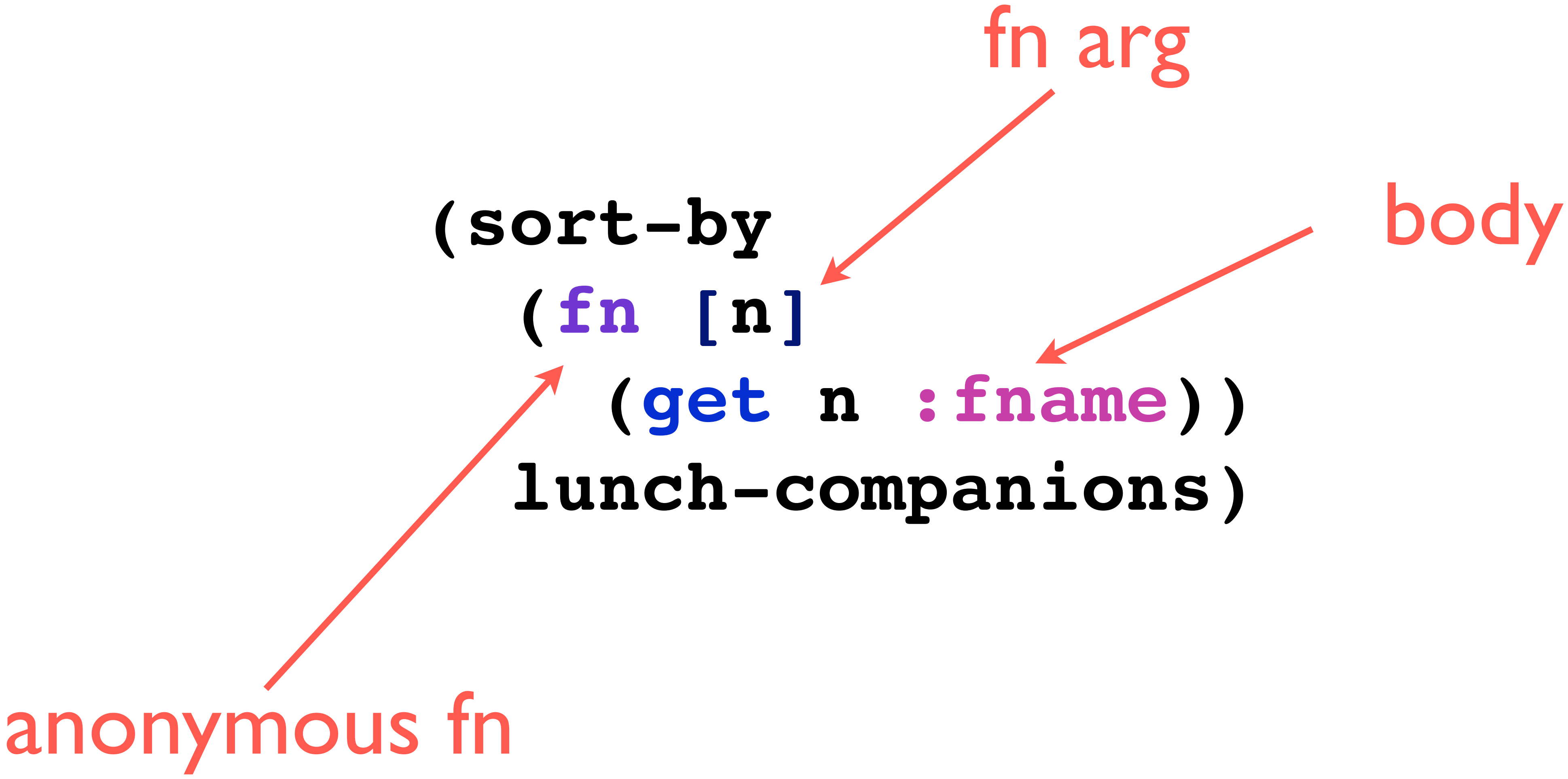
call fn

fn arg

data arg

```
(sort-by
  first-name
  lunch-companions)
-> ({:fname "Dan", :lname "North"}
    {:fname "Neal", :lname "Ford"}
    {:fname "Stu", :lname "Halloway"})
```

