clojure

for rubyists

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

stuart halloway
http://thinkrelevance.com

assumptions

seven reasons

syntax

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}`

homiconicity: code is data

function call

```
(concat [1 2] [3 4])
-> (1 2 3 4)
```

“operators”

```
(+ 1 2)
-> 3

(+ 1 2 3)
-> 6
```

function definitions

```
(defn
 hello
 "Returns greeting that includes name"
 [name]
 (str "Hello, " name))
-> #'user/hello
```

control flow

```
(if hell-freezes-over
  (skate!)
  (roast!))
```

java interop

```
(.. "stupid"
    (substring 0 3)
    (toUpperCase))
-> "STU"
```

error handling

```
(try
 (/ 1 0)
 (catch ArithmeticException e
   "should have expected that"))
```

namespaces and imports

```
(ns demo
  (:use clojure.contrib.pprint
        compojure)
  (:import java.io.File)
  (:require
   [clojure.http.resourcefully
    :as r]))
```

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

homiconic benefits

easy to parse

regularity simplifies metaprogramming

data library = metaprogramming library

"language" features are library code

example: 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 #()

fn arg

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

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)
```

good implementations have a 1-1 ratio of pseudocode/code

1. 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.

2. persistent data structures

persistent data structures

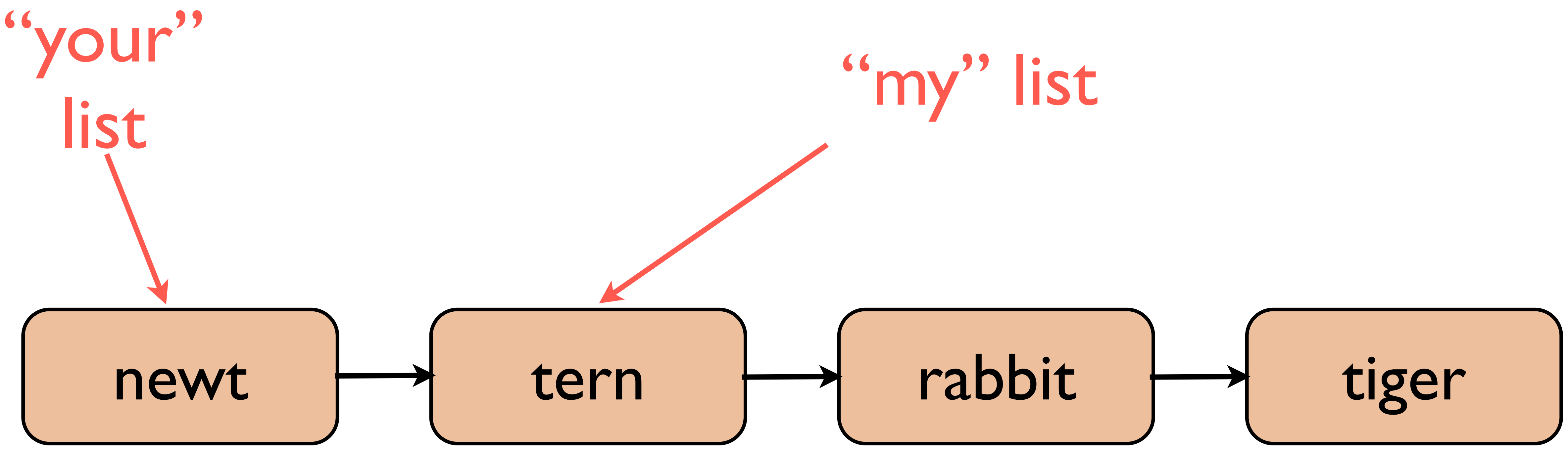
immutable

“change” by function application

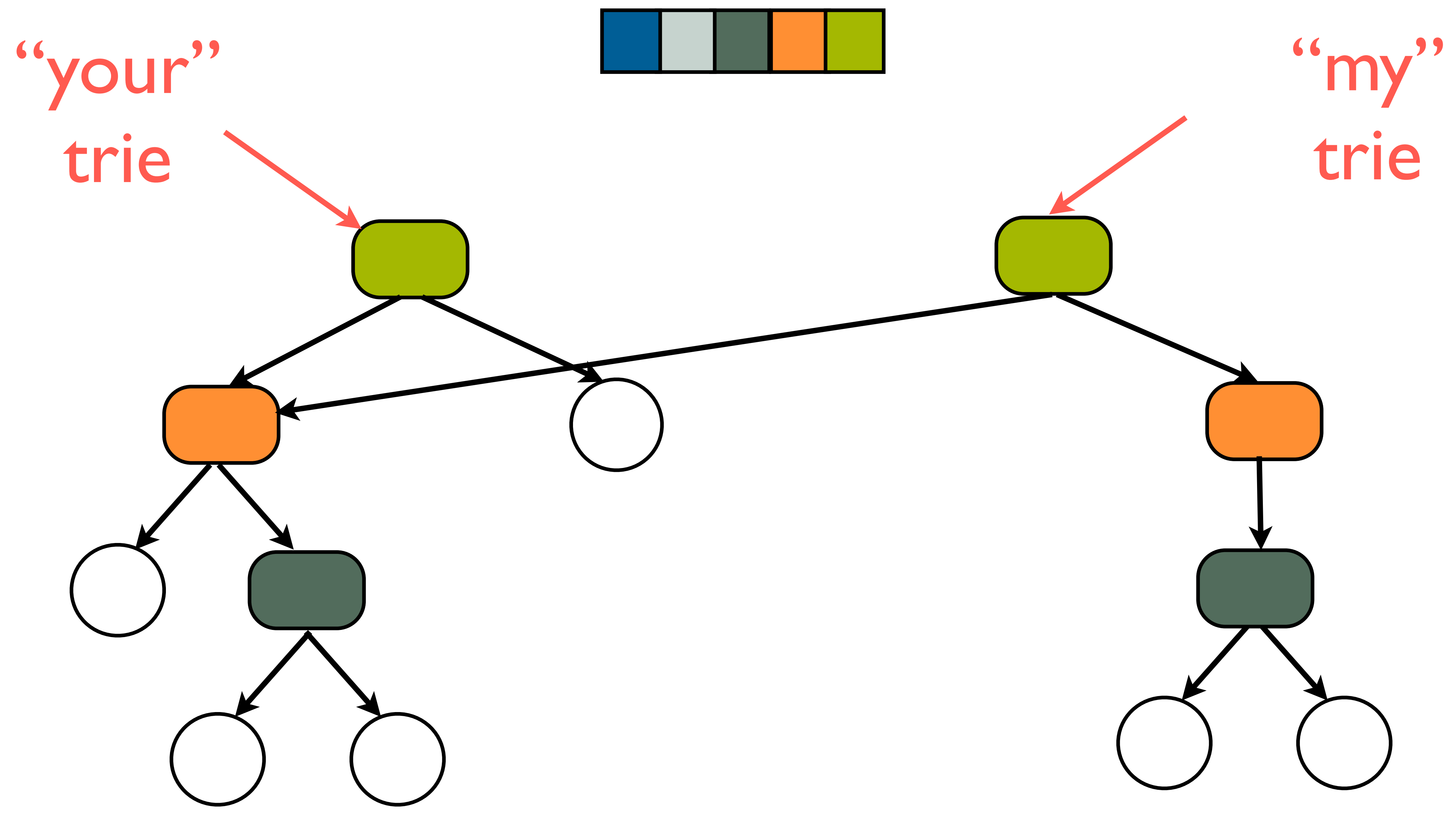
maintain performance guarantees

full-fidelity old versions

persistent example: linked list

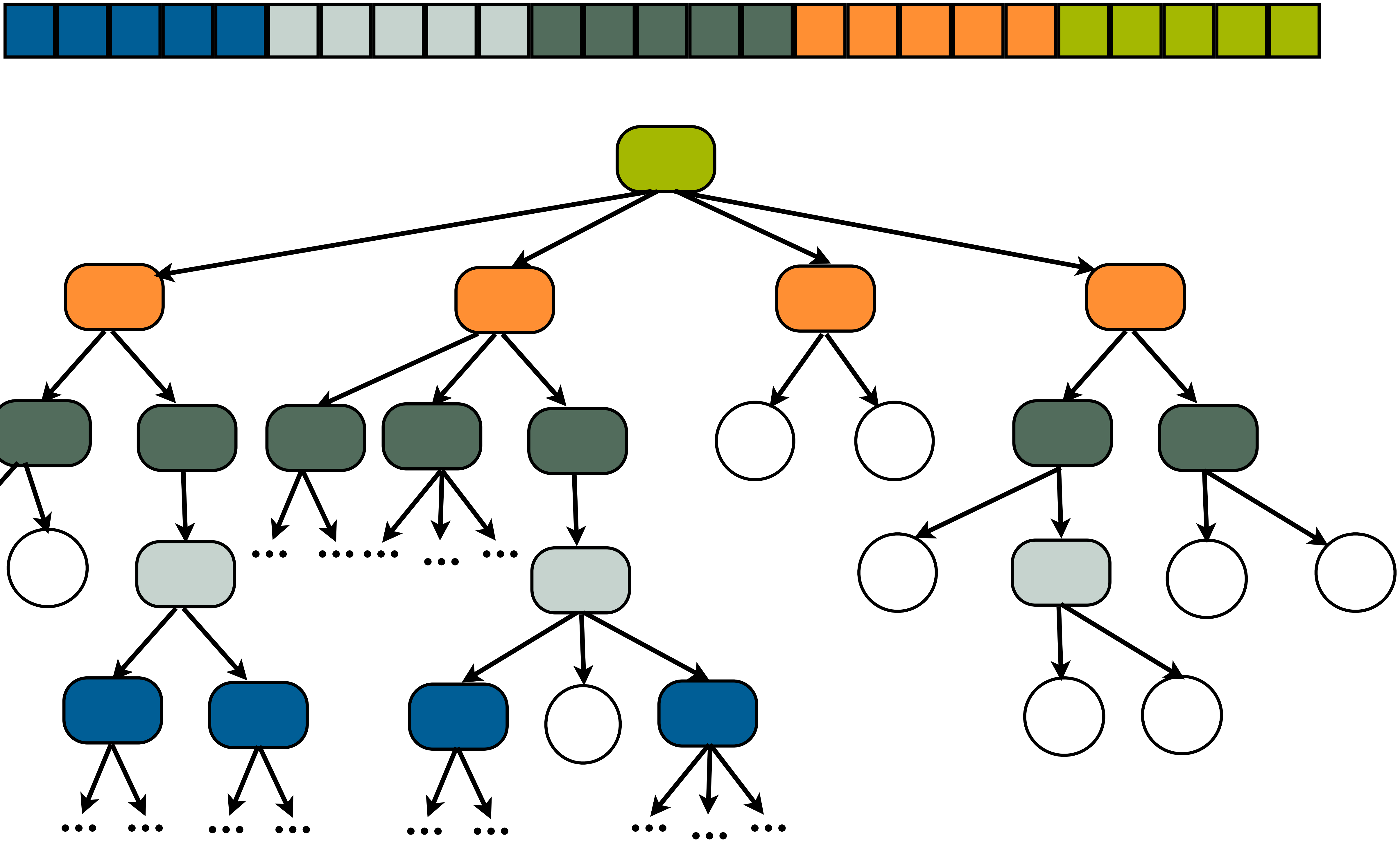


bit-partitioned tries



log2 n: too slow!

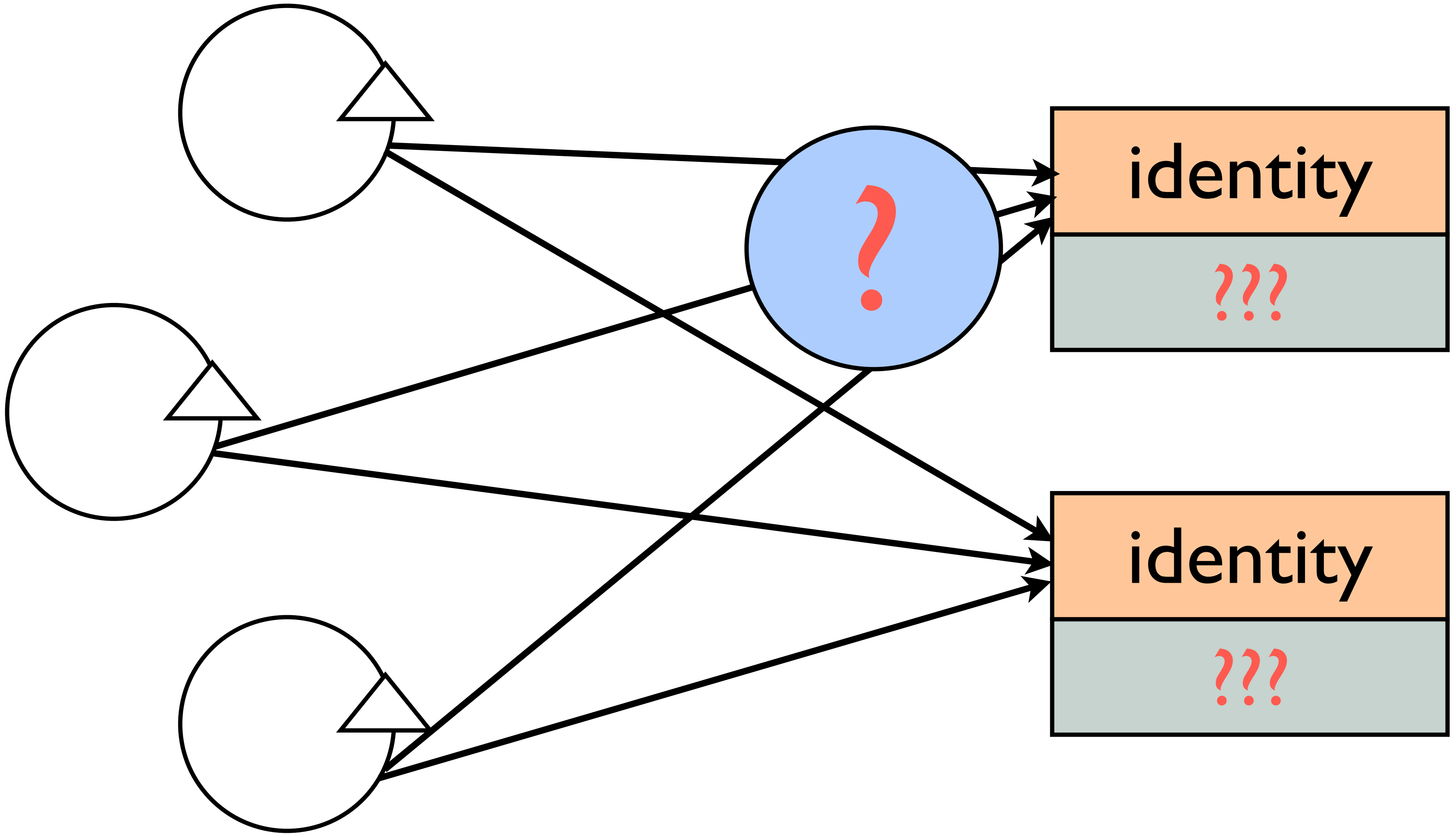
