Clojure Time

Stuart Halloway
stu@clojure.com
@stuarthalloway

the plan

modeling the world

total control model

concurrency and parallelism

an approach

time

when things happen

- before/after
- later
- concurrency
- now

relative

identity

continuity over time

built by minds

sameness across a series of *perceptions*

not a name, but can be named

can be composite

perception

becoming aware via the senses

uncoordinated

provides *value*s (snapshots)

contemplate values for as long as you like

values

units of *perception*

points in *time* of *identities*

immutable

possibly composite

action

change to *identity(ies)* over *time*

independent of other *perceivers*

makes new *value*s available to *perceivers*

many possible semantics

might be coordinated

total control model

global, total control

one processor

one memory

anything else is deep voodoo

roll-your-own time model

difficulties with rolling your own

time	present unreliable, past nonexistent
identity	locking + convention
perception	ad hoc (copying?)
values	class-level convention?
action	side effects everywhere

the problems with convention

add concurrency and parallelism to the mix

“where did this dangerous assumption that Parallelism == Concurrency come from?”

http://ghcmutterings.wordpress.com/2009/10/06/parallelism-concurrency/

we need to switch mental models

Clojure's approach

syntax

atomic data types

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
regex	`#"fo*"`	Pattern
integer	`42`	Long
a.r. integer	`42N`	BigInteger
double	`3.14159`	Double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
ratio	22/7	N/A
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`	N/A

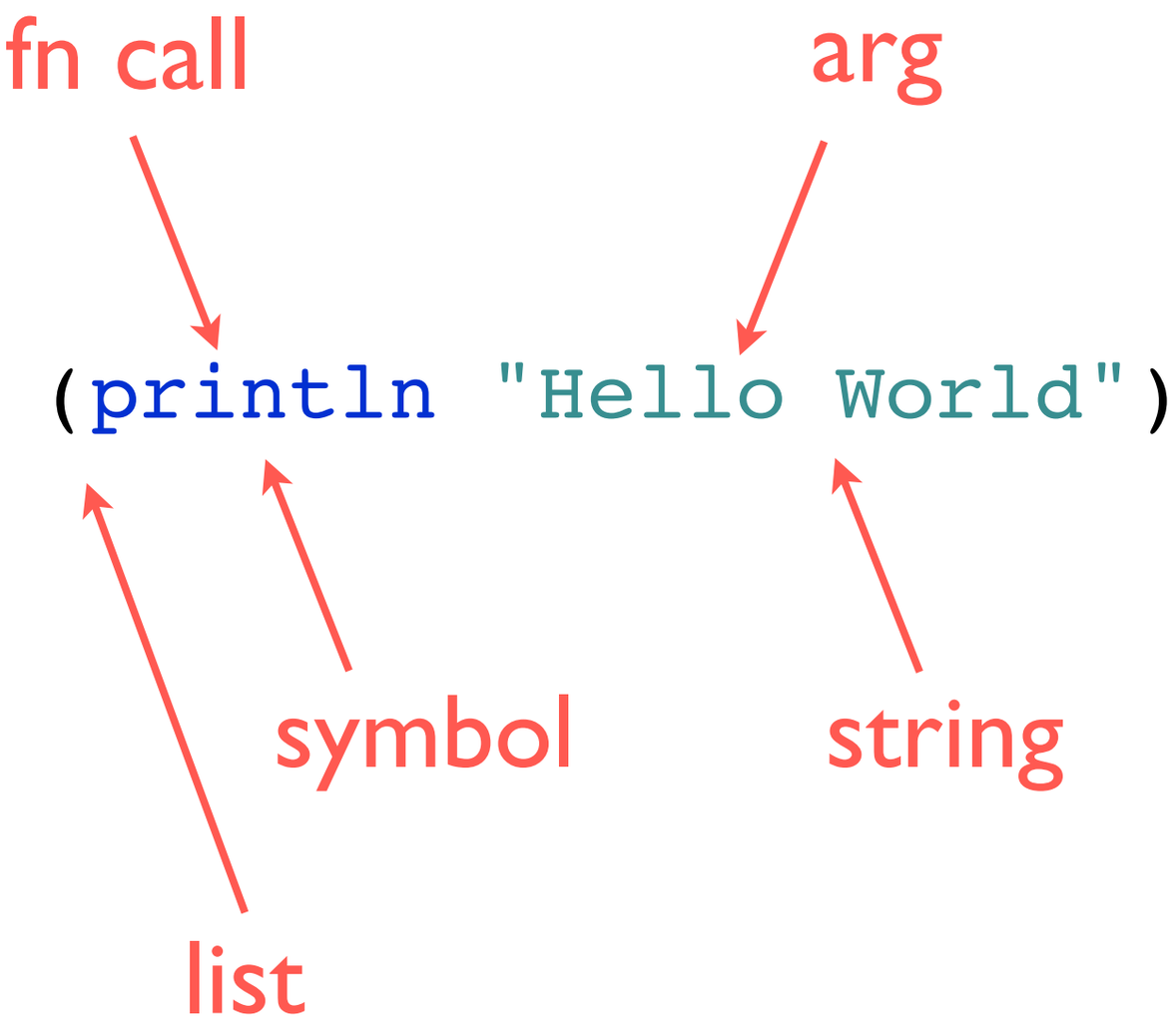
data literals

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}`

function call

semantics:

structure:



function definition

define a fn

fn name

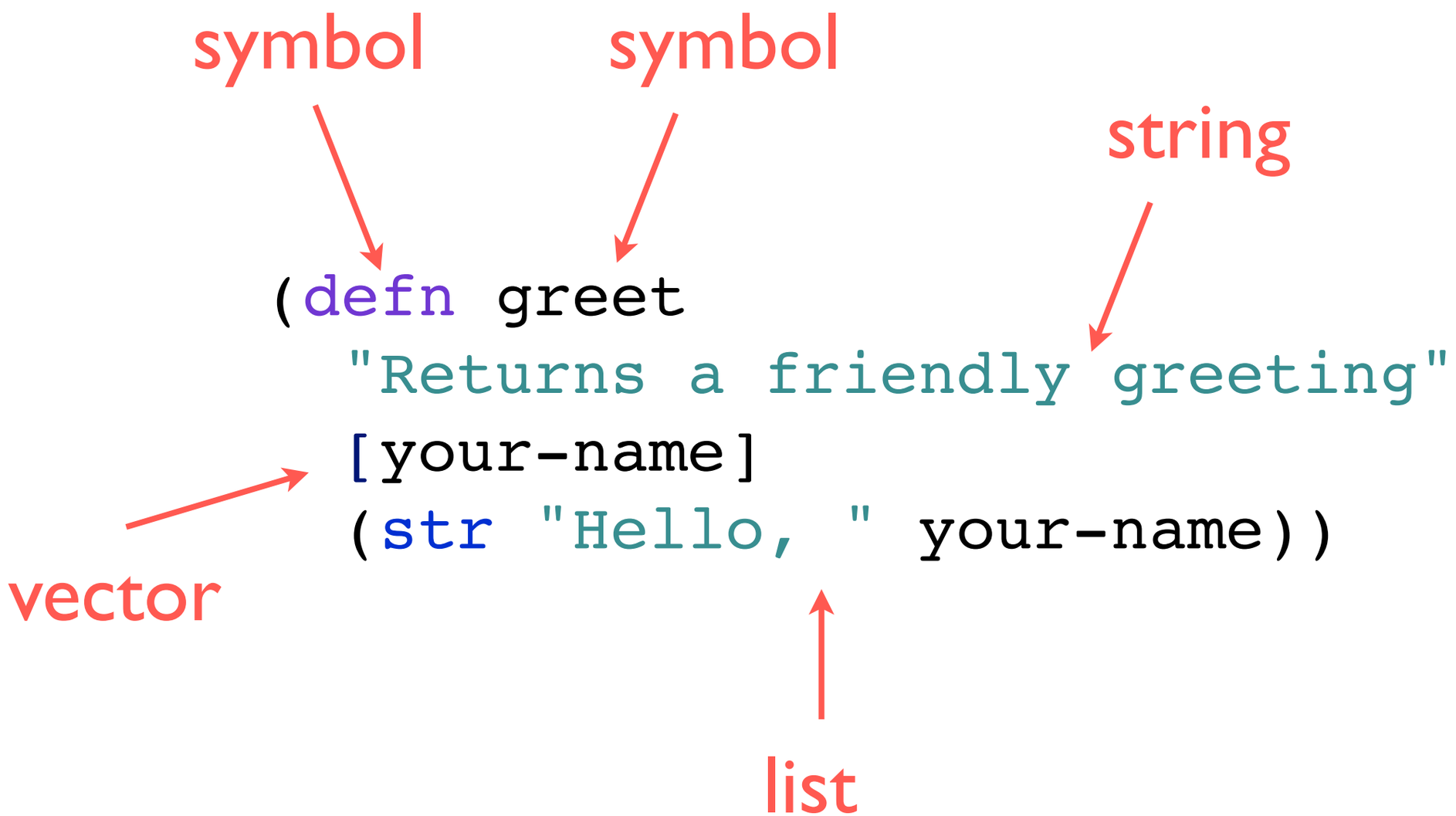
docstring

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

arguments

fn body

it's all data


symbol
symbol
string
(defn greet
"Returns a friendly greeting"
[your-name]
(str "Hello, " your-name))
vector
list