anonymous fn



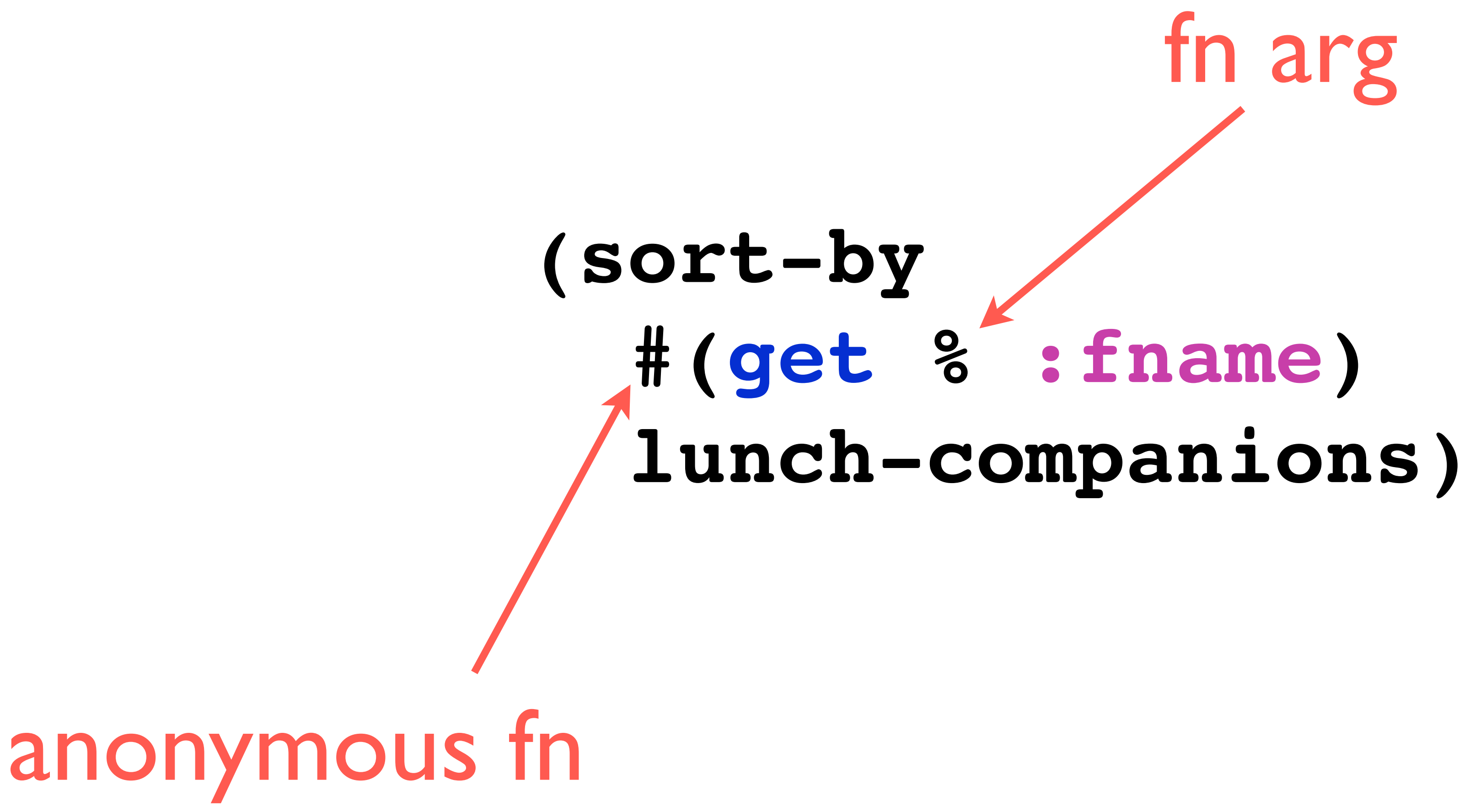
fn arg

body

```
(sort-by
  (fn [n]
    (get n :fname))
  lunch-companions)
```

anonymous fn

anonymous #()



```
(sort-by
  #(get %  :fname)
  lunch-companions)
```

fn arg

anonymous fn

maps are functions

map is fn!

```
(sort-by
   #(% :fname)
   lunch-companions)
```

keywords are functions

keyword
is fn!

```
(sort-by
  #(:fname %)
  lunch-companions)
```

beautiful

```
(sort-by :fname lunch-companions)
```

real languages give a 1-1 ratio of pseudocode/code

persistent data structures

persistent data structures

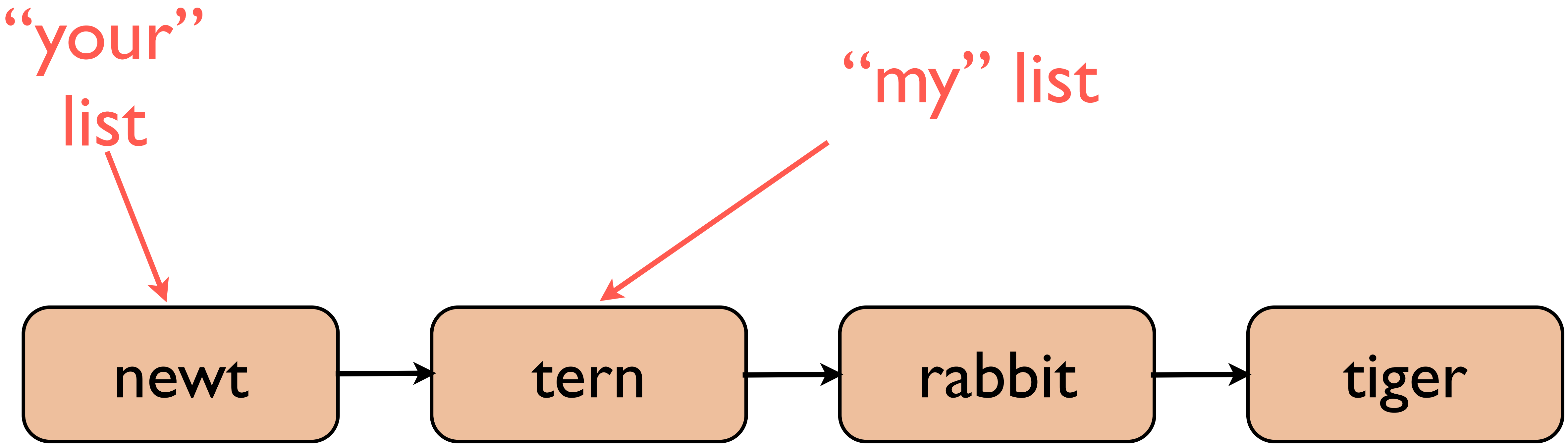
immutable

“change” by function application

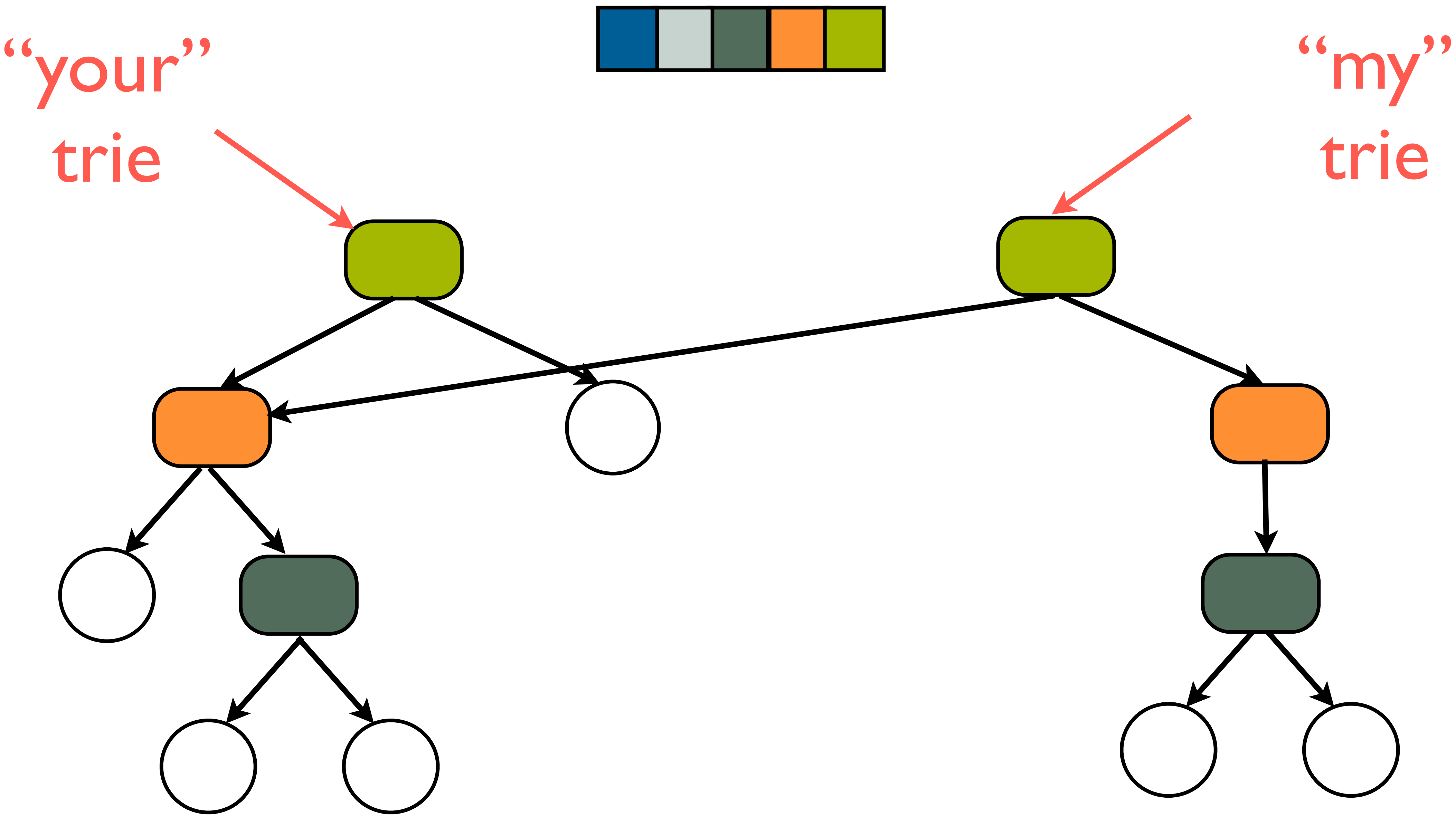
maintain performance guarantees

full-fidelity old versions

persistent example: linked list

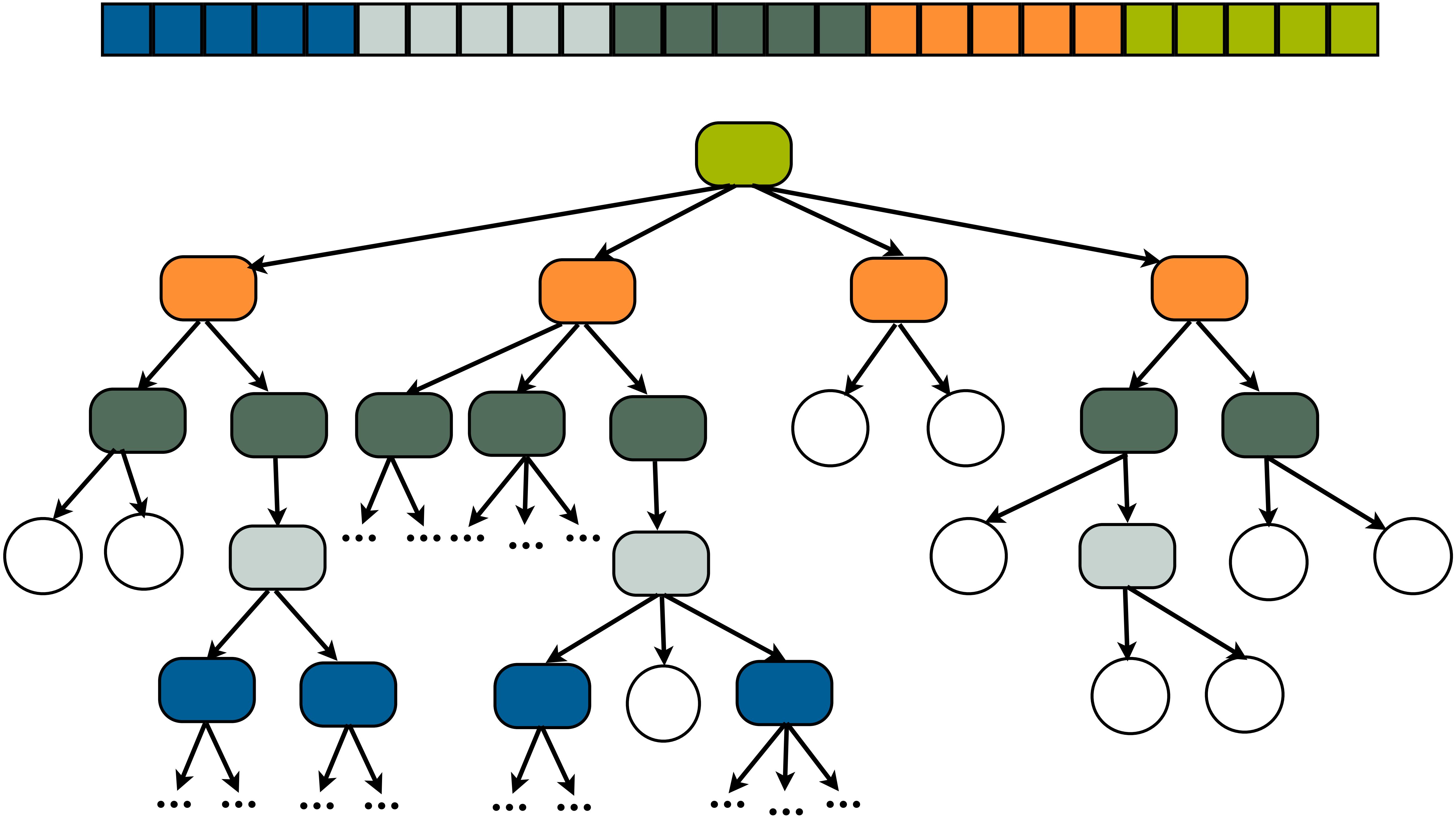


bit-partitioned tries



log2 n:
too slow!

32-way tries



clojure: ‘cause log32 n is fast enough!

sequence library

first / rest / cons

```
(first [1 2 3])
-> 1

(rest [1 2 3])
-> (2 3)

(cons "hello" [1 2 3])
-> ("hello" 1 2 3)
```

take / drop

```
(take 2 [1 2 3 4 5])
-> (1 2)

(drop 2 [1 2 3 4 5])
-> (3 4 5)
```

map / filter / reduce

```
(range 10)
-> (0 1 2 3 4 5 6 7 8 9)

(filter odd? (range 10))
-> (1 3 5 7 9)

(map odd? (range 10))
-> (false true false true false true
false true false true)

(reduce + (range 10))
-> 45
```

sort

```
(sort [ 1 56 2 23 45 34 6 43])
-> (1 2 6 23 34 43 45 56)

(sort > [ 1 56 2 23 45 34 6 43])
-> (56 45 43 34 23 6 2 1)

(sort-by #(.length %)
  ["the" "quick" "brown" "fox"])
-> ("the" "fox" "quick" "brown")
```

conj / into

```
(conj '(1 2 3) :a)
-> (:a 1 2 3)

(into '(1 2 3) '(:a :b :c))
-> (:c :b :a 1 2 3)

(conj [1 2 3] :a)
-> [1 2 3 :a]

(into [1 2 3] [:a :b :c])
-> [1 2 3 :a :b :c]
```

lazy, infinite sequences

```
(set! *print-length* 5)
-> 5

(iterate inc 0)
-> (0 1 2 3 4 ...)

(cycle [1 2])
-> (1 2 1 2 1 ...)

(repeat :d)
-> (:d :d :d :d :d ...)
```

interpose

```
(interpose \, ["list" "of" "words"])
-> ("list" \, "of" \, "words")

(apply str
  (interpose \, ["list" "of" "words"]))
-> "list,of,words"

(use 'clojure.contrib.str-utils)
(str-join \, ["list" "of" "words"]))
-> "list,of,words"
```

predicates

```
(every? odd? [1 3 5])
-> true

(not-every? even? [2 3 4])
-> true

(not-any? zero? [1 2 3])
-> true

(some nil? [1 nil 2])
-> true
```

nested ops

```
(def jdoe {:name "John Doe",
           :address {:zip 27705, ...}})

(get-in jdoe [:address :zip])
-> 27705

(assoc-in jdoe [:address :zip] 27514)
-> {:name "John Doe", :address {:zip 27514}}

(update-in jdoe [:address :zip] inc)
-> {:name "John Doe", :address {:zip 27706}}
```

Ash zna durbatulûk,
ash zna gimbatul,
ash zna thrakatulûk
agh burzum-ishi
krimpatul.

where are we?

1. java interop
2. lisp
3. functional

does it work?

example:
refactor apache
commons
indexOfAny

indexOfAny behavior

```
StringUtils.indexOfAny(null, *)                = -1
StringUtils.indexOfAny("", *)                  = -1
StringUtils.indexOfAny(*, null)                = -1
StringUtils.indexOfAny(*, [])                  = -1
StringUtils.indexOfAny("zzabyycdxx",['z','a']) = 0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) = 3
StringUtils.indexOfAny("aba", ['z'])           = -1
```

indexOfAny impl

```java
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars)
{
  if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
    return -1;
  }
  for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    for (int j = 0; j < searchChars.length; j++) {
      if (searchChars[j] == ch) {
        return i;
      }
    }
  }
  return -1;
}
```

simplify corner cases

```
public static int indexOfAny(String str, char[] searchChars)
{
  when (searchChars)
    for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);
      for (int j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

- type decls

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      for (j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

+ when clause

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      when searchChars(ch) i;
    }
  }
}
```

+ comprehension

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for ([i, ch] in indexed(str)) {
      when searchChars(ch) i;
    }
  }
}
```

lispify!