32-way tries



clojure: ‘cause log32 n is fast enough!

3. values, identity, and state

mutable oo is incoherent



mutable objects don't compose

mutation crosscuts contracts

duck typing doesn't help

- "deadlocks like a duck" ?
- "corrupts like a duck" ?

coordination required *even for reads*

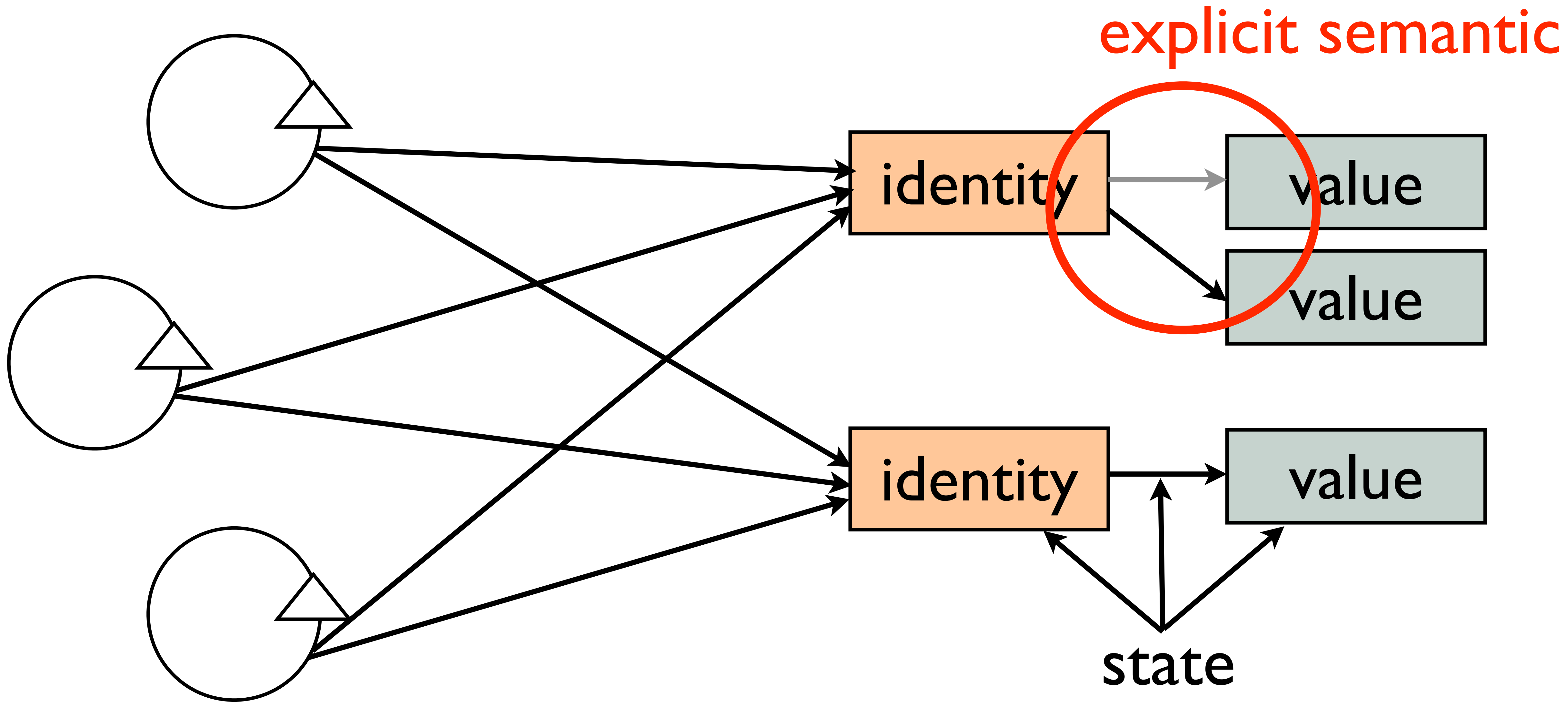
no general way to be lazy/cached

code reading is recursive speculation

cause and effect

if	**reuse will take the form of**
objects compose well	objects
objects compose poorly	domain-limited frameworks and plugins with constrained non-language semantics for state

clojure


explicit semantic
identity
identity
value
value
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**	-

4. unified update model

refs and stm

ref example: chat

identity

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

initial value

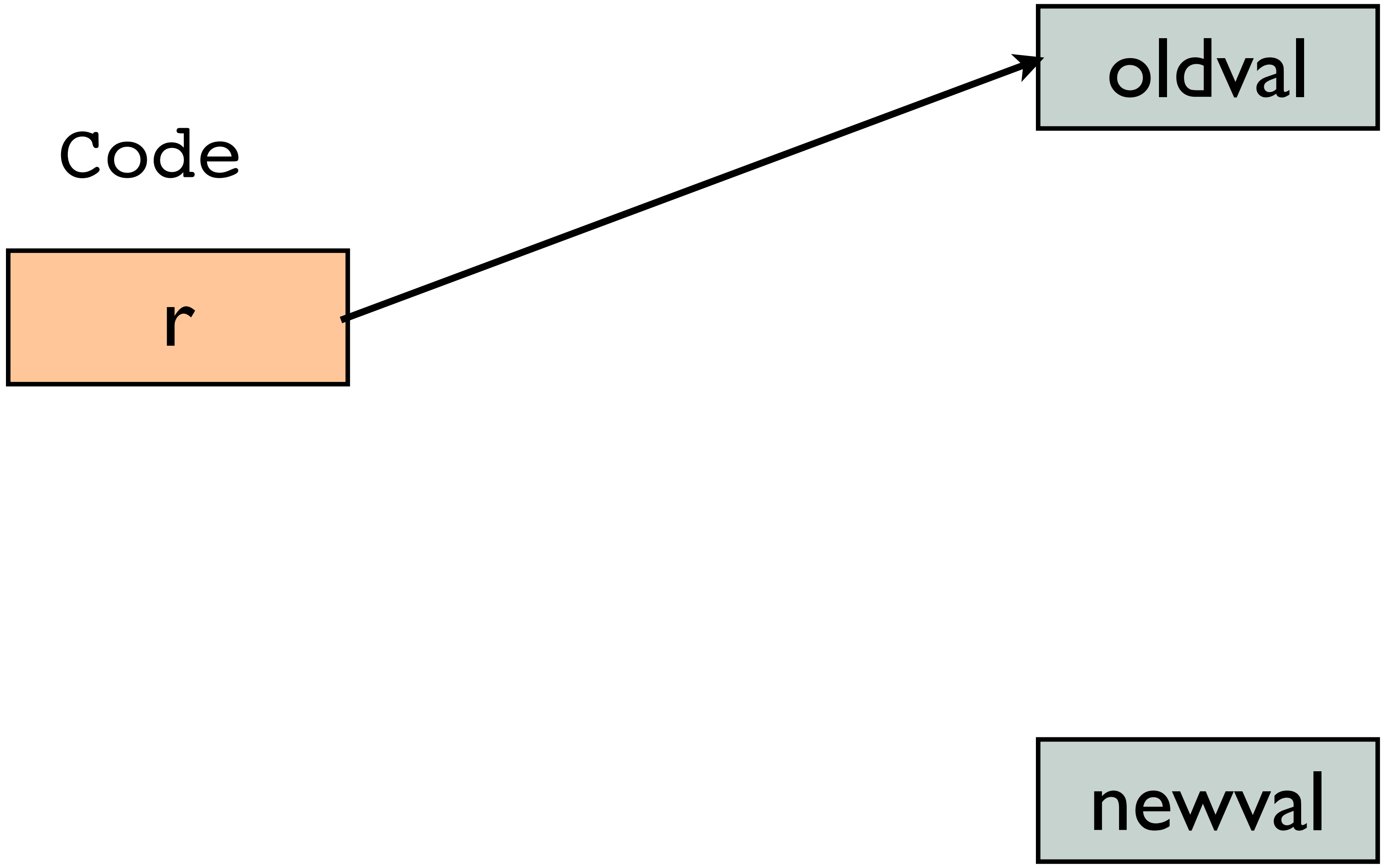
reading value