metadata

prefix with ^

class name or
arbitrary map

```
(defn ^String greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

values

persistent data structures

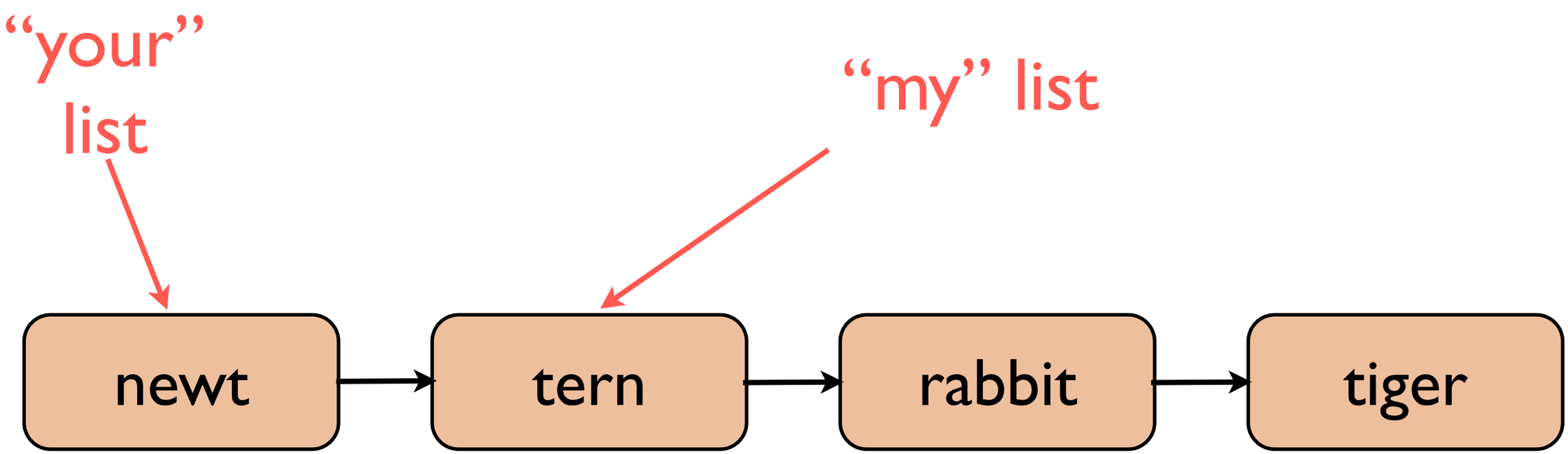
immutable

“change” by function application

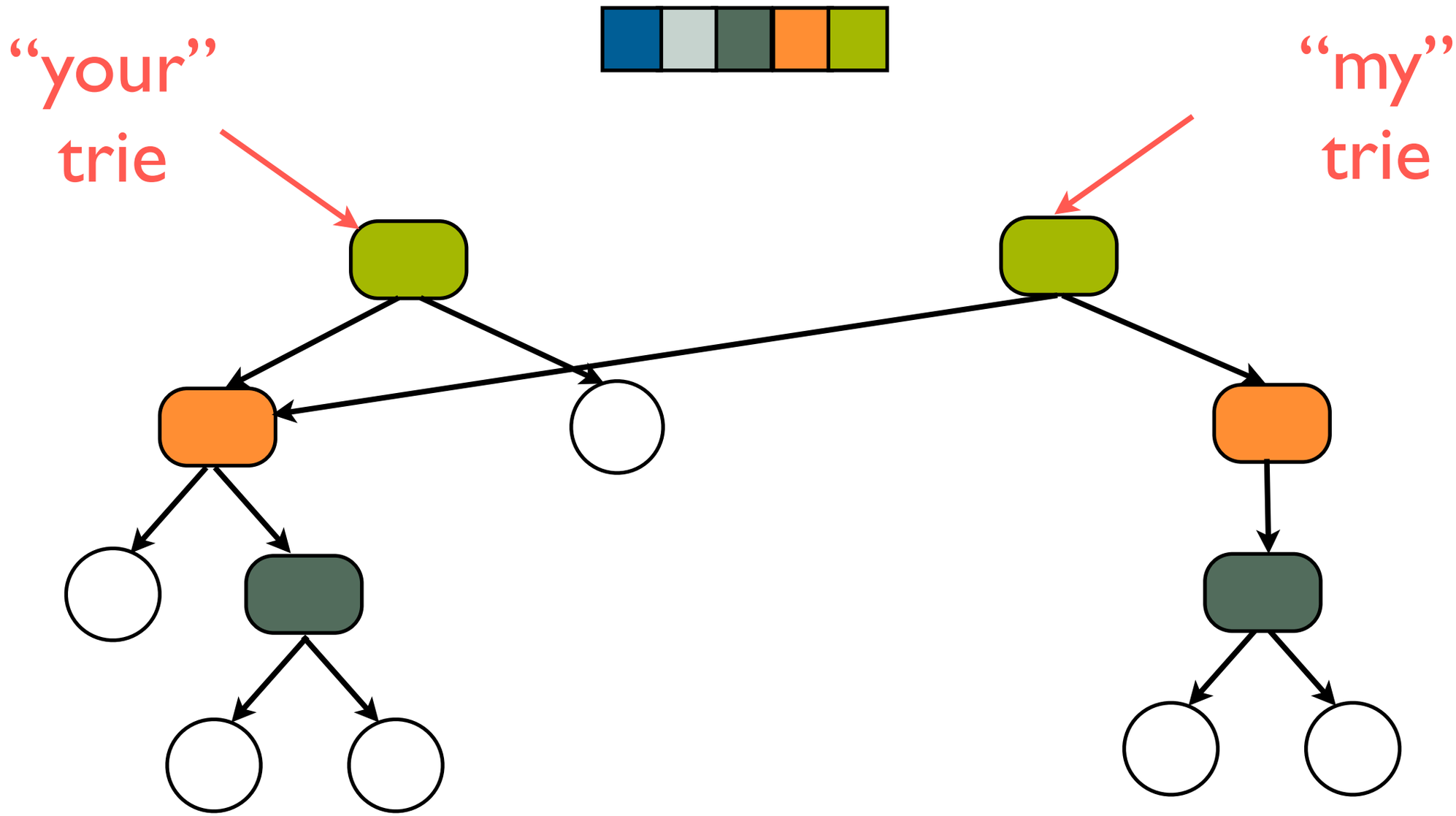
maintain performance guarantees