```
(defn index-filter [pred coll]
  (when pred
    (for [[idx elt] (indexed coll) :when (pred elt)] idx)))
```

functional

is

simpler

	imperative	**functional**
functions	1	1
classes	1	0
internal exit points	2	0
variables	3	0
branches	4	0
boolean ops	1	0
function calls*	6	3
total	***18***	***4***

functional

is

more general!

reusing index-filter

```
; idxs of heads in stream of coin flips
(index-filter #{:h}
[:t :t :h :t :h :t :t :t :h :h])
-> (2 4 8 9)

; Fibonaccis pass 1000 at n=17
(first
  (index-filter #(> % 1000) (fibo)))
-> 17
```

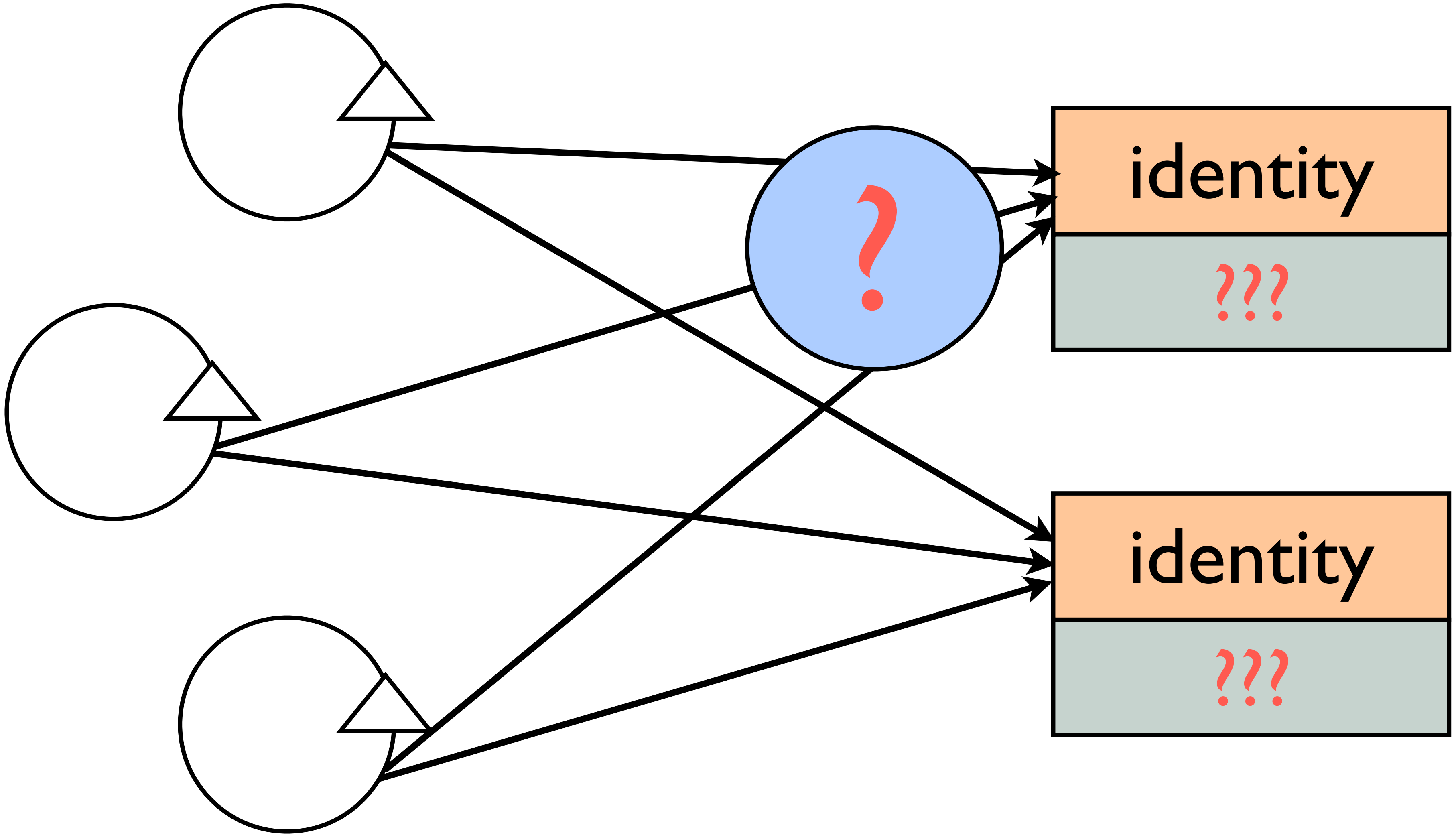
imperative	functional
searches strings	searches *any sequence*
matches characters	matches *any predicate*
returns first match	returns *lazy seq of all matches*

fp reduces incidental complexity by an order of magnitude

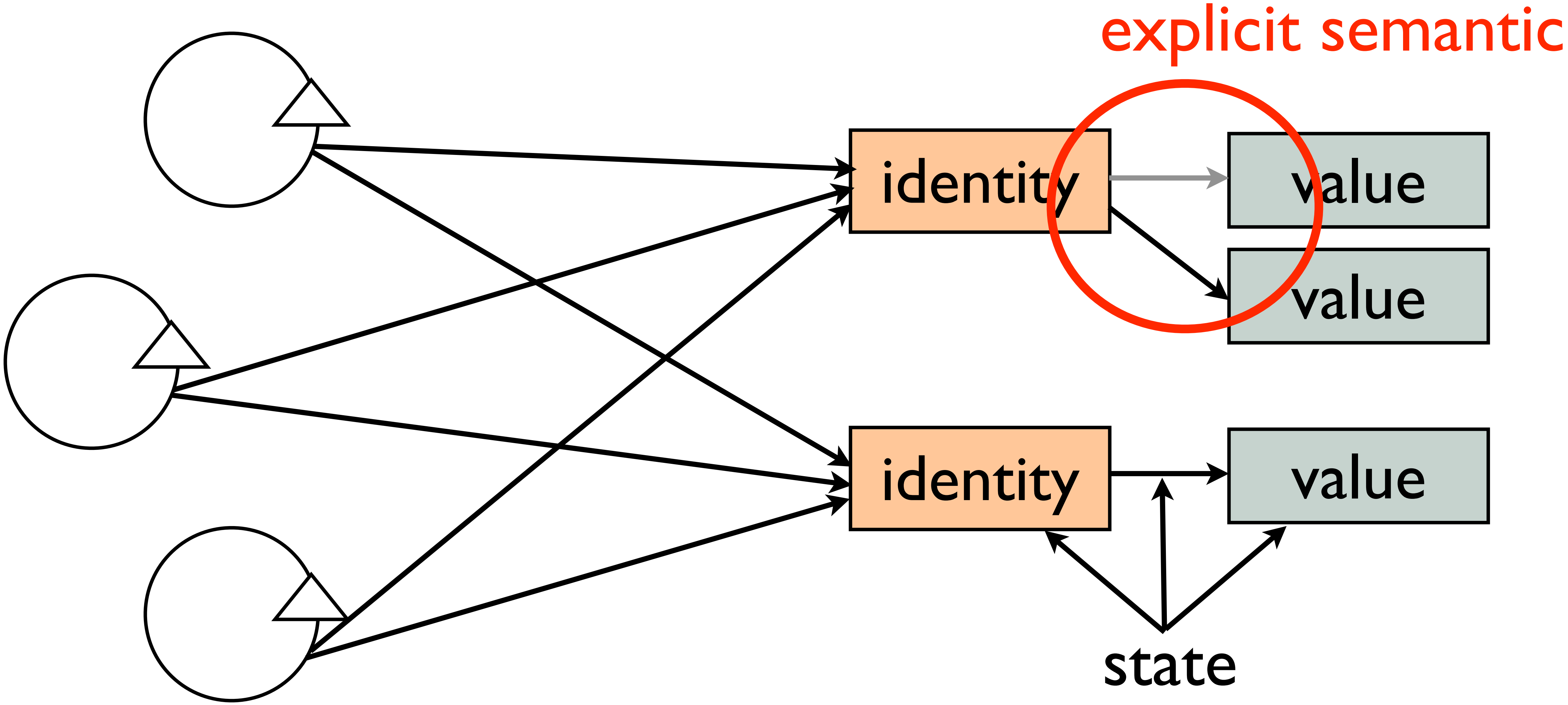
4. concurrency

4. state ~~concurrency~~

oo is incoherent



clojure


explicit semantic
identity
value
value
identity
value
state

terms

1. **value:** immutable data in a persistent data structure

2. **identity:** series of causally related values over time

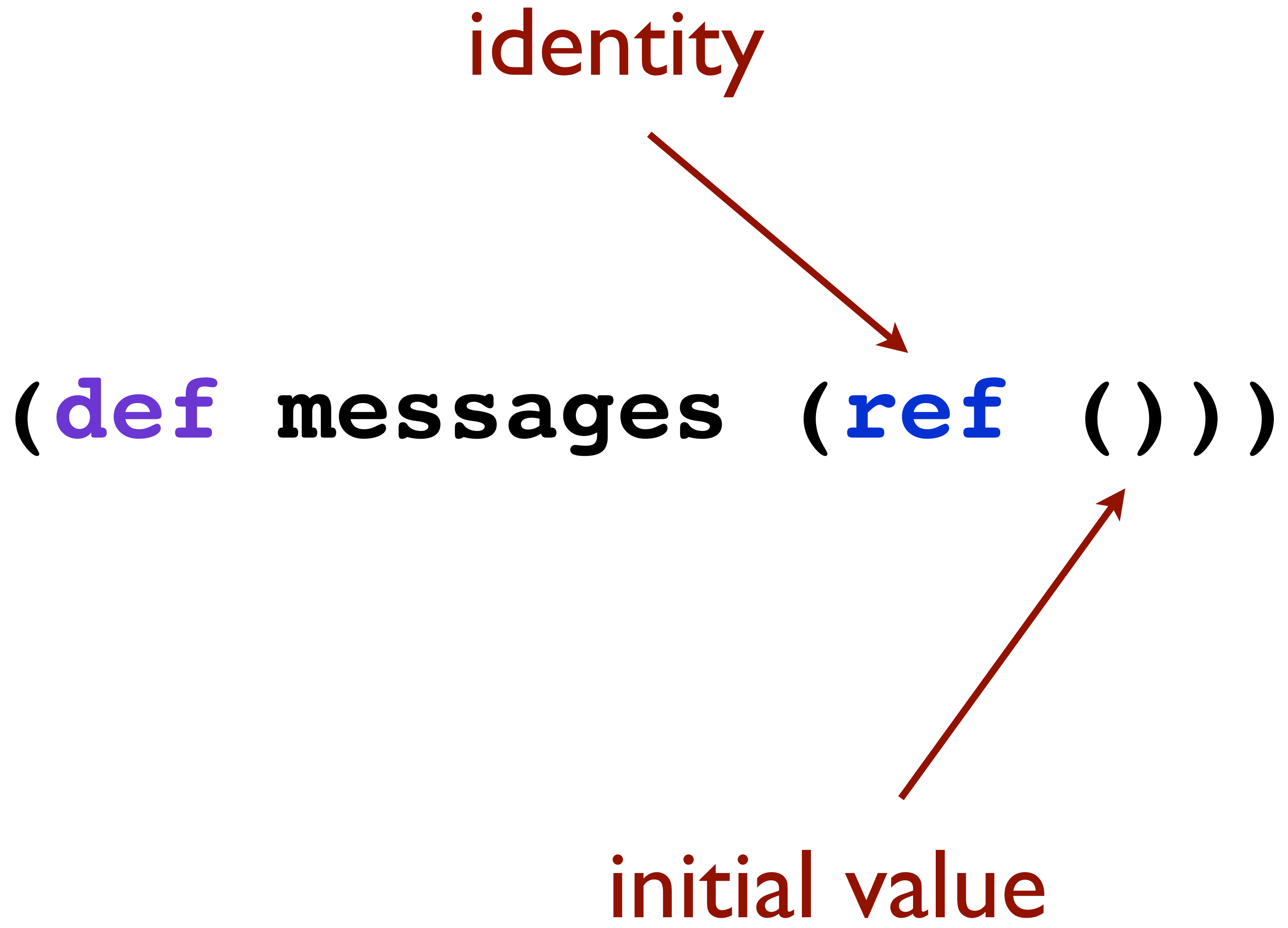
3. **state:** identity at a point in time

identity types (references)

	shared	isolated
synchronous/ coordinated	**refs/stm**	-
synchronous/ autonomous	**atoms**	**vars**
asynchronous/ autonomous	**agents**	-

identity 1:
refs and stm

ref example: chat



identity