```
(deref messages)
-> ()

@messages
-> ()
```

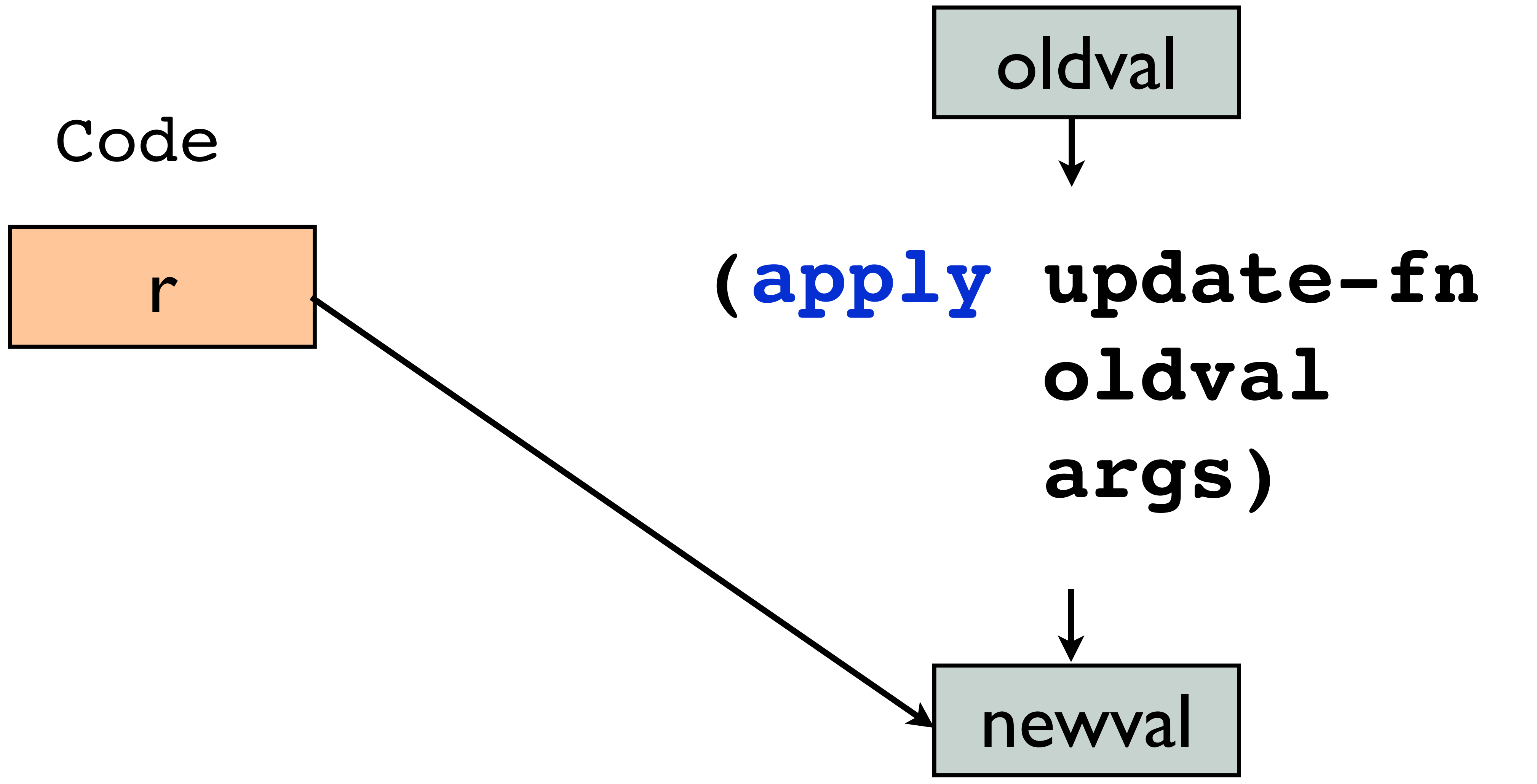
alter

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



alter

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



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
- additional semantics per ref type
  - automatic retry
  - sync/async
  - shared/isolated

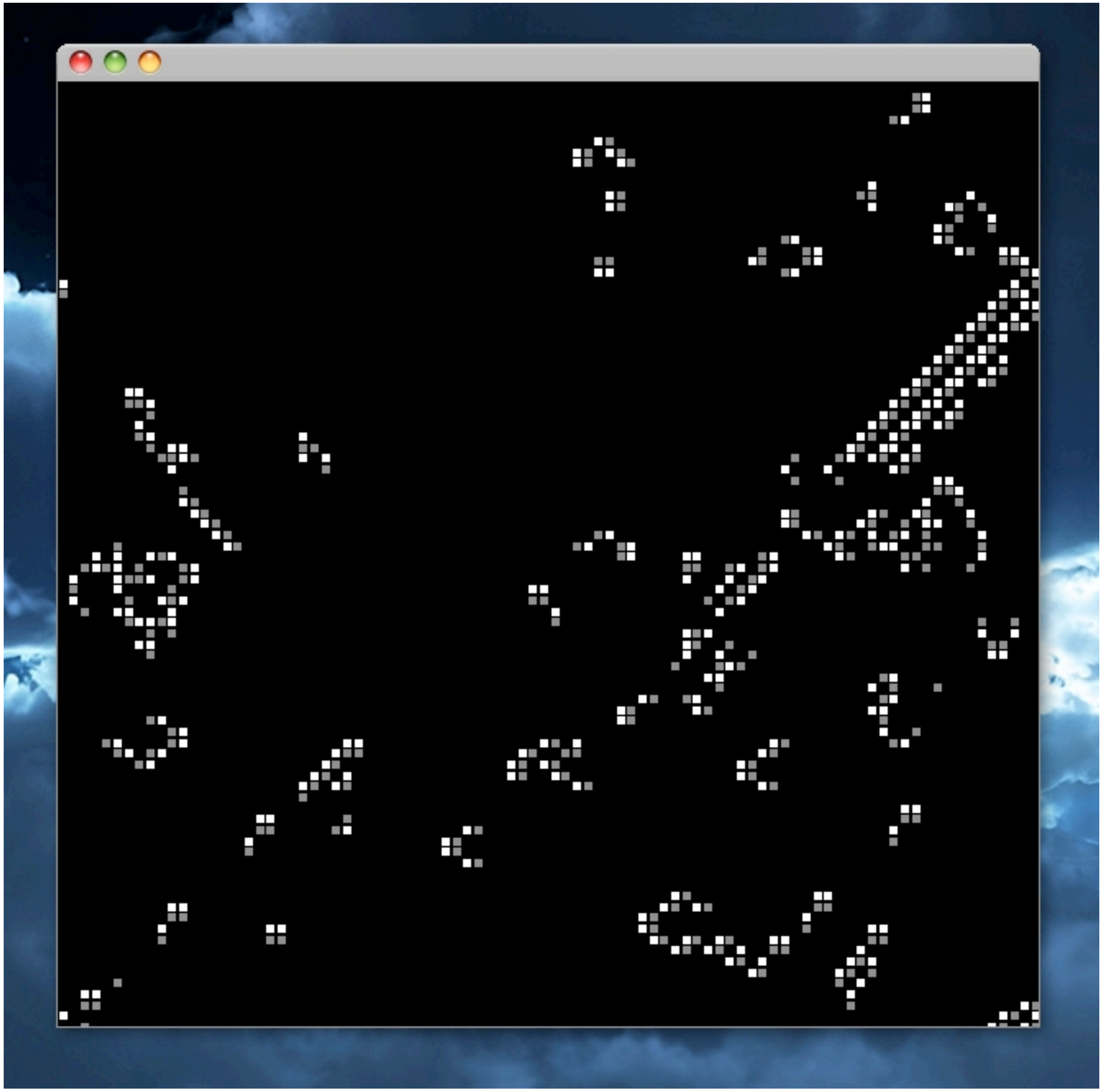
a sane approach
to local state
permits coordination,
but does not require it

unified update model

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

atoms

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



board is just a value