full-fidelity old versions

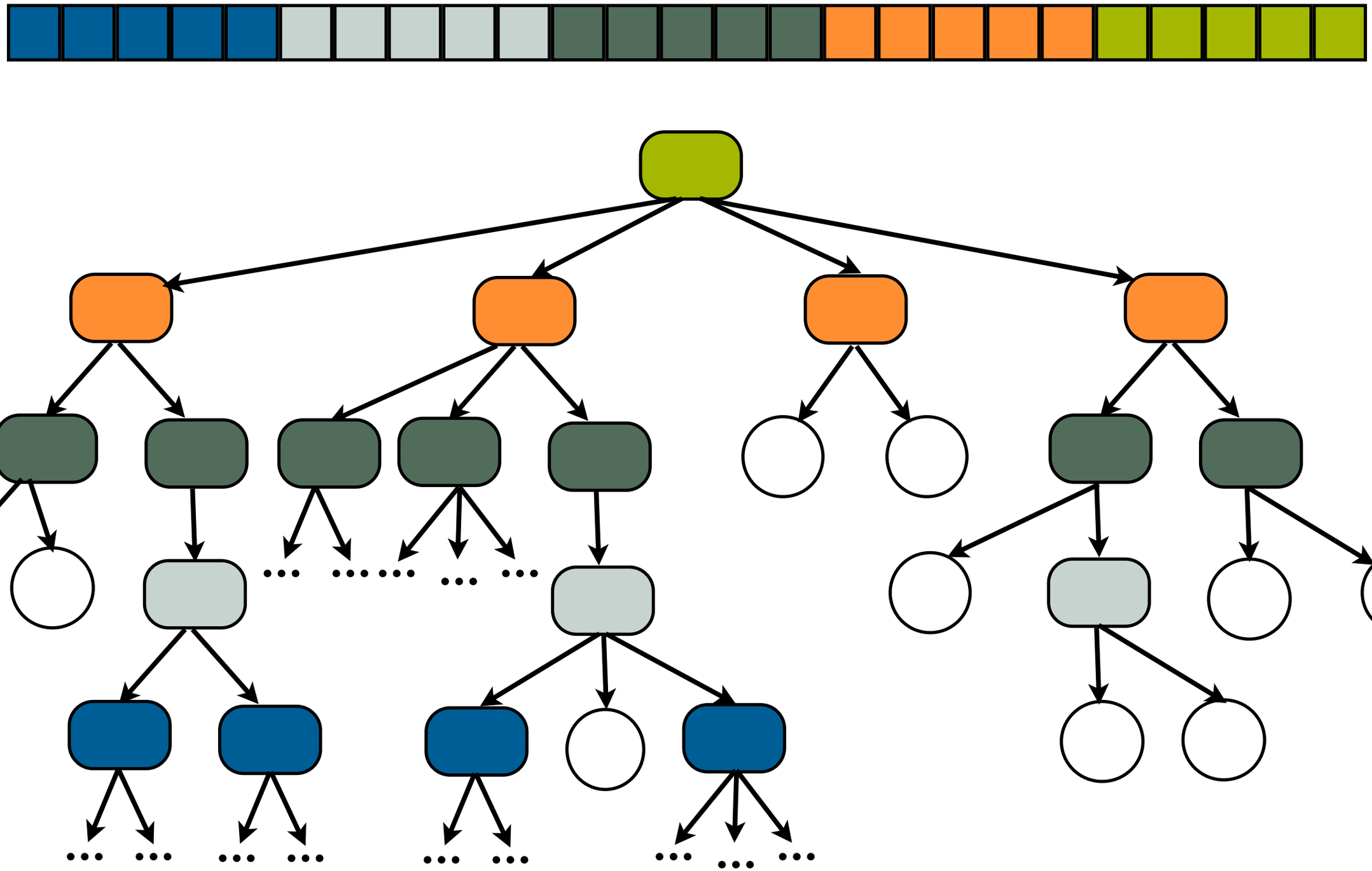
persistent example: linked list



bit-partitioned tries

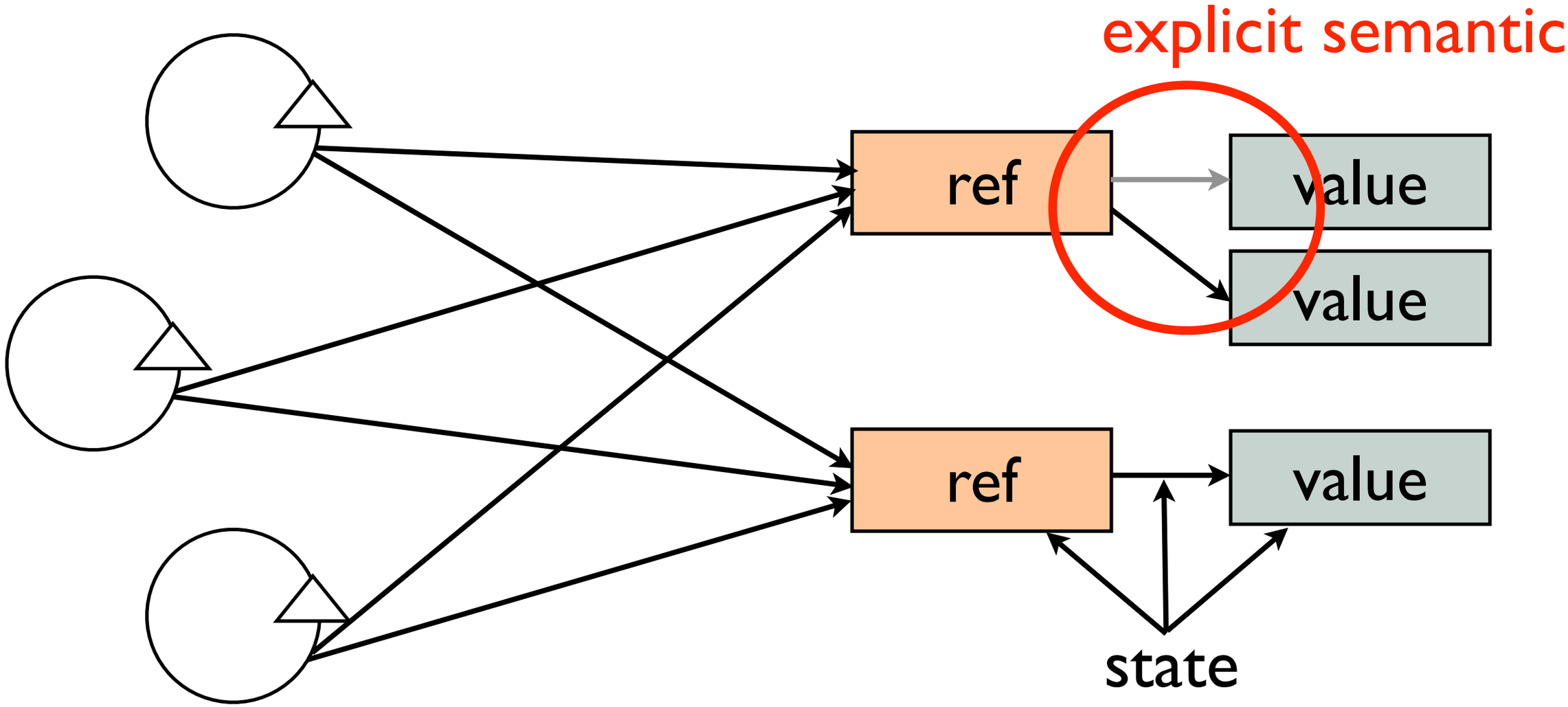


32-way tries

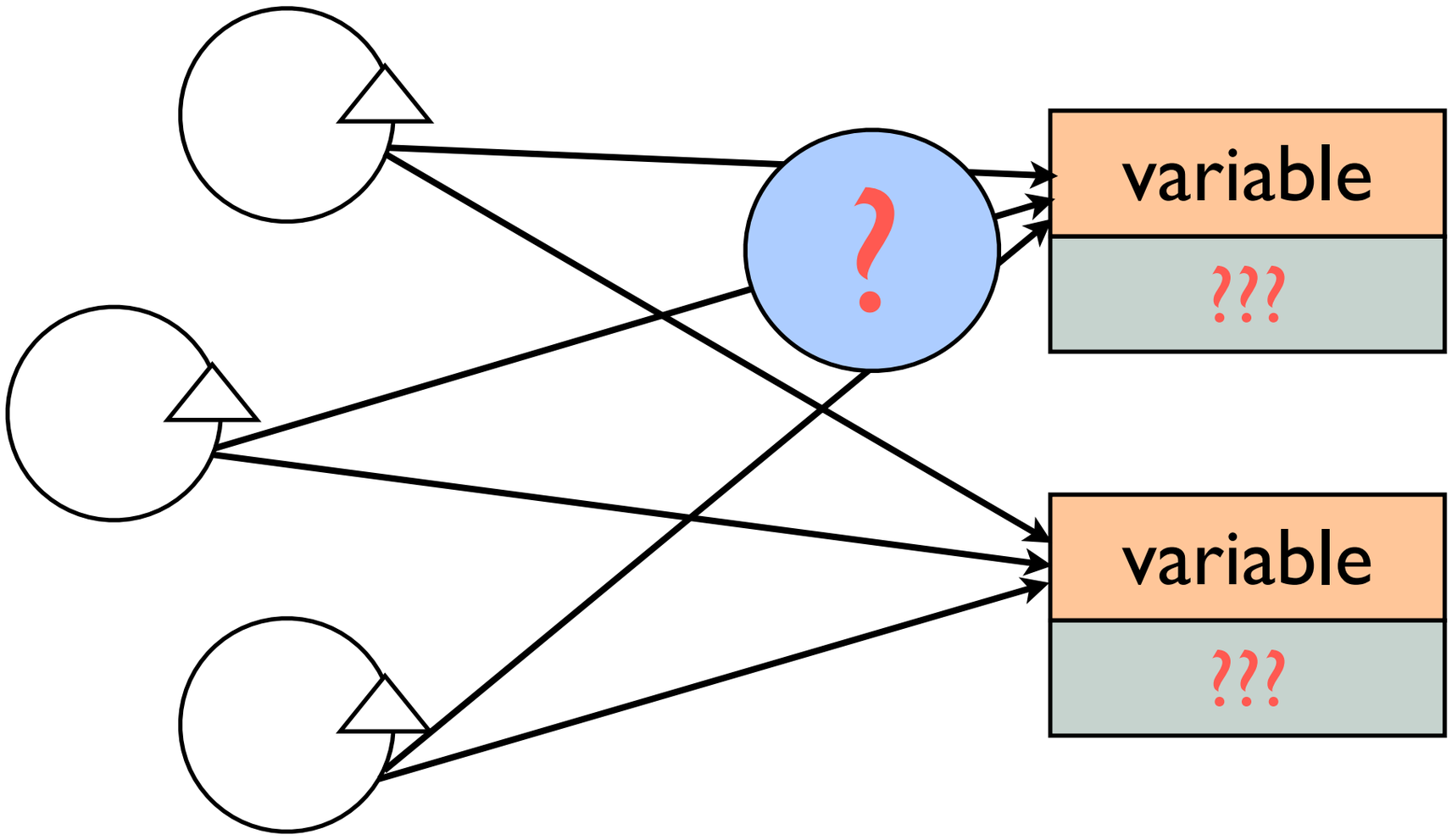


identities

refs

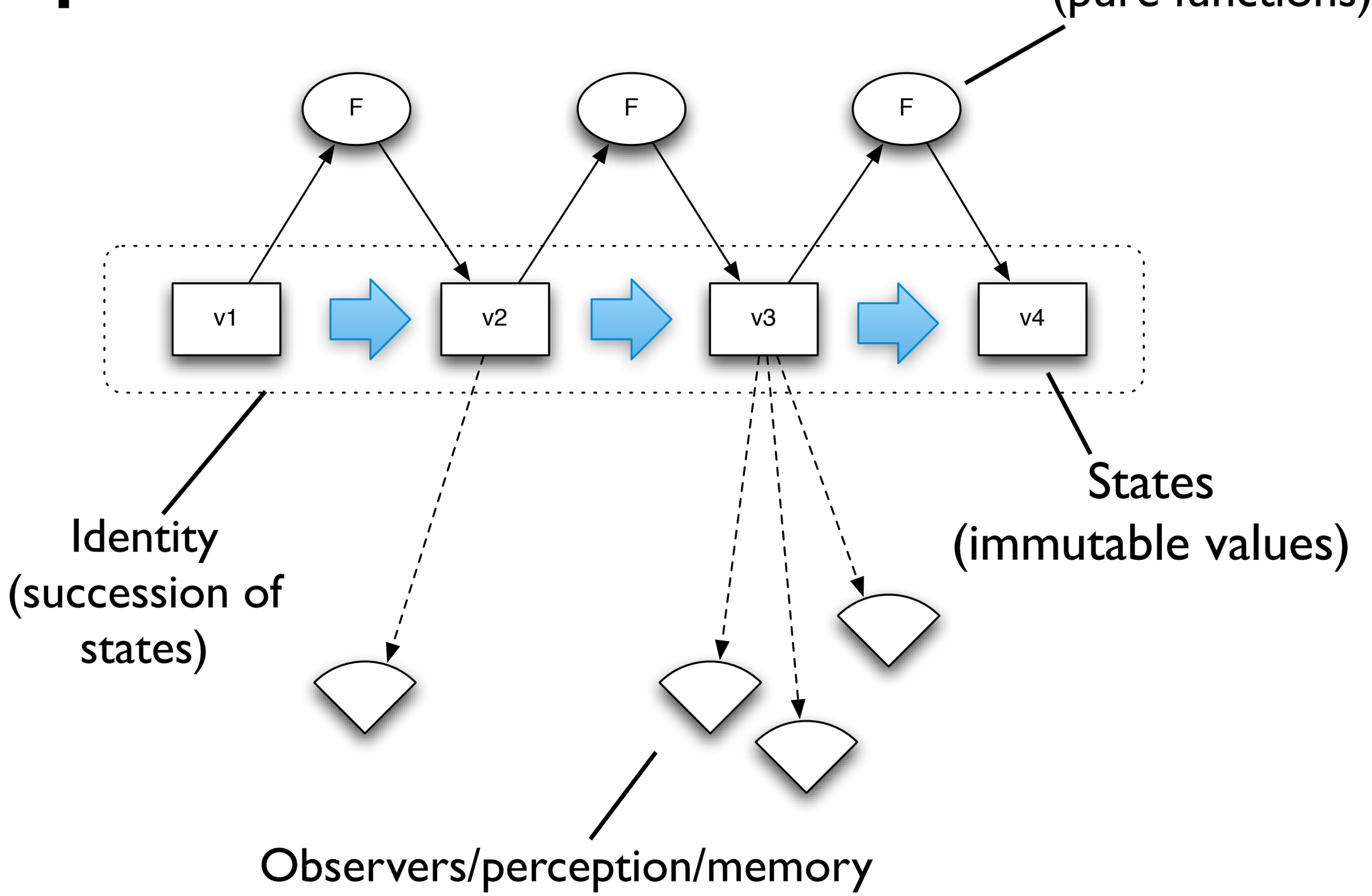


life with variables



perceptions

Epochal Time Model