```
(def messages (ref ()))
```

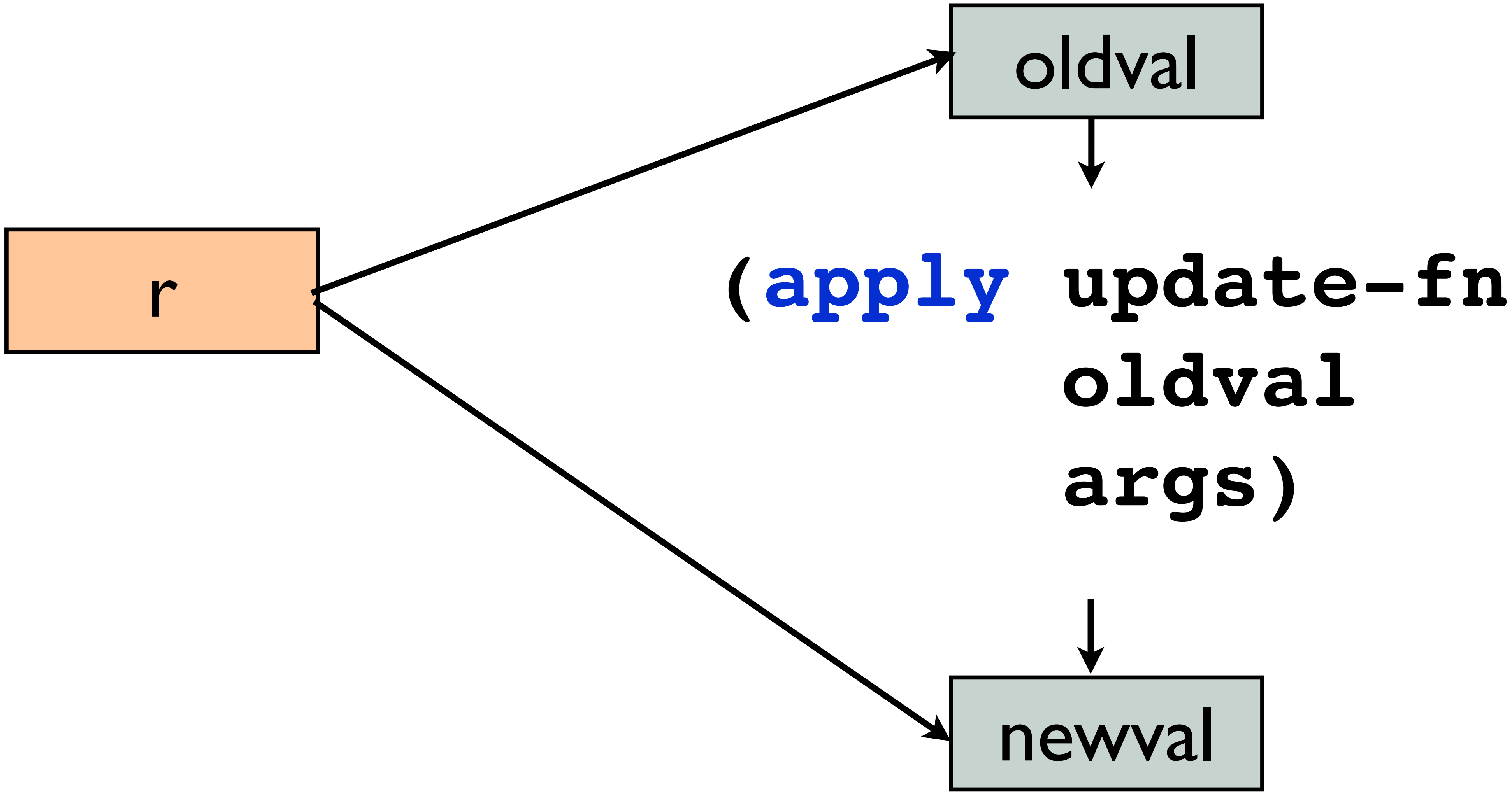
initial value

reading value

```
(deref messages)
=> ()

@messages
=> ()
```

alter

```
(alter r update-fn & args)
```



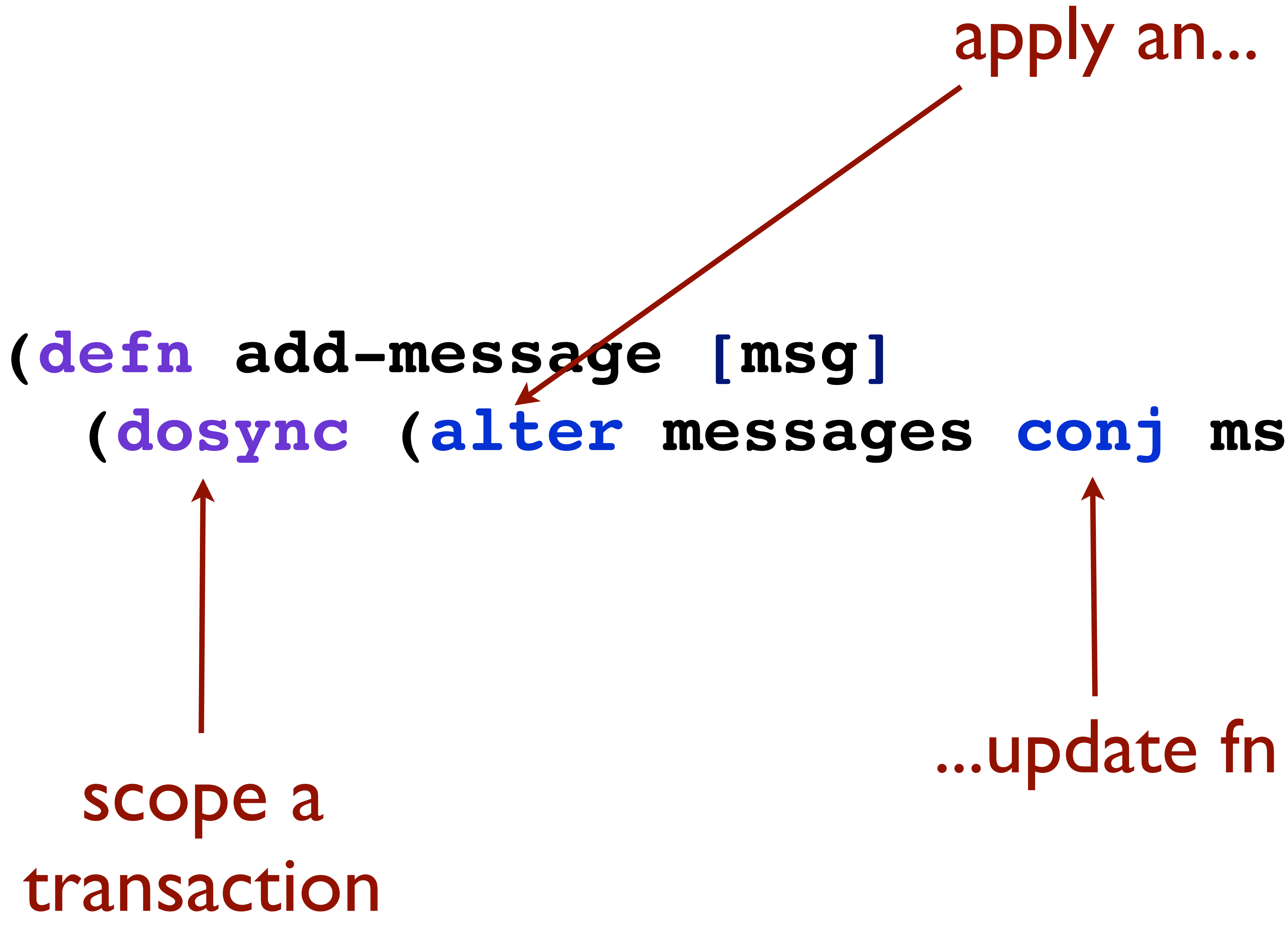
oldval

r

```
(apply update-fn
       oldval
       args)
```

newval

updating



apply an...