```
(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

state is localized

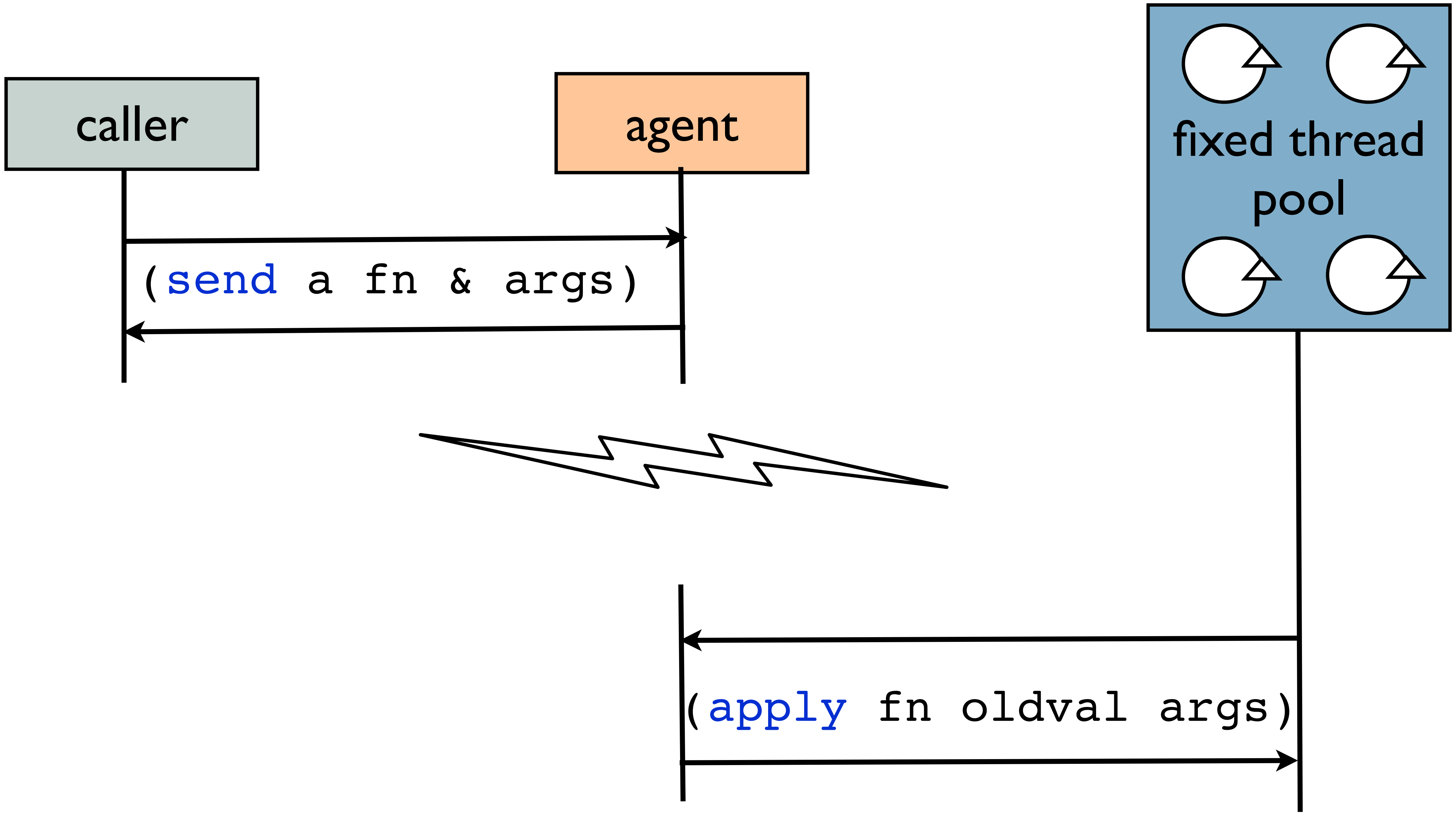
```
(let [stage (atom (new-board))]
  ...)

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

only stateful line in the entire model

agents

send



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

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

example: deferred logging