actions

unified update model

`(`*`change-state`* `ref fn [args*])`

fn gets current state of ref

fn return becomes next state of ref

snapshot always available

no user locking

no deadlocks

writers never impede readers

unified update

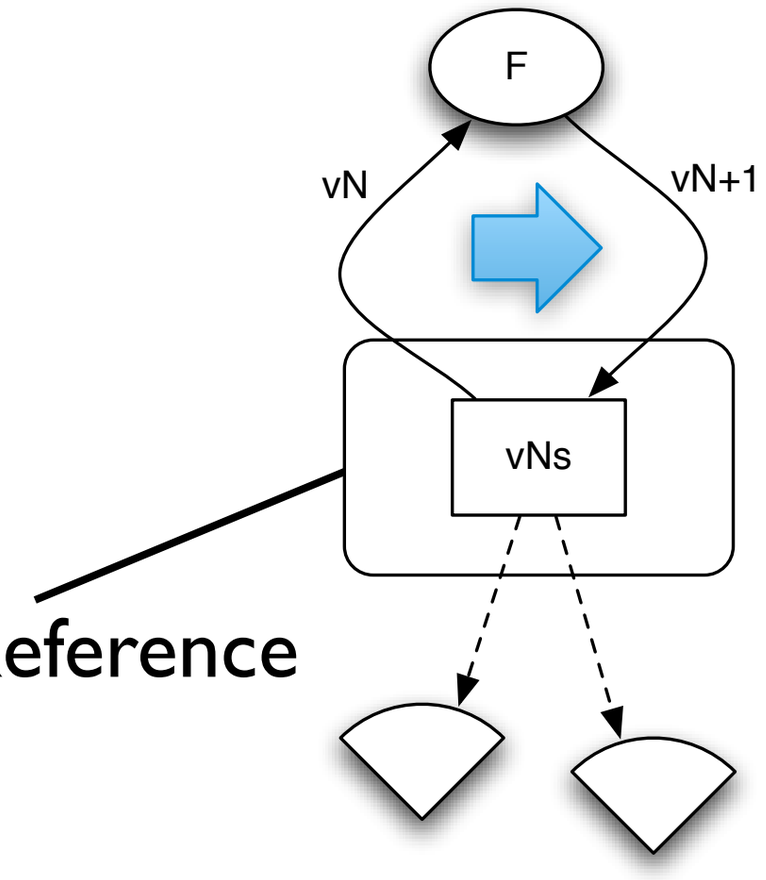
```
;refs
(dosync
  (alter foo  assoc :a "lucy"))