```
(defn add-message [msg]
  (dosync (alter messages conj msg)))
```

scope a transaction

...update fn

unified update model

update by function application

readers require no coordination

readers never block anybody

writers never block readers

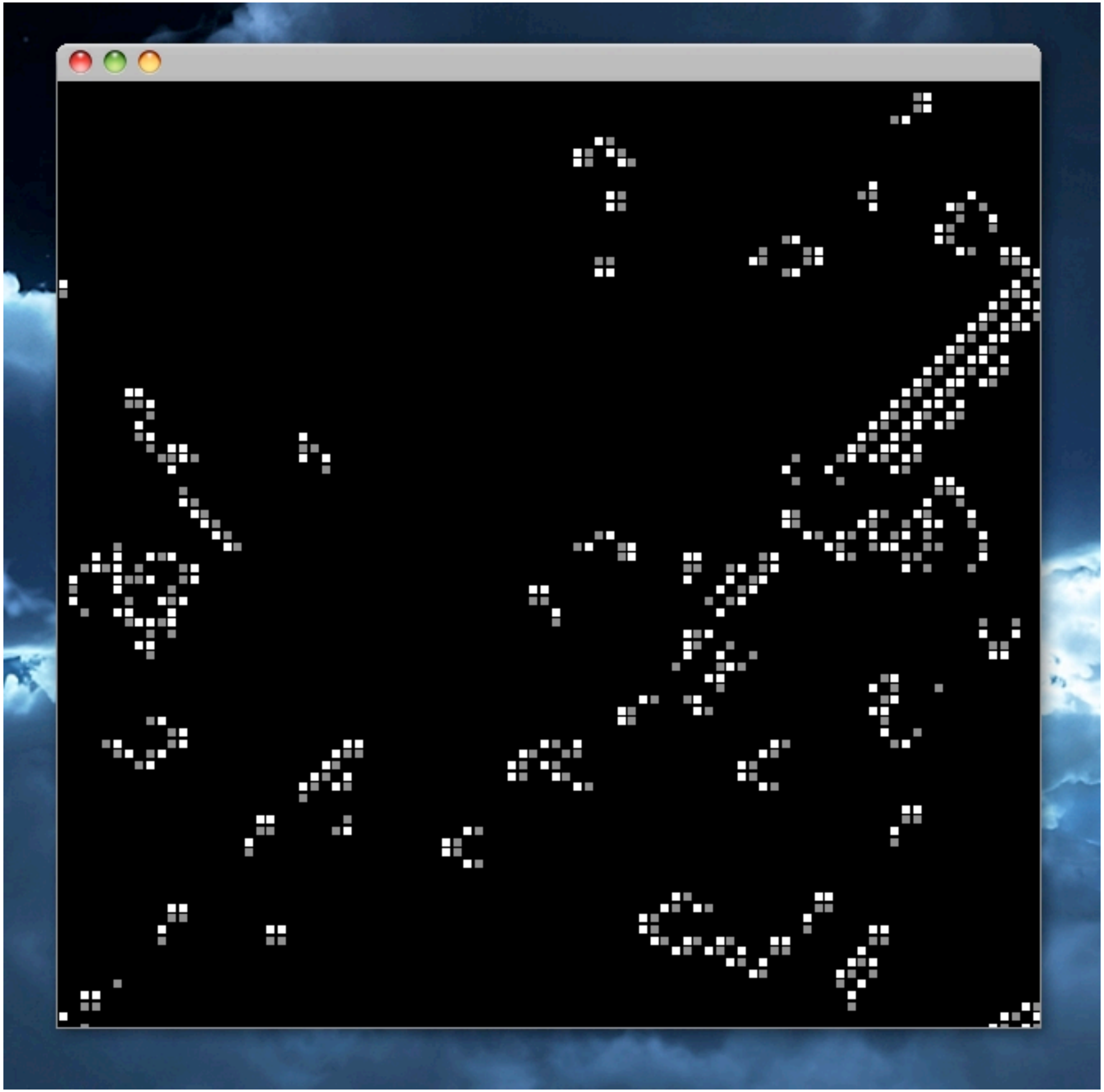
a sane approach
to local state
permits coordination,
but does not require it

unified update model

	ref	atom	agent	var
create	ref	atom	agent	def
deref	deref/@	deref/@	deref/@	deref/@
update	alter	swap!	send	alter-var-root

identity 2:
atoms

http://blog.bestinclass.dk/index.php/2009/10/brians-functional-brain/



board is just a value

```clojure
(defn new-board
  "Create a new board with about half the cells set
   to :on."
  ([] (apply new-board dim-board))
  ([dim-x dim-y]
     (for [x (range dim-x)]
       (for [y (range dim-y)]
         (if (< 50 (rand-int 100)) :on :off)))))
```

distinct bodies by arity

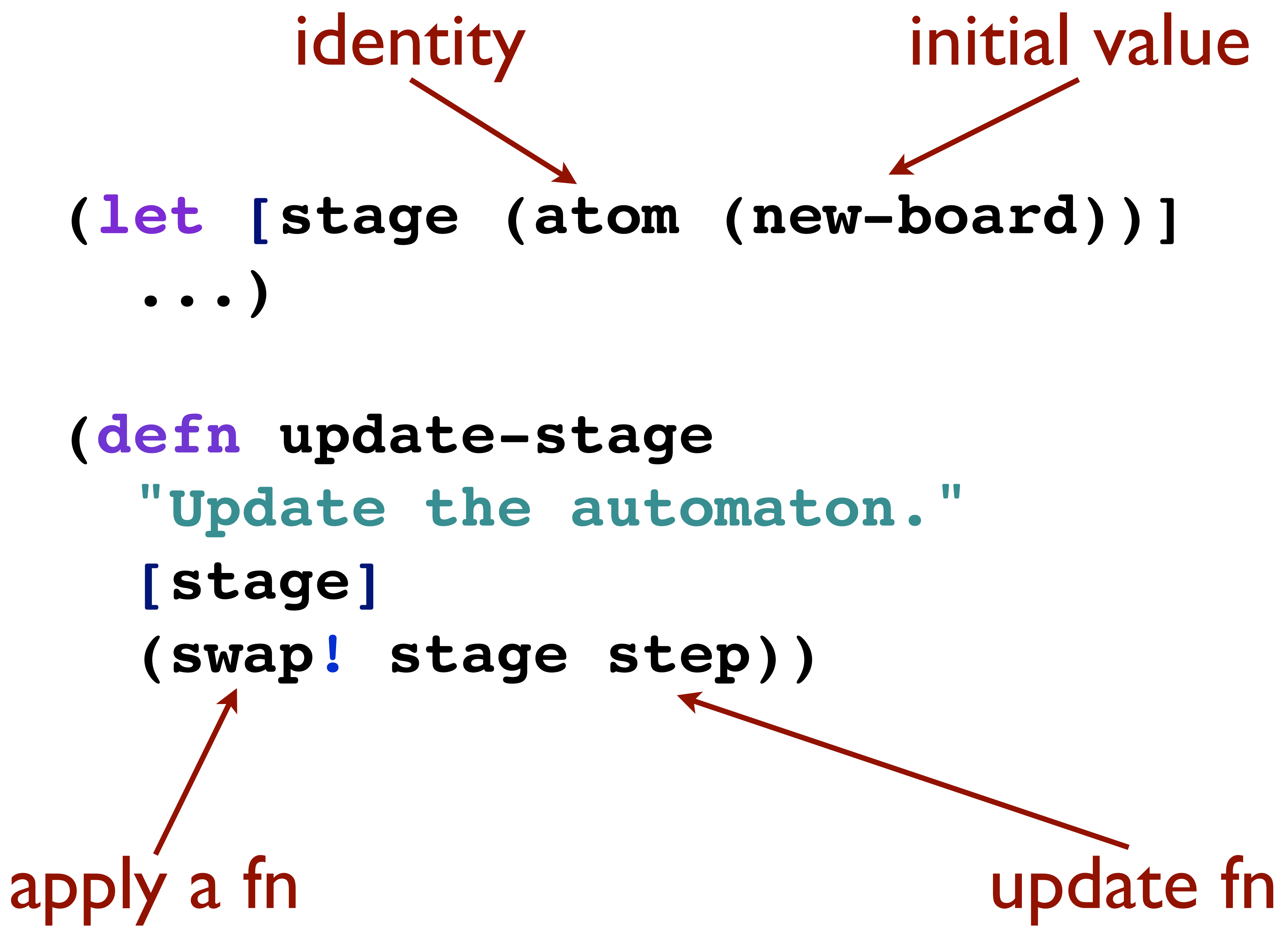
update is just a function

rules

```
(defn step
  "Advance the automation by one step, updating all
   cells."
  [board]
  (doall
   (map (fn [window]
          (apply #(doall (apply map rules %&))
                 (doall (map torus-window window))))
        (torus-window board))))
```

cursor over previous, me, next

state is trivial



identity

initial value

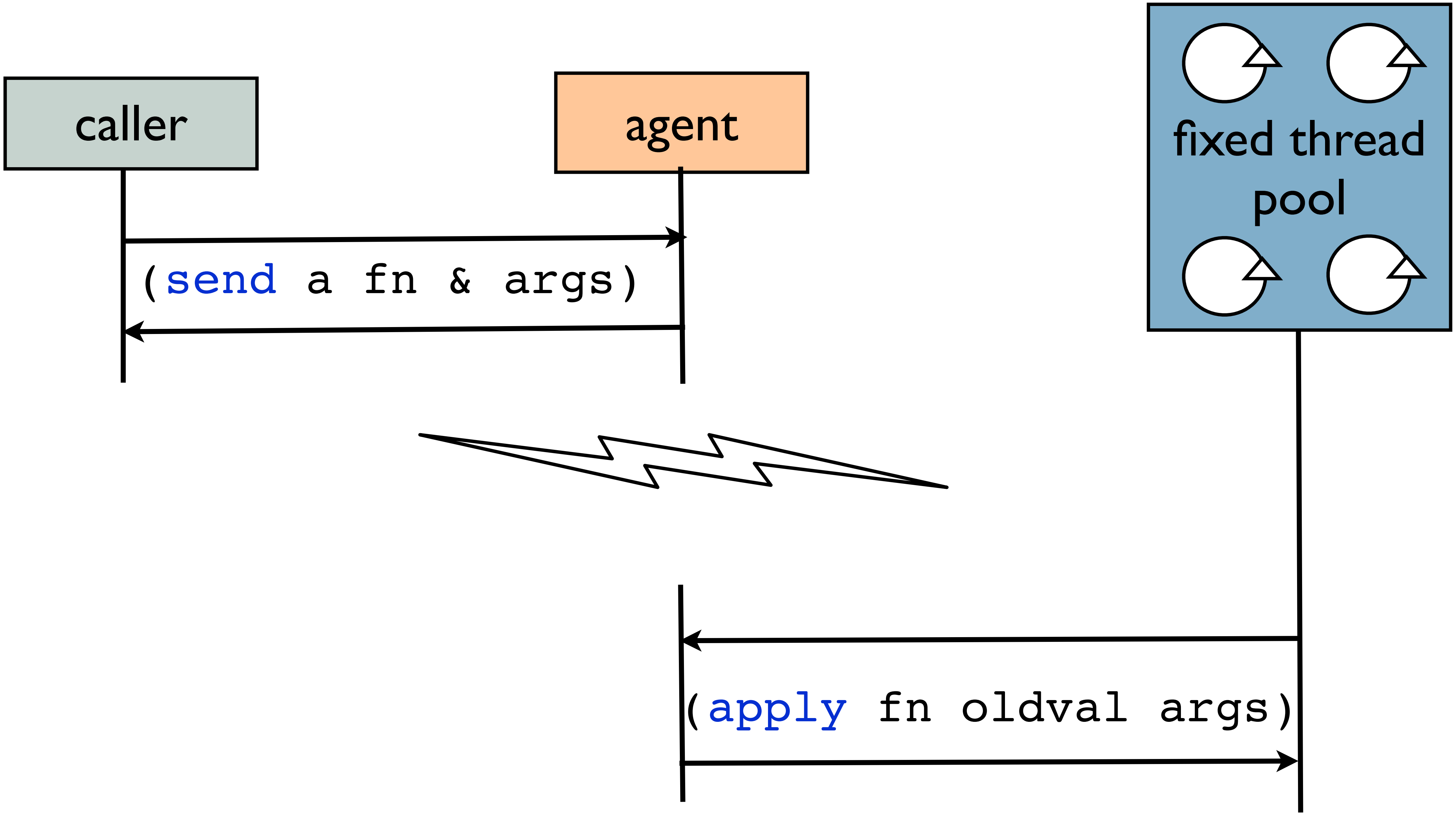
```
(let [stage (atom (new-board))]
  ...)