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

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

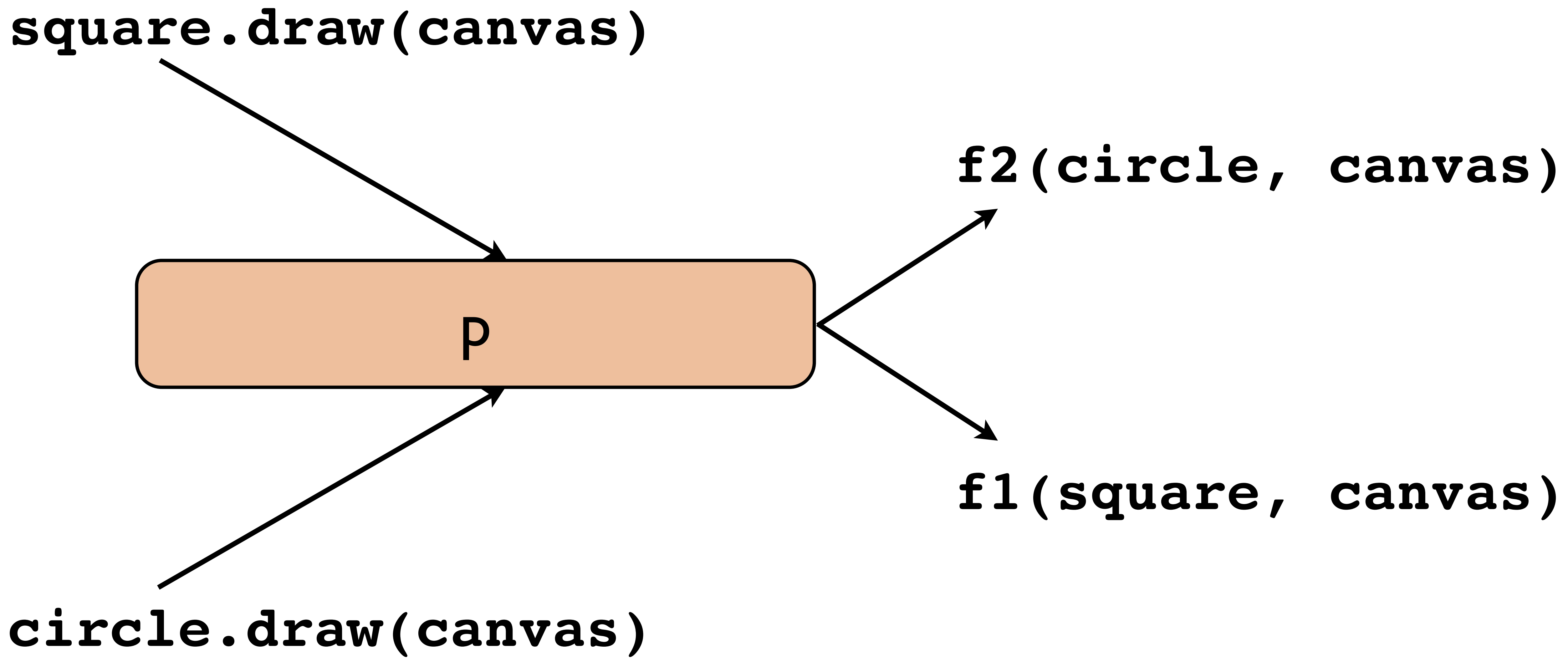
other def forms

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

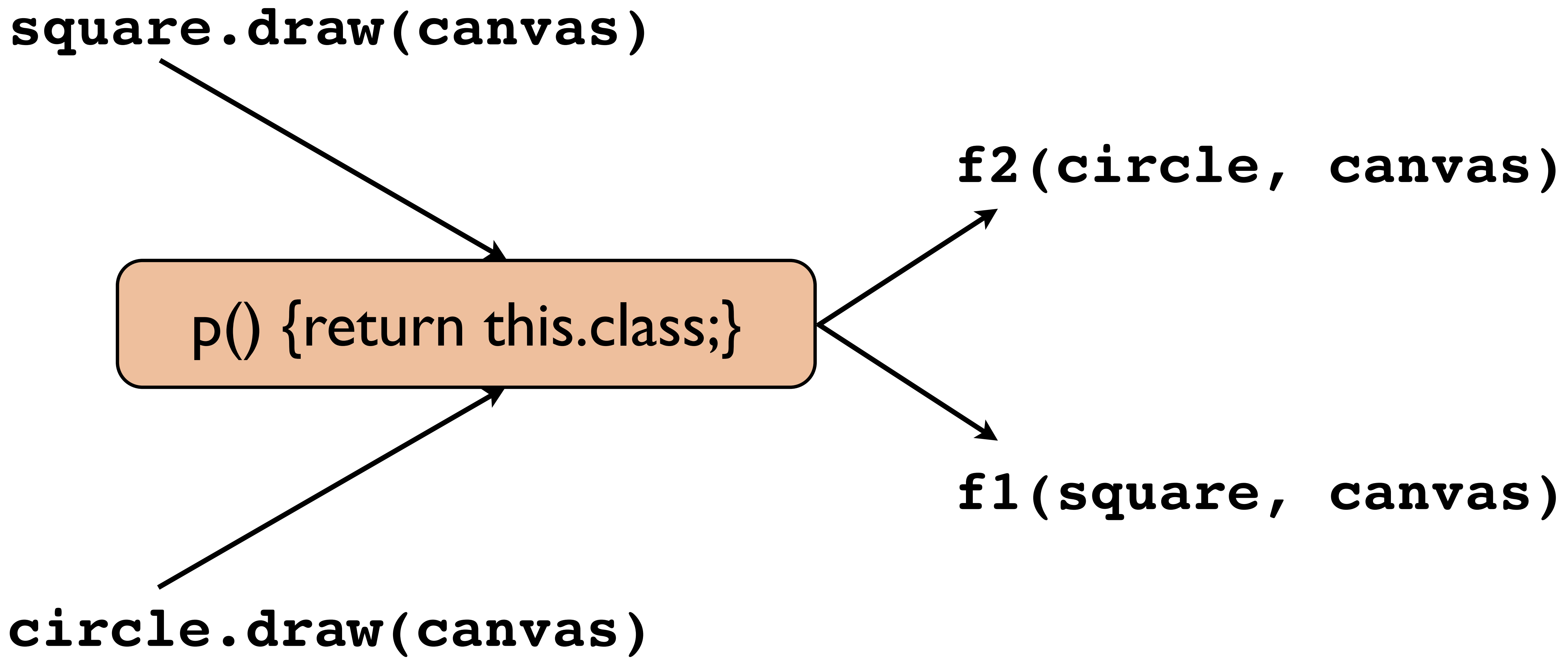
many of these are in clojure.contrib.def...

5. multimethods

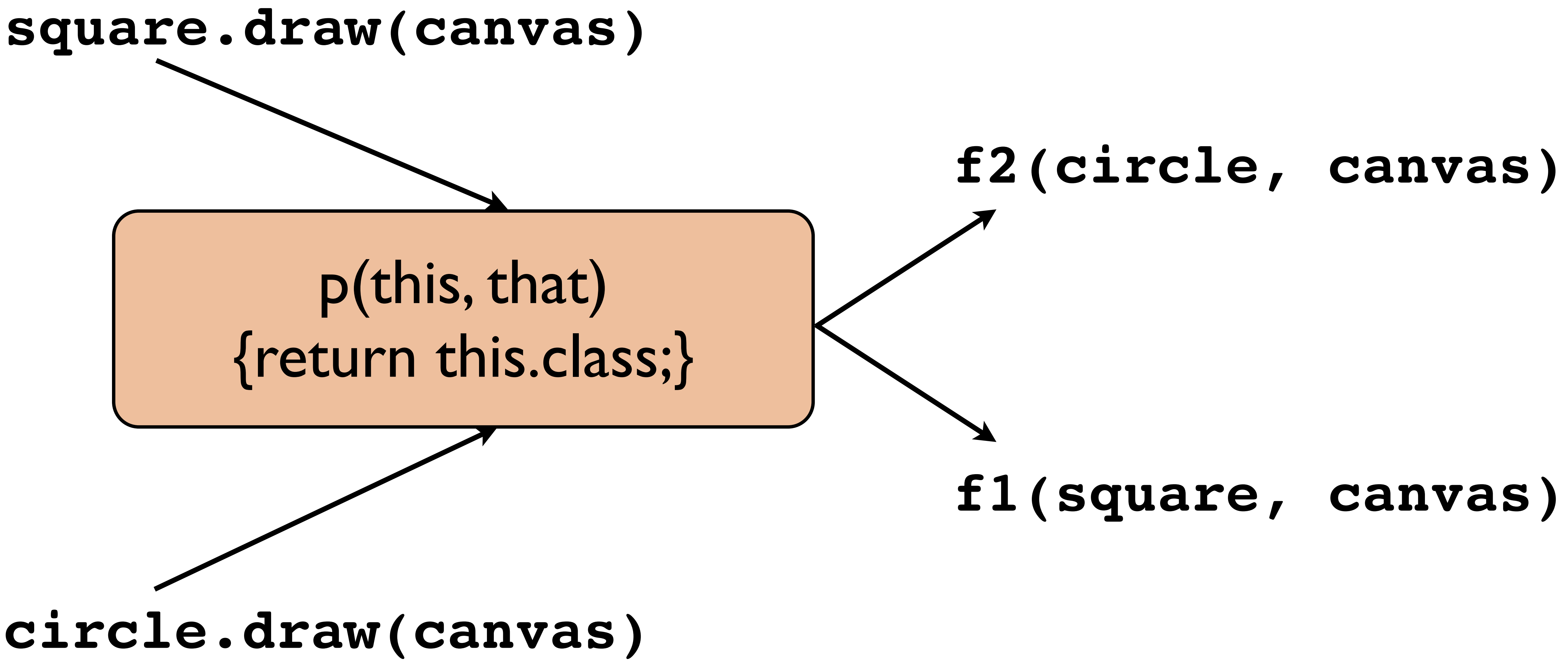
polymorphism



p is just a function



this isn't special



check all args