;agents
(send foo assoc :a "lucy")

;atoms
(swap! foo assoc :a "lucy")
```

atoms

cas as time construct

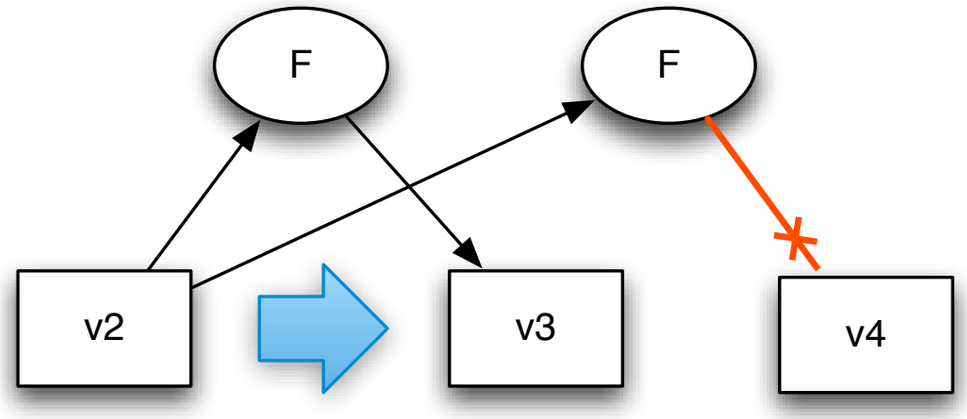


AtomicReference

```
(swap! an-atom f args)
```

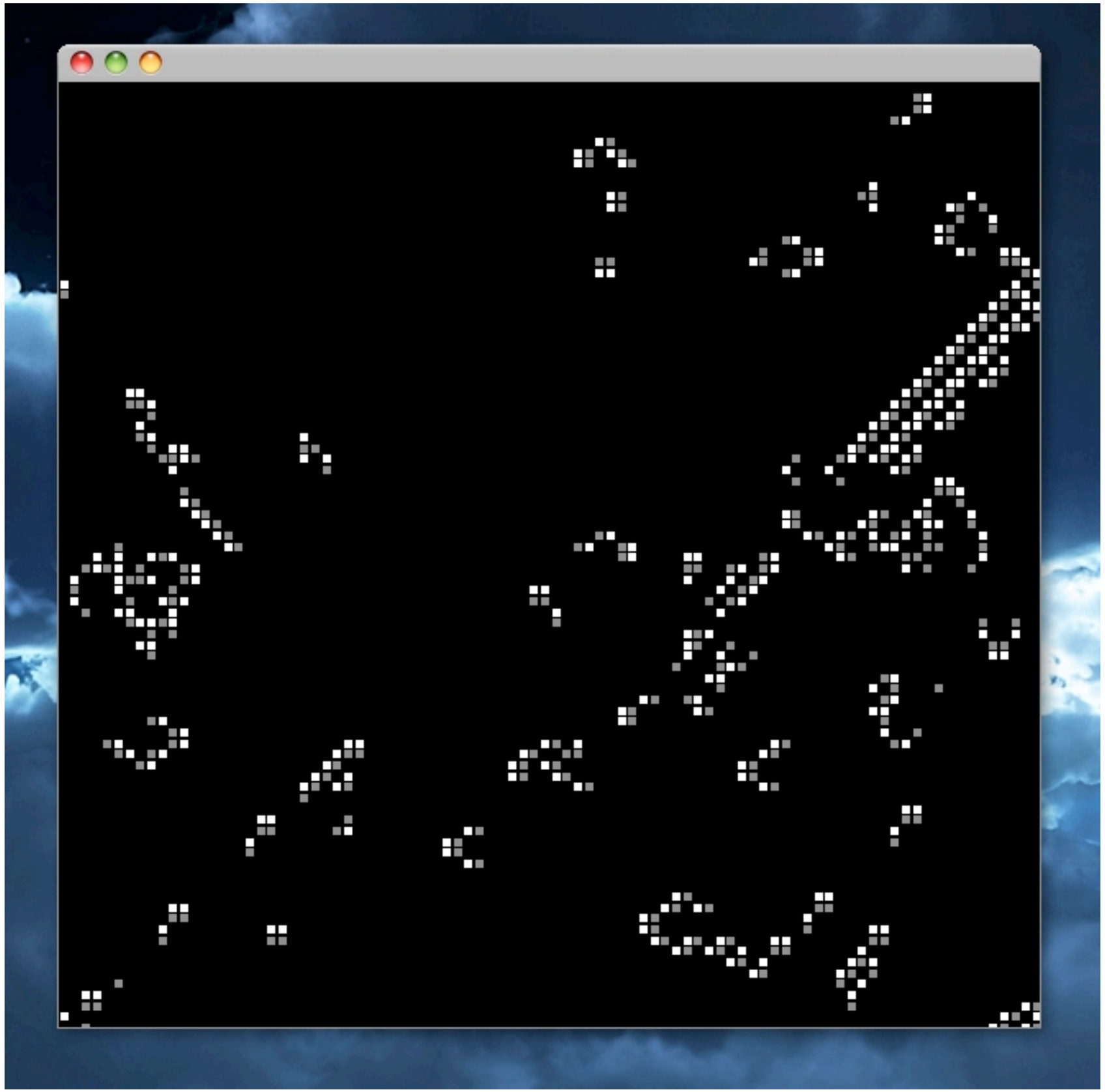
`(f vN args)` *becomes* vN+1

- can automate spin



- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception

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

```
(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))))
```

rules

cursor over previous, me, next

state is trivial

identity

initial value

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

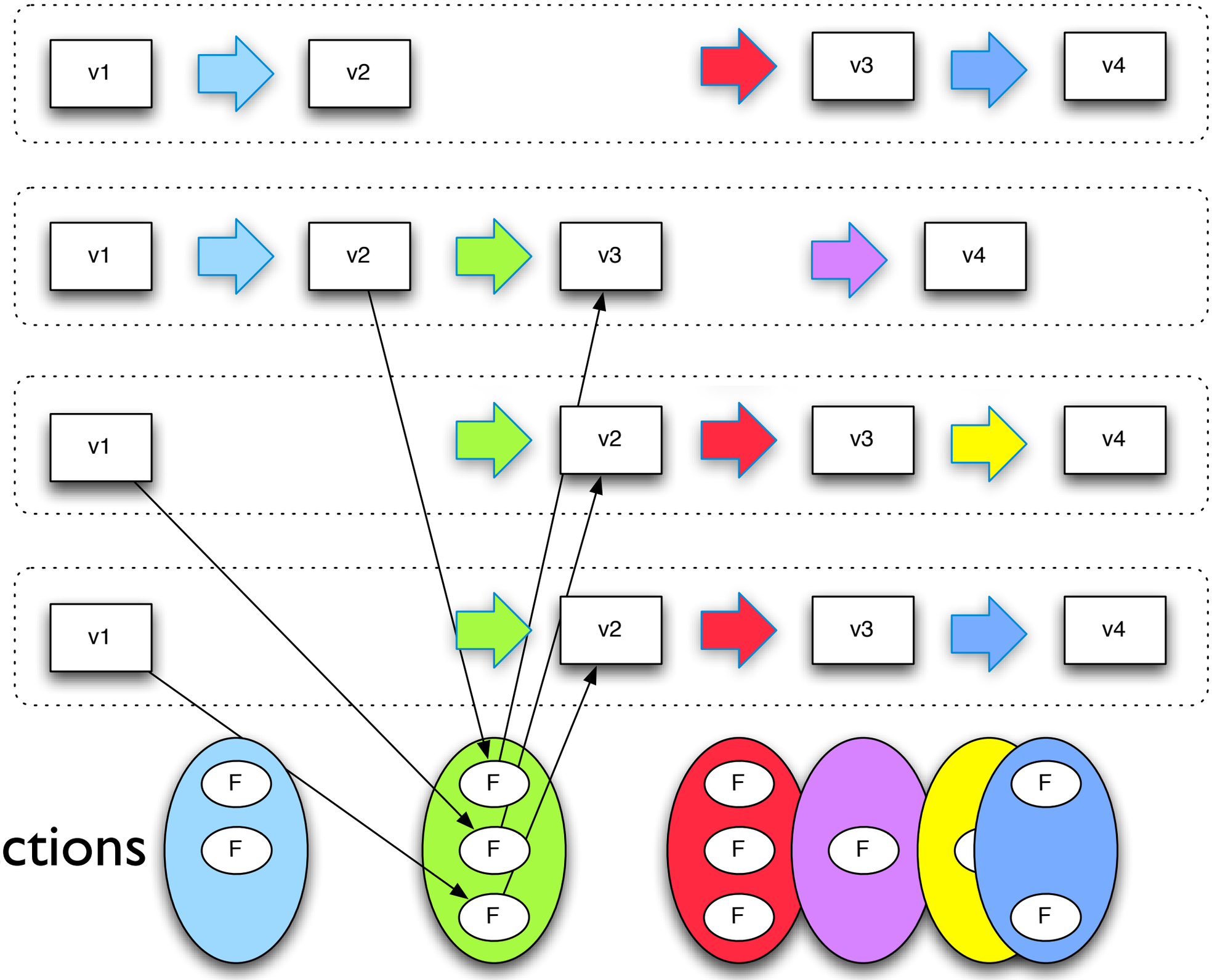
```
(defn update-stage
  "Update the automaton."
  [stage]
  (swap! stage step))
```

apply a fn

update fn

software transactional memory

stm as time construct



Transactions

ref example: chat

identity