(defn update-stage
  "Update the automaton."
  [stage]
  (swap! stage step))
```

apply a fn

update fn

identity 3: agents

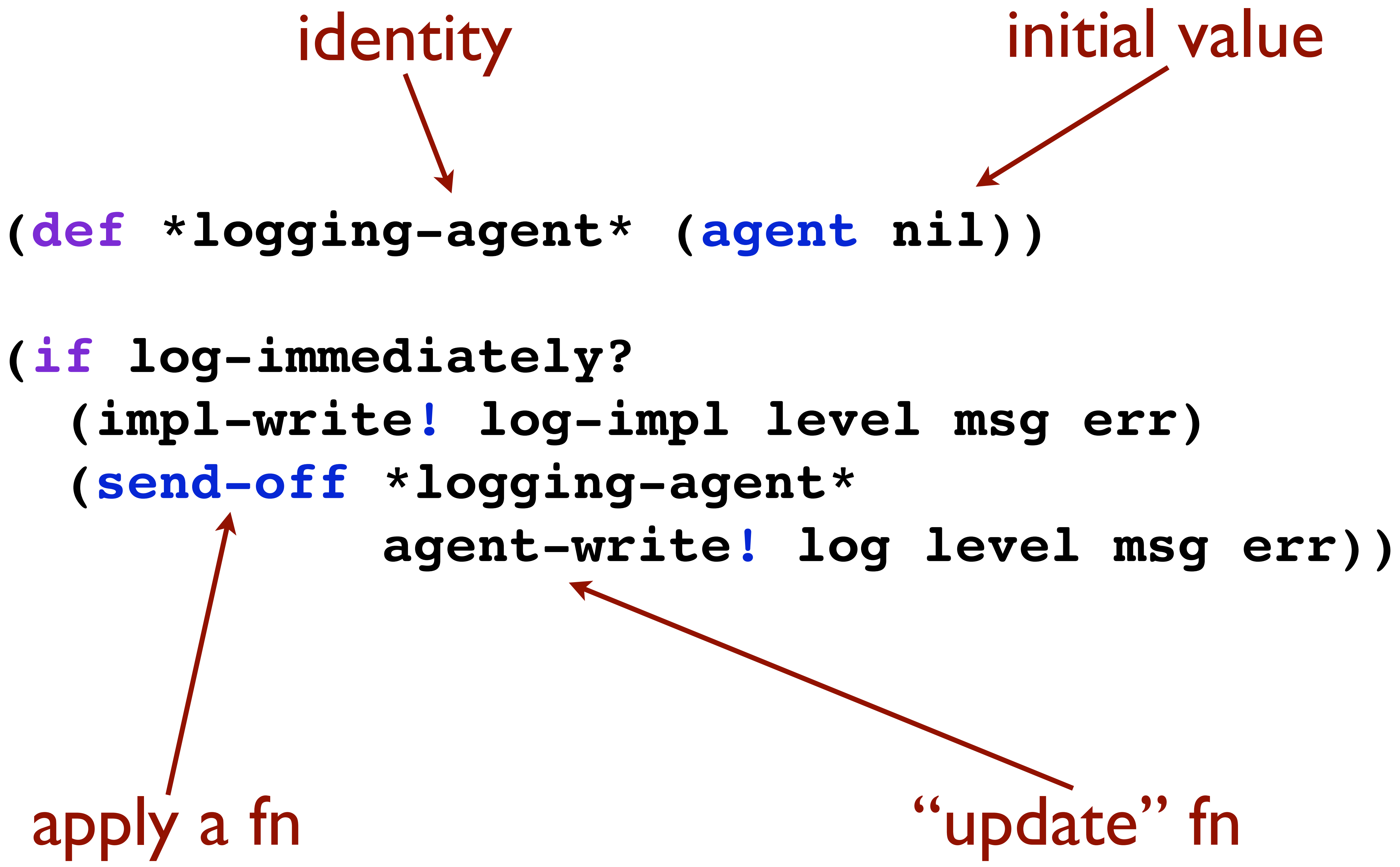
send



```
(send a fn & args)
```

```
(apply fn oldval args)
```

state is trivial



identity

initial value

```
(def *logging-agent* (agent nil))

(if log-immediately?
  (impl-write! log-impl level msg err)
  (send-off *logging-agent*
            agent-write! log level msg err))
```

apply a fn

“update” fn

identity 4: vars

def forms create vars

```
(def greeting "hello")

(defn make-greeting [n]
  (str "hello, " n))
```

vars can be rebound

api	scope
`alter-var-root`	root binding
`set!`	thread-local, permanent
`binding`	thread-local, dynamic

system settings

```
(set! *print-length* 20)
=> 20

primes
=> (2 3 5 7 11 13 17 19 23 29 31 37 41
    43 47 53 59 61 67 71 ...)

(set! *print-length* 5)
=> 5

primes
=>(2 3 5 7 11 ...)
```

var	usage
`*in*, *out*, *err*`	standard streams

`*print-length*,`
`*print-depth*`
	structure printing

`*warn-on-reflection*`	performance tuning
`*ns*`	current namespace
`*file*`	file being evaluated
`*command-line-args*`	*guess*

with-... helper macros

```
(def bar 10)
-> #'user/bar

(with-ns 'foo (def bar 20))
-> #'foo/bar

user/bar
-> 10

foo/bar
-> 20
```

bind a var
for a dynamic
scope

other def forms

form	usage
`defonce`	set root binding once
`defvar`	var plus docstring
`defunbound`	no initial binding
`defstruct`	map with slots
`defalias`	same metadata as original
`defhinted`	infer type from initial binding
`defmemo`	defn + memoize

many of these are in clojure.contrib.def...

identity:
more options

use commute
when update
can happen
anytime

not safe for commute

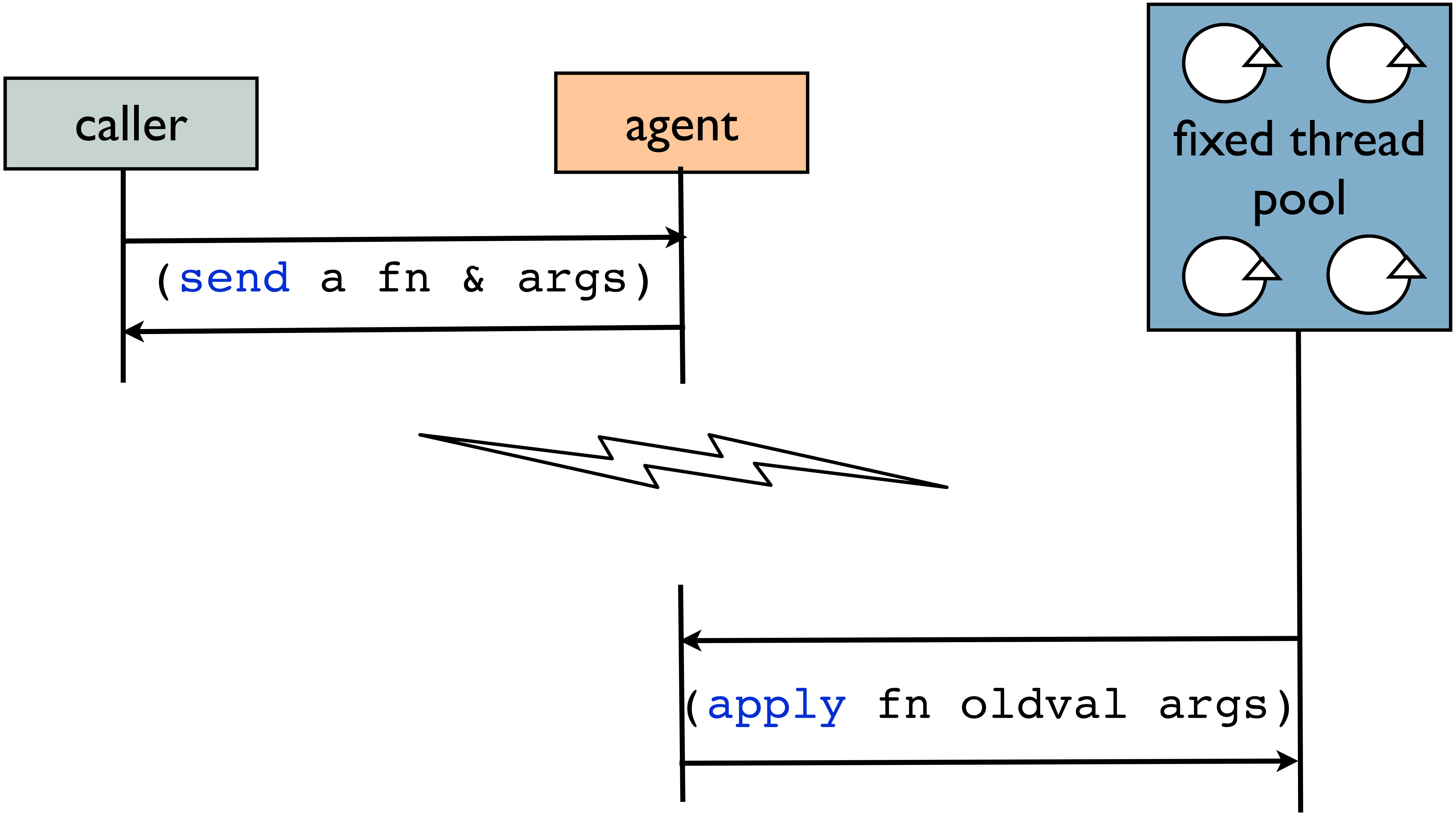
```
(defn next-id
  "Get the next available id."
  []
  (dosync
   (alter ids inc)))
```

safe!