```
(fn [this, that]
  [(class this)
   (class that)])
```

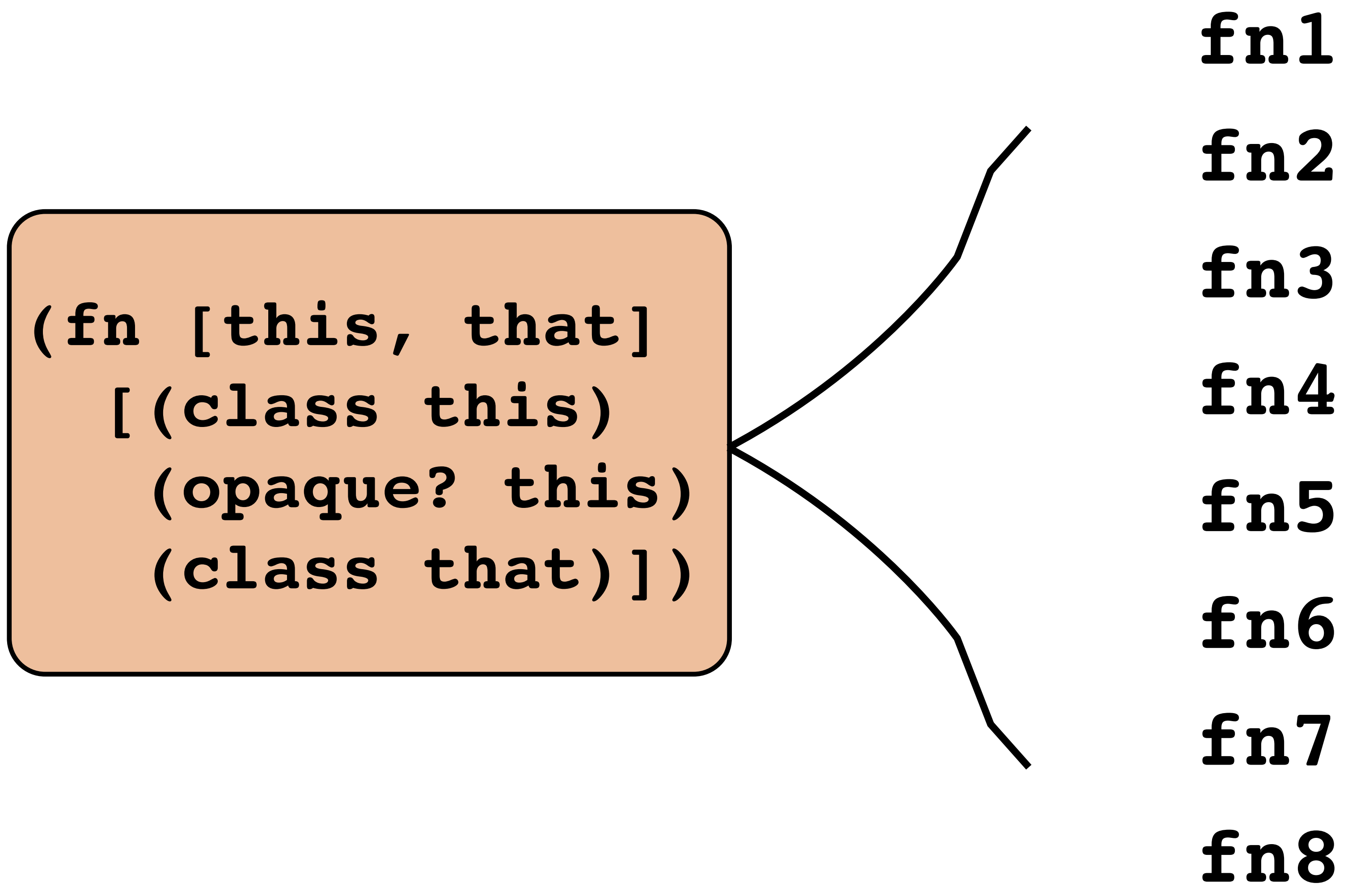
```
(fn [square, canvas])

(fn [circle, canvas])

(fn [square, surface])

(fn [circle, surface])
```

check arg twice



```
(fn [this, that]
  [(class this)
   (opaque? this)
   (class that)])
```

example: coerce

define a
multimethod

```
(defmulti coerce
  (fn [dest-class src-inst]
    [dest-class (class src-inst)]))
```

based on
dest (a class)

and src
(an inst)

method impls

dispatch value
to match

args

body

```
(defmethod coerce
  [java.io.File String]
  [_ str]
  (java.io.File. str))

(defmethod coerce
  [Boolean/TYPE String] [_ str]
  (contains?
   #{"on" "yes" "true"}
   (.toLowerCase str)))
```

defaults

```
(defmethod coerce
  :default
  [dest-cls obj]
  (cast dest-cls obj))
```

dispatch comparison

language	java	ruby	clojure
basic polymorphism?	✔	✔	✔
change "methods" at runtime?		✔	✔
change "types" at runtime?		✔	✔
dispatch based on all arguments?		*	✔
arbitrary fn dispatch?		*	✔
pattern matching		*	*

dispatch workarounds are the middle managers of the design patterns movement

class inheritance

```
(defmulti whatami? class)

(defmethod whatami? java.util.Collection
  [_] "a collection")

(whatami? (java.util.LinkedList.))
-> "a collection"

(defmethod whatami? java.util.List
  [_] "a list")

(whatami? (java.util.LinkedList.))
-> "a list"
```

add methods anytime

most derived type wins

name inheritance