```
(def messages (ref []))
```

initial value

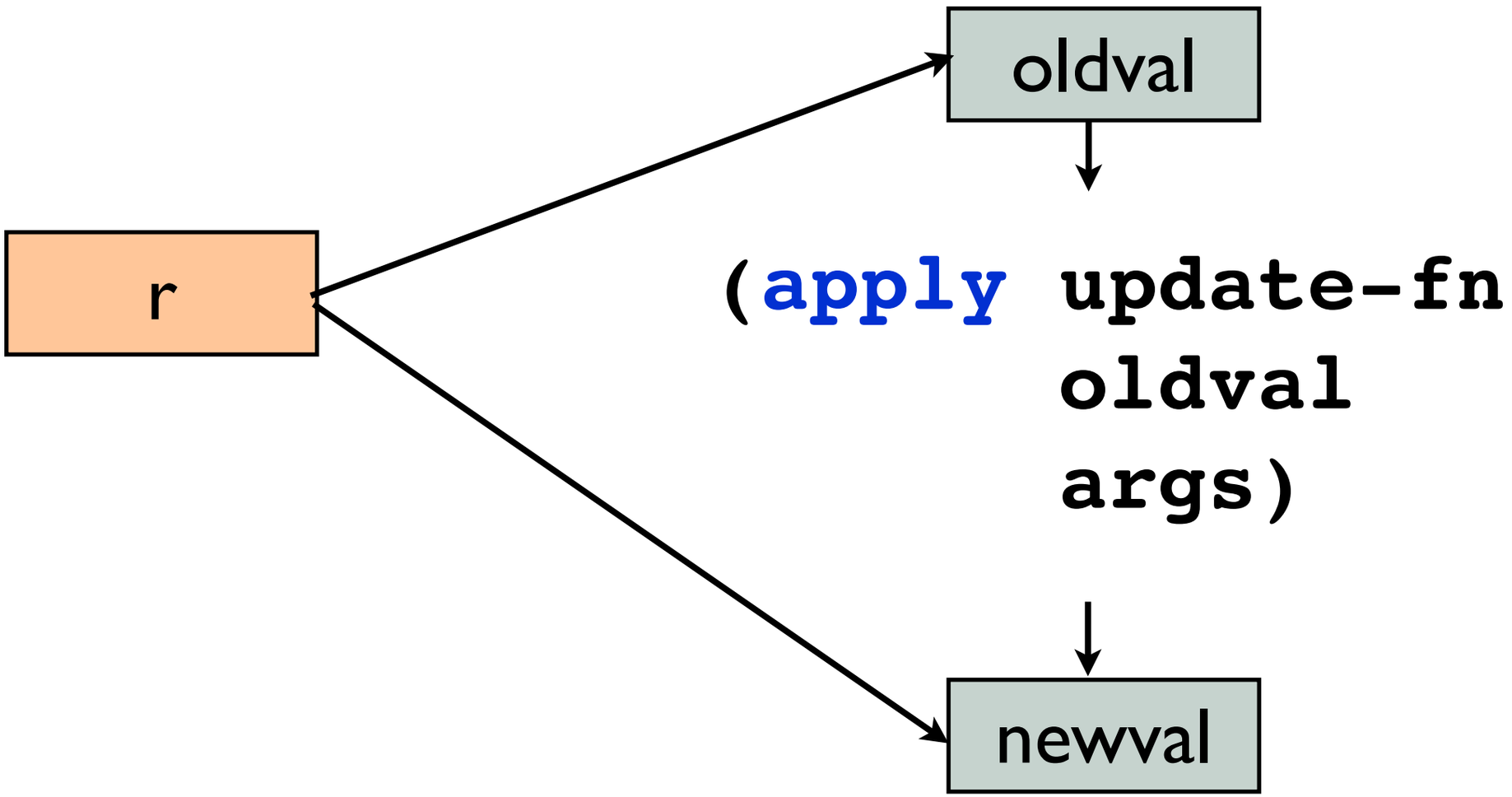
reading value