```
(defn increment-counter
  "Bump the internal count."
  []
  (dosync
   (alter ids inc))
  nil)
```

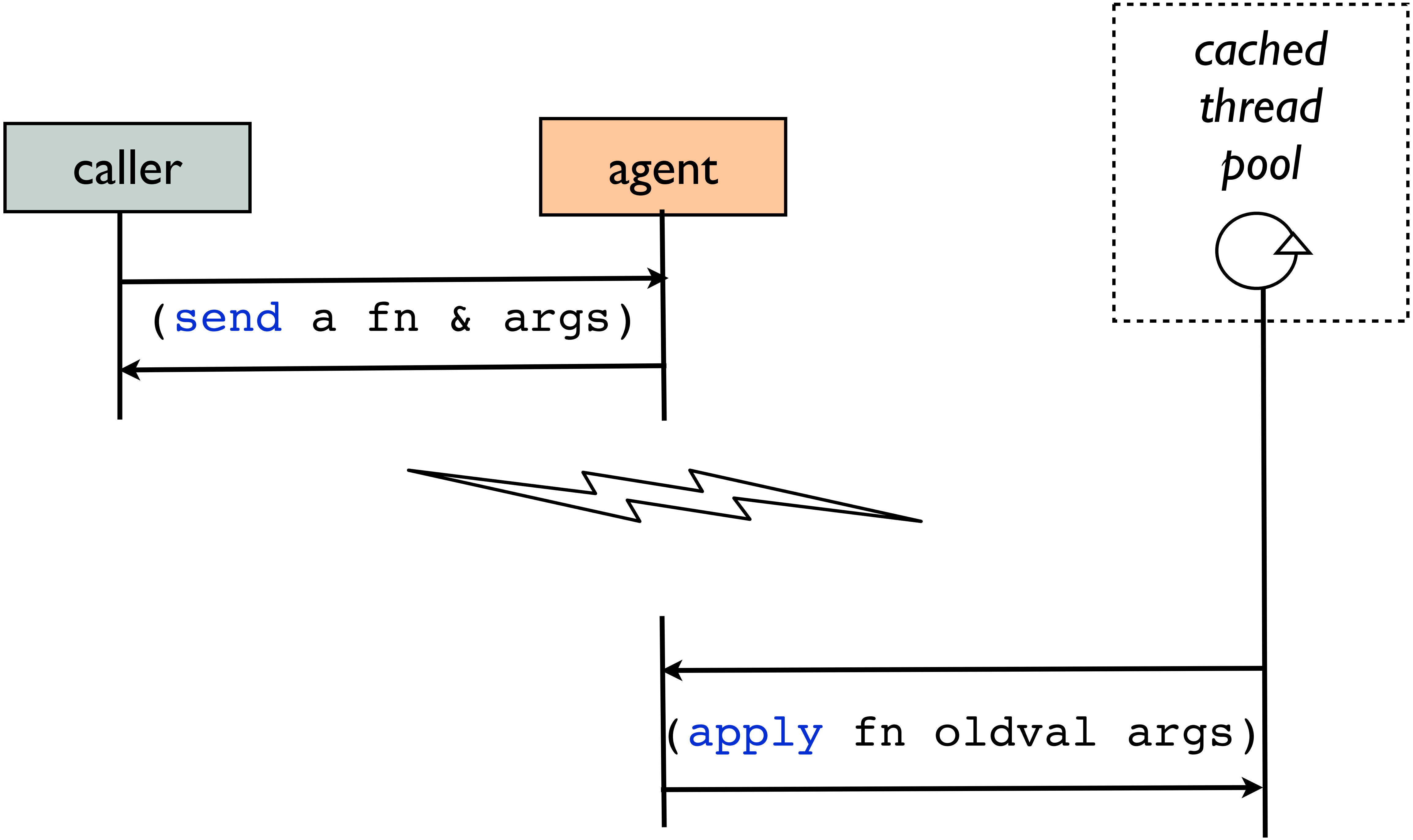
prefer send-off
if agent ops
might block

send



caller
agent
fixed thread pool
(send a fn & args)
(apply fn oldval args)

send-off



`(send a fn & args)`

`(apply fn oldval args)`

use ref-set to set initial/base state

unified update, revisited

update mechanism	ref	atom	agent
pure function application	`alter`	`swap!`	`send`
pure function (commutative)	`commute`	-	-
pure function (blocking)	-	-	`send-off`
setter	`ref-set`	`reset!`	-

send-off
to *agent*
for background
iteration

monte carlo via ongoing agent

```
(defn background-pi [iter-count]
  (let
    [agt (agent {:in-circle 0 :total 0})
     continue (atom true)
     iter (fn sim [a-val]
            (when continue (send-off *agent* sim))
            (run-simulation a-val iter-count))]
    (send-off agt iter)
    {:guesser agt :continue atom}))
```

queue more work

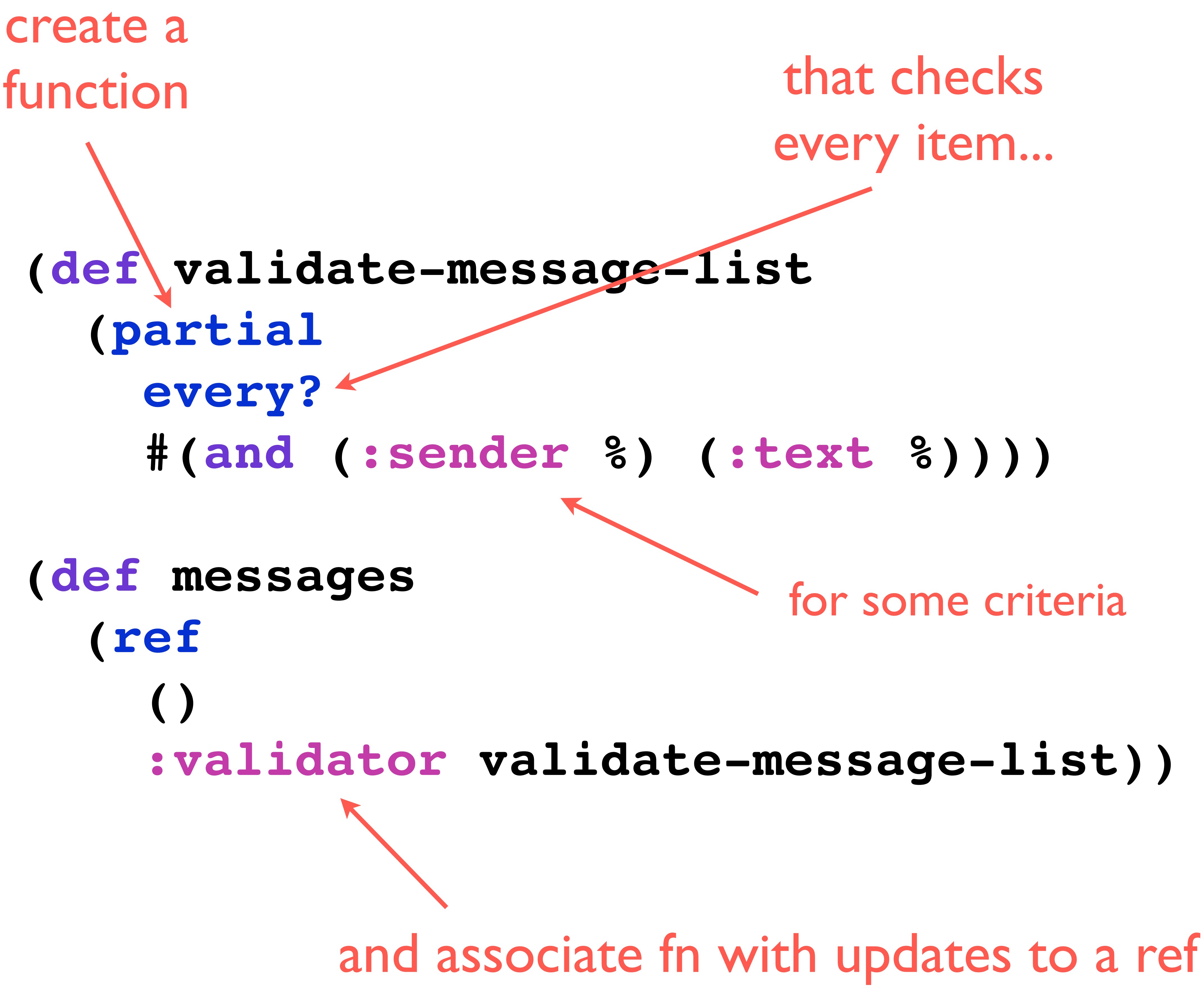
do the work

escape hatch

(not= agents actors)

agents	actors
in-process only	oop
no copying	copying
no deadlock	can deadlock
no coordination	can coordinate

validation



create a function

that checks every item...

```
(def validate-message-list
  (partial
    every?
    #(and (:sender %) (:text %))))

(def messages
  (ref
    ()
    :validator validate-message-list))
```

for some criteria

and associate fn with updates to a ref

agent error handling

```
(def counter (agent 0 :validator integer?))
-> #'user/counter

(send counter (constantly :fail))
-> #<Agent 0>

(agent-errors counter)
-> (#<IllegalStateException
     java.lang.IllegalStateException:
     Invalid reference state>)

(clear-agent-errors counter)
-> nil

@counter
-> 0
```

will fail soon

list of errors

reset and move on

agents
and
transactions

tying agent to a tx

```
(defn add-message-with-backup [msg]
  (dosync
   (let [snapshot (alter messages conj msg)]
     (send-off backup-agent (fn [filename]
                    (spit filename snapshot)
                    filename))
     snapshot)))
```

exactly once if tx succeeds

where are we?

1. java interop
2. lisp
3. functional
4. value/identity/state

does it work?

a workable approach to state

good values: persistent data structures

good identities: references

mostly functional?

usable by mortals?

mostly functional?

1 line in 1000 creates a reference

project	loc	calls to ref	calls to agent	calls to atom
clojure	7232	3	1	2
clojure-contrib	17032	22	2	12
compojure	1966	1	0	0
incanter	6248	1	0	0

usable by mortals?

```
; compojure session management
(def memory-sessions (ref {}))

(defmethod read-session :memory
  [repository id]
  (@memory-sessions id))

(defmethod write-session :memory
  [repository session]
  (dosync
    (alter memory-sessions
      assoc (session :id) session)))
```

multimethod dispatch

read

update

cache previous results

```
; from clojure core
(defn memoize [f]
  (let [mem (atom {})]
    (fn [& args]
      (if-let [e (find @mem args)]
        (val e)
        (let [ret (apply f args)]
          (swap! mem assoc args ret)
          ret)))))
```

cache hit

cache miss: call f, add to cache

clojure

values are

- immutable, persistent

identities are

- well-specified, consistent

state is

- mostly functional
- usable by mortals

languages that emphasize immutability are *better at mutation*

time management

prepare to parallelize

```
(defn step
  "Advance the automation by one step, updating all
   cells."
  [board]
  (doall
   (map (fn [window]
          (apply #(doall (apply map rules %&))
                 (doall (map torus-window window))))
        (torus-window board))))
```

done

```
(defn step
  "Advance the automation by one step, updating all
   cells."
  [board]
  (doall
   (pmap (fn [window]
          (apply #(doall (apply map rules %&))
                 (doall (map torus-window window))))
        (torus-window board))))
```

delay

```
(def e (delay (expensive-calculation)))
-> #'demo.delay/e

(delay? e)
-> true

(force e)
-> :some-result

(deref e)
-> :some-result

@e
-> :some-result
```

first call blocks until work completes on **this** thread, later calls hit cache

future

```
(def e1 (future (expensive-calculation)))
-> #'demo.future/e1

(deref e1)
-> :some-result

@e1
-> :some-result
```

first call blocks until work completes on **other** thread, later calls hit cache

cancelling a future