```
(defmulti interest-rate :type)
(defmethod interest-rate ::account
  [_] 0M)
(defmethod interest-rate ::savings
  [_] 0.02)
```

double colon (::) is shorthand for resolving keyword into the current namespace, e.g. ::savings == :my.current.ns/savings

deriving names

derived name

base name

```
(derive ::checking ::account)
(derive ::savings ::acount)

(interest-rate {:type ::checking})
-> 0M
```

there is no ::checking method, so select method for base name ::account

multimethods lfu

function	notes
`prefer-method`	resolve conflicts
`methods`	reflect on {dispatch, meth} pairs
`get-method`	reflect by dispatch
`remove-method`	remove by dispatch
`prefers`	reflect over preferences

6. macros

the **if** special form

```
(if test
   then
   else?)
```

evaluate only if `test` is logical true

evaluate only if `test` is logical false

if-like things cannot be functions!

function calls

- evaluate their args
- pass args to implementation

if

- evaluates an arg
- decides which other args to evaluate, and when

lisp macros

- get access to source forms
  - after they are read
  - before compile/interpret
- macroexpand forms into other forms
- choose when/how to evaluate each argument

example: **when**

```
(when x
  (println "x is true")))
```

```
(defmacro when
[test & body]
(list
 'if test
 (cons 'do body)))
```

$$\Big\downarrow$$ macroexpansion

```
(if x
  (do (println "x is true")))
```

quoting and list-building

```
(defmacro when
  [test & body]
  (list
   'if test
   (cons 'do body)))
```

quoting

list-building

syntax-quoting

```
(defmacro when
  [test & body]
  (list
   'if test
   (cons 'do body)))
```

=

unquote

```
(defmacro when
  [test & body]
  `(if ~test
     (do ~@body)))
```

syntax-quote

unquote-splicing

test your macros!

```
(macroexpand-1
  '(when x
     (println "x is true")))

-> (if x
      (do (println "x is true")))
```

a bench macro

```
(defmacro bench [expr]
  `(let [start (System/nanoTime)
         result ~expr]
     {:result result
      :elapsed (- (System/nanoTime)
                  start)}))
```

not done yet...

capture?

```
(defmacro bench [expr]
  `(let [start (System/nanoTime)
         result ~expr]
     {:result result
      :elapsed (- (System/nanoTime)
                  start)}))
```

could these
collide with
names in scope
of expr?

not done yet...

avoiding accidental capture

```
(defmacro bench [expr]
  `(let [start (System/nanoTime)
         result ~expr]
     {:result result
      :elapsed (- (System/nanoTime)
                  start)}))
```

```
(bench (x))
-> java.lang.Exception:
   Can't let qualified name: user/start
```

not done yet...

use auto-gensyms

```
(defmacro bench [expr]
  `(let [start# (System/nanoTime)
         result# ~expr]
     {:result result#
      :elapsed (- (System/nanoTime)
                  start#)}))
```

# suffix generates unique symbol within a quoted form

generated symbols

```
(macroexpand-1 '(bench (x)))

-> (clojure.core/let
     [start__37__auto__ (java.lang.System/nanoTime)
      result__38__auto__ (x)]
     {:result result__38__auto__,
      :elapsed (clojure.core/-
                 (java.lang.System/nanoTime)
                  start__37__auto__)})
```

common macro types

type	examples
control flow	`when when-not and or`
vars	`defn defmacro defmulti`
java interop	`.. doto deftype proxy`
rearranging	`-> ->> -?>`
scopes	`dosync time with-open`
“special form”	`fn lazy-seq let`

7. metadata

metadata:
data that is
orthogonal to the
value of an object

metadata uses

documentation

serialization

protection

optimization

relationships (e.g. test -> testee)

grouping/typing (?)

add & retrieve metadata

add
metadata

data

```
(def x (with-meta
         {:password "swordfish"}
         {:secret true}))
-> #'user/x
```

metadata

```
x
-> {:password "swordfish"}

(meta x)
-> {:secret true}
```

retreive
metadata

sugar: #^, ^

add metadata

metadata first!

```
(def y #^{:secret true}
         {:password "swordfish"}))
-> #'user/y
```

```
^y
-> {:secret true}
```

retrieve metadata

subtleties

- metadata can be on data, **or on a var**
- to place metadata on a var:
  - put it on the symbol when defing the var
  - compiler will copy it to the var
- metadata cannot be added to a function

var: add metadata

add
metadata

to symbol

```
(def #^{:secret true} z
  {:password "swordfish"})
-> #'user/z
```

```
^z
-> nil
```

z's data has no metadata

var: retrieve metadata

`#'` is var-quote

```
^#'z
-> {:ns #<Namespace user>,
      :name z,
      :file "NO_SOURCE_PATH",
      :line 34,
      :secret true}
```

implicit metadata

explicit metadata

type metadata example

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [s]
  (if (.isEmpty s)
    s
    (let [up (.. s
                 (substring 0 1)
                 (toUpperCase))
          down (.. s
                   (substring 1)
                   (toLowerCase))]
      (.concat up down))))
```

warn-on-reflection

```
(set! *warn-on-reflection* true)
-> true

(require :reload 'demo.capitalize)
Reflection warning, demo/capitalize.clj:6 -
  reference to field isEmpty can't be resolved.
Reflection warning, demo/capitalize.clj:8 -
  call to substring can't be resolved.
Reflection warning, demo/capitalize.clj:8 -
  call to toUpperCase can't be resolved.
Reflection warning, demo/capitalize.clj:11 -
  call to substring can't be resolved.
Reflection warning, demo/capitalize.clj:11 -
  call to toLowerCase can't be resolved.
Reflection warning, demo/capitalize.clj:14 -
  call to concat can't be resolved.
-> nil
```

add type metadata

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [#^String s]
  (if (.isEmpty s)
    s
    (let [up (.. s
                 (substring 0 1)
                 (toUpperCase))
          down (.. s
                   (substring 1)
                   (toLowerCase))]
      (.concat up down))))
```

s is known to be a String

no more warnings

```
(set! *warn-on-reflection* true)
-> true

(require :reload 'demo.capitalize)
-> nil
```

more idiomatic

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [#^String s]
  (if (.isEmpty s)
    s
    (.concat
     (.toUpperCase (subs s 0 1))
     (.toLowerCase (subs s 1)))))
```

still non-reflective,
if subs is non-reflective

var metadata convenience

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...

seven reasons

lazy sequences

persistent data structures

values, identity, and state

unified update model

multimethods

macros

metadata

love, sex, friendship

ruby	clojure
built for love	built for love
consenting adults	consenting adults
friendship	friendship

variety

ruby	**clojure**
mutable objects ad hoc update model	immutable data unified update model
open classes open instances	"copying" is cheap multimethods ~ "open fns"
many tasteful hooks flexible punctuation	macros homoiconicity

The Pragmatic Programmers

Programming Clojure

Stuart Halloway

Edited by Susannah Davidson Pfalzer

Me

Email:	stu@thinkrelevance.com
Office:	919-442-3030
Twitter:	stuarthalloway
Facebook:	stuart.halloway
Github:	stuarthalloway

Talking

This talk:	github.com/stuarthalloway/ clojure-presentations
Talks:	blog.thinkrelevance.com/ talks

Writing

Blog:	blog.thinkrelevance.com
Book:	tinyurl.com/clojure