```
(deref messages)
=> []
```

```
@messages
=> []
```

alter

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



updating

apply an...

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

scope a transaction

...update fn

stms are not all created equal

Clojure's **stm**

not lock free

uses locks, latches, to avoid churn

deadlock detection & barging

no read tracking

readers never impede writers

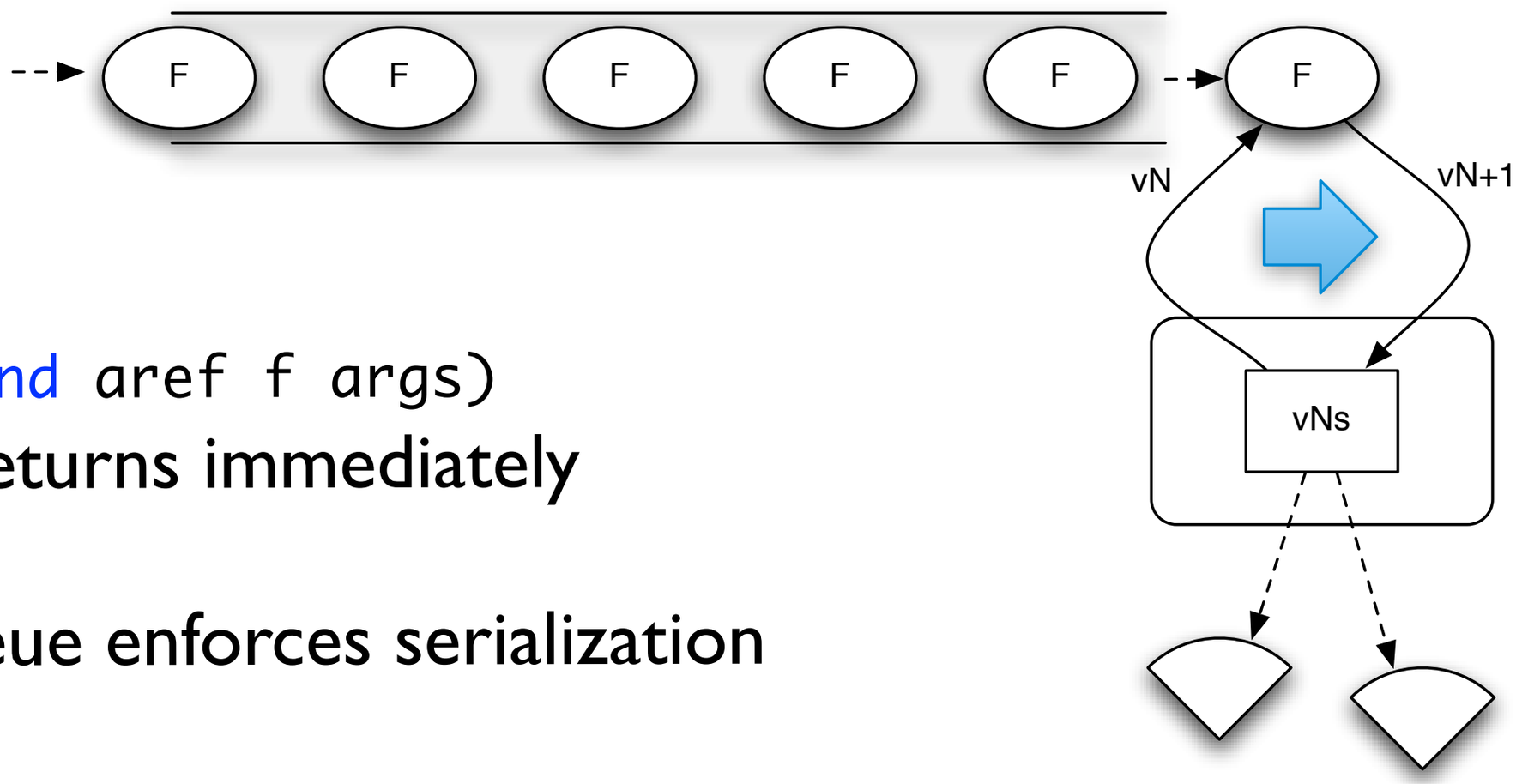
nobody ever impedes readers

commute

ensure

agents

agents as time construct



(send aref f args)

returns immediately

queue enforces serialization

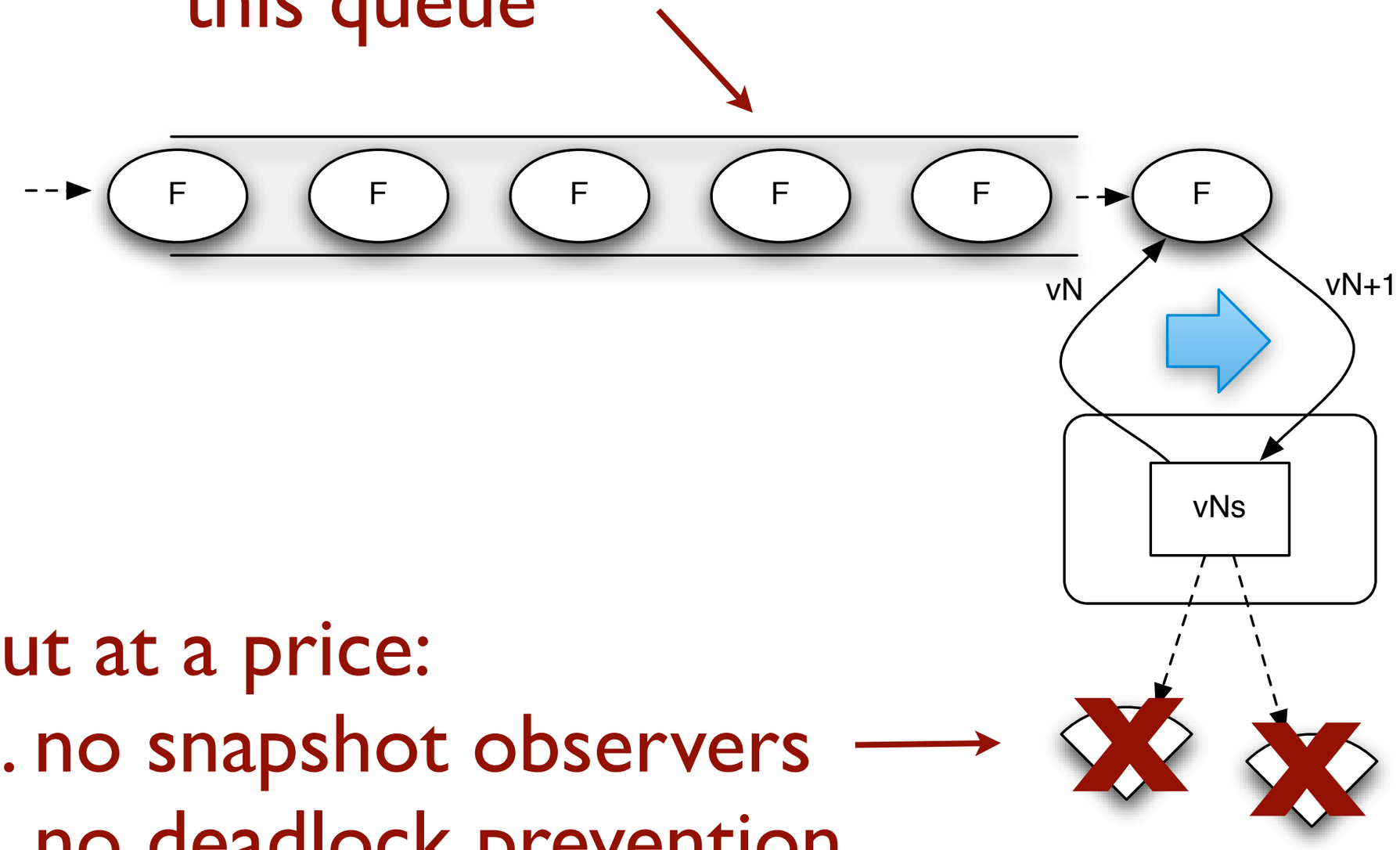
(f vN args) becomes vN+1

happens asynchronously in thread pool thread

- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception

agents are not actors

actors can distribute
this queue



but at a price:

1. no snapshot observers
2. no deadlock prevention

locks | **stm** | actors

is not a useful partition

richer taxonomy

semantics	leverage locality	presume distance
uncoordinated synchronous	atoms	N/A
coordinated synchronous	stm, pods	N/A
uncoordinated asynchronous	agents	actors

53

the devil is in the details

thread ready

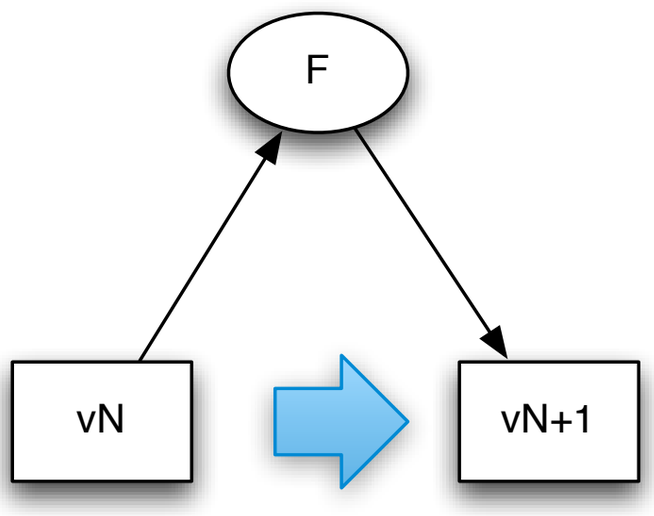
all Clojure constructs work from any thread

- fns are `Callable` and `Runnable`
- use `(future ...)` to throw work to a thread pool