```
(def e2 (future (expensive-calculation)))
-> #'demo.future/e2

(future-cancel e2)
-> true

(future-cancelled? e2)
-> true

(deref e2)
-> java.util.concurrent.CancellationException
```

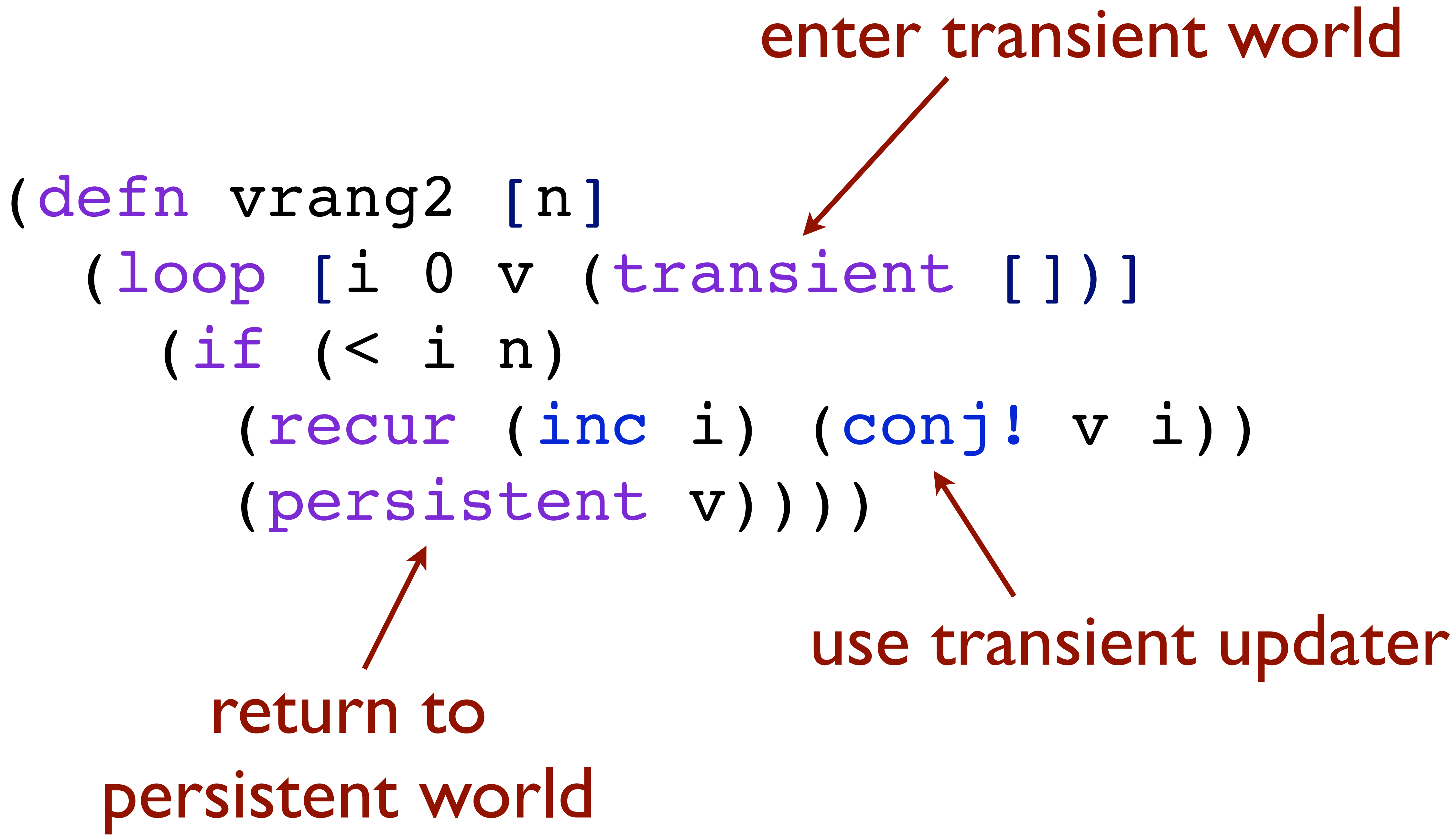
transients

build structure
on one thread,
then release into
the wild

persistent...

```
(defn vrange [n]
 (loop [i 0 v []]
  (if (< i n)
   (recur (inc i) (conj v i))
   v)))
```

...to transient



enter transient world

```
(defn vrang2 [n]
  (loop [i 0 v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent v))))
```

use transient updater

return to
persistent world

fast!

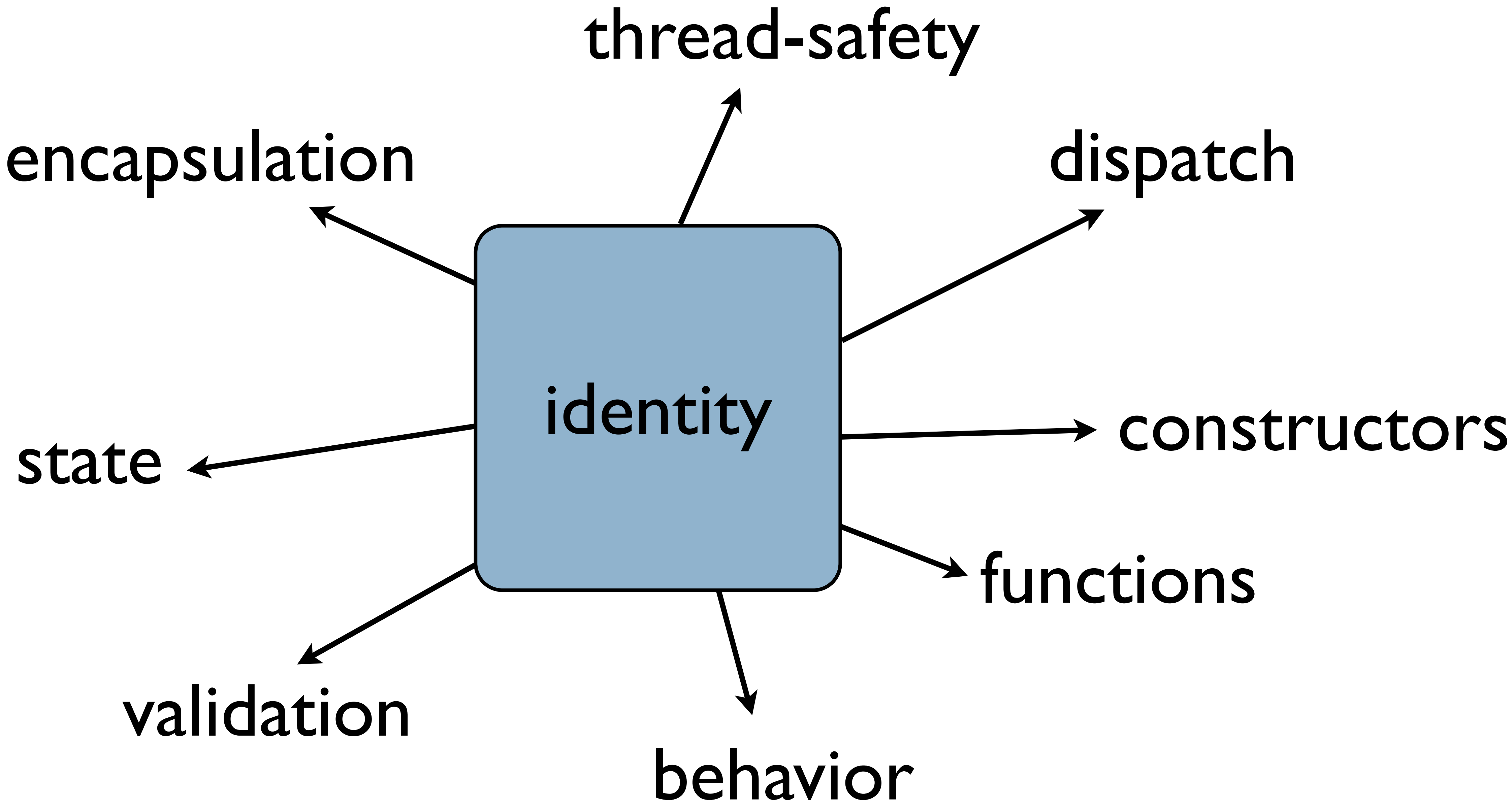
```
(time (def v (vrange 1000000)))
"Elapsed time: 1130.721 msecs"

(time (def v2 (vrange2 1000000)))
"Elapsed time: 82.191 msecs"
```

transients

- usage:
  - **`transient`**
  - bang updates: **`assoc!`** **`conj!`** etc.
  - **`persistent!`**
- optimization, not coordination
  - O(1) creation from persistent object
  - fast, isolated updates
  - O(1) conversion back to persistent object

what about objects?

oo: one identity fits all



clojure: bespoke code in an off-the-rack world

clojure’s four elevators

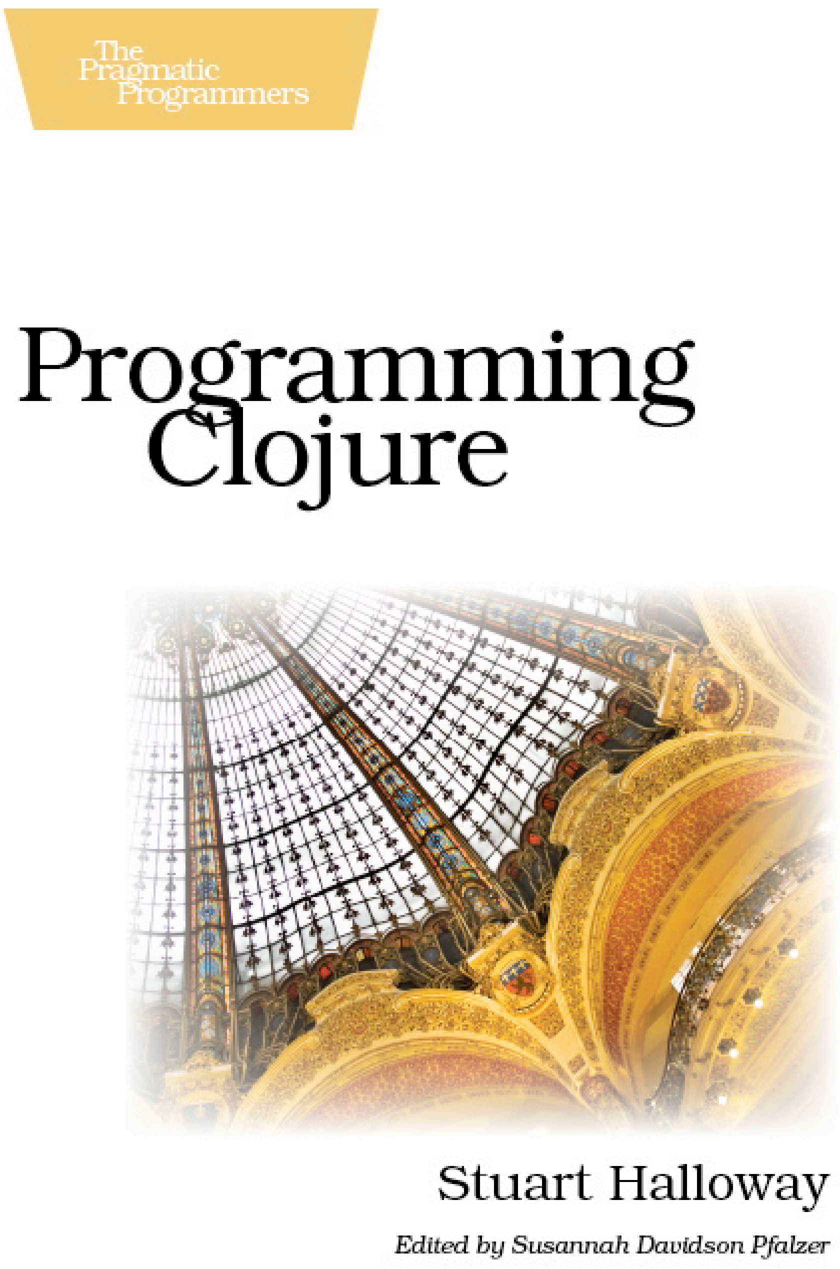
java interop

lisp

functional

state

Email:	stu@thinkrelevance.com
Office:	919-442-3030
Twitter:	twitter.com/stuarthalloway
Facebook:	stuart.halloway
Github:	stuarthalloway
This talk:	http://github.com/stuarthalloway/clojure-presentations
Talks:	http://blog.thinkrelevance.com/talks
Blog:	http://blog.thinkrelevance.com
Book:	http://tinyurl.com/clojure



The Pragmatic Programmers

Programming Clojure

Stuart Halloway

Edited by Susannah Davidson Pfalzer