```
(defn fight
  [term1 term2]
  (let [r1 (future (estimated-hits-for term1))
        r2 (future (estimated-hits-for term2))]
    (future {term1 @r1 term2 @r2})))
```

transients

- Persistent data structures *are* slower in sequential use (especially 'writing')
- But - no one can see what happens inside F



- I.e. the 'birthing process' of the next value can use our old (and new) performance tricks:
- Mutation and parallelism
- Parallel map on persistent vector same speed as loop on j.u.ArrayList on quad-core
- Safe 'transient' versions of PDS possible, with O(1) conversions between persistent/transient

use commute
when update
can happen
anytime

not safe for commute

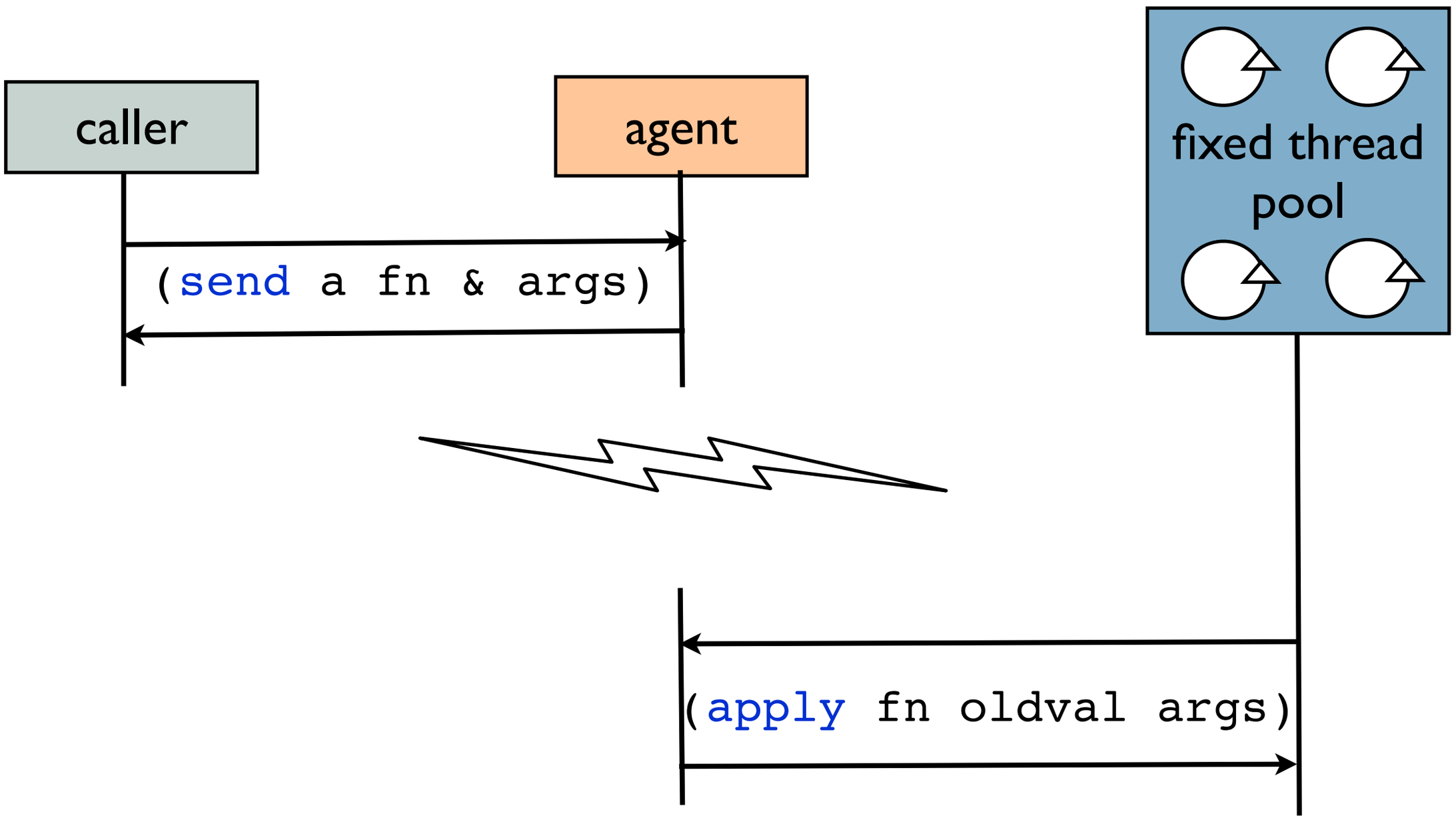
```
(defn next-id
  "Get the next available id."
  []
  (dosync
   (alter ids inc)))
```

safe!

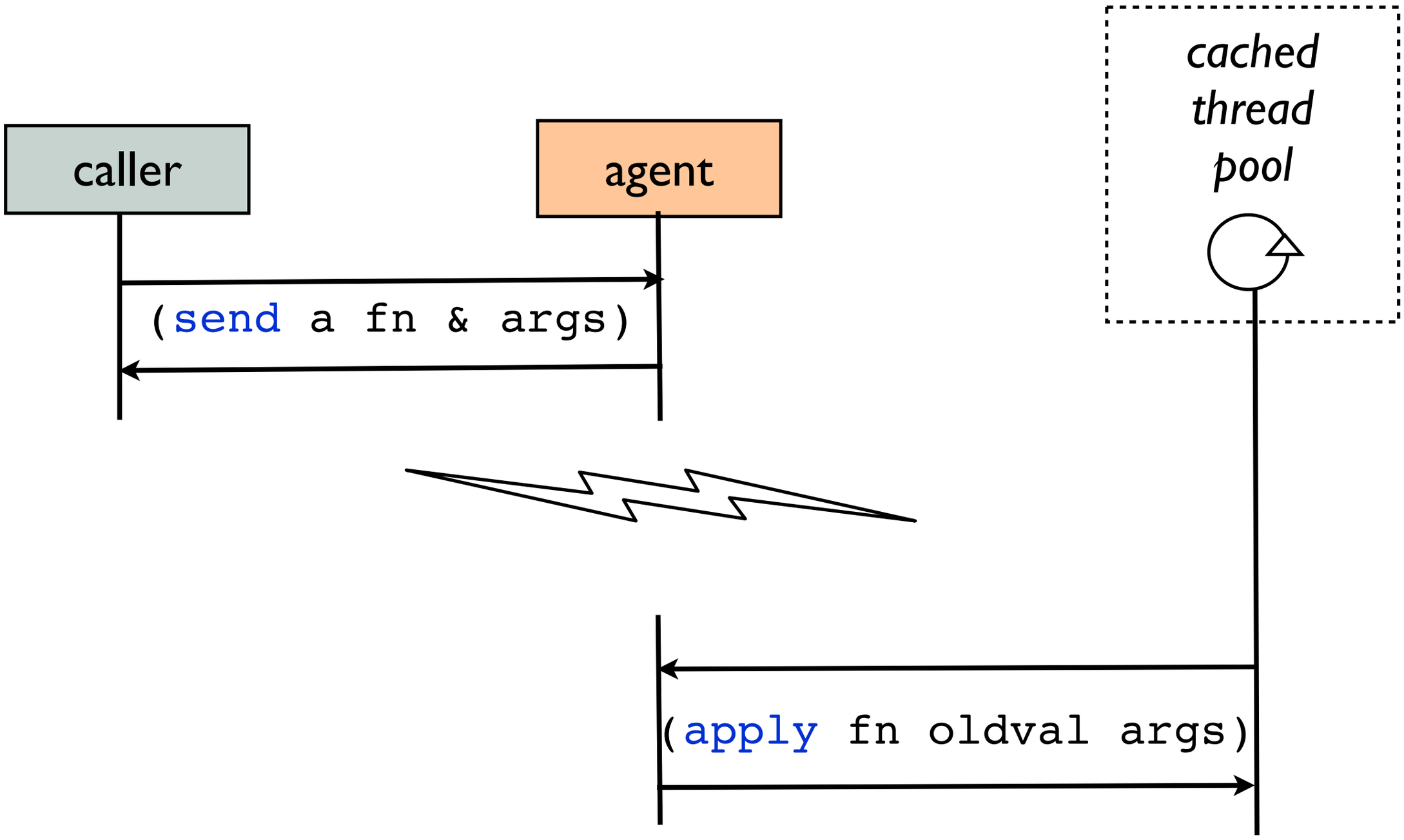
```
(defn increment-counter
  "Bump the internal count."
  []
  (dosync
   (alter ids inc))
  nil)
```

prefer send-off if agent ops might block

send


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

send-off


caller
agent
(send a fn & args)
cached
thread
pool
(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!`**	-

validation

create a function

that checks every item...

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

for some criteria

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

and associate fn with updates to a ref

sending to agents from within 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?

time model beats control model

- resembles reality more closely
- makes design/code/test easier in general

now is a good time to switch

- easier concurrency
- easier parallelism

Clojure's provides an approach that is

- unified
- multi-faceted

69

thanks!

http://clojure.org