State in Clojure

TIMTOWTDI -- Choose Your Poison Carefully

DCBPW 2014
Brock Wilcox
awwaiid@thelackthereof.org

State: Scope

```
foreach my $person (@people) {
  my $home = $person->home_address;
  say $home;
}

say $home; # NO NO!
```

```
(doseq [person people] (
  (let [home (home_address person)]
    (prn home)
  )
  (prn home) ; NOPE!
)

(prn home) ; NO WAY!
```

```
(let [home (home_address person)]
  (prn home)
)

my $home = $person->home_address;
say $home;
```

```
(let [home (home_address person)]
  (prn home)
)

{
  my $home = $person->home_address;
  say $home;
}
```

```
(let [home (home_address person)]
  (prn home)
)

{
  Readonly my $home => $person->home_address;
  say $home;
}
```

State: Immutability

```
(let [newperson (add-address person home)]
  ; ...
)

my $newperson = $person->add_address($home);
```

```
(let [newperson (add-address person home)]
  ; ...
)

{
  my $newperson = $person->add_address($home);
  # ...
}
```

```
(let [newperson (add-address person home)]
  ; ...
)

{
  Readonly my $newperson => $person->add_address($home);
  # ...
}
```

```
(def newperson (add-address person home))
```

```
Readonly our $newperson => $person->add_address($home);
```

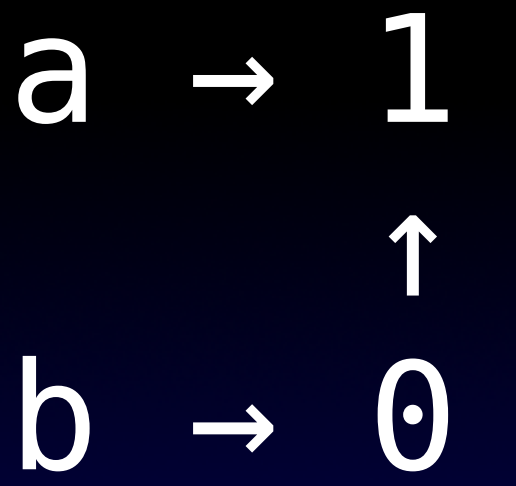
State: Persistence

```
(let [newperson (add-address person home)]
  ; ...
)

{
  my $newperson = $person->add_address($home);
  # ...
}

# Can still use $person!
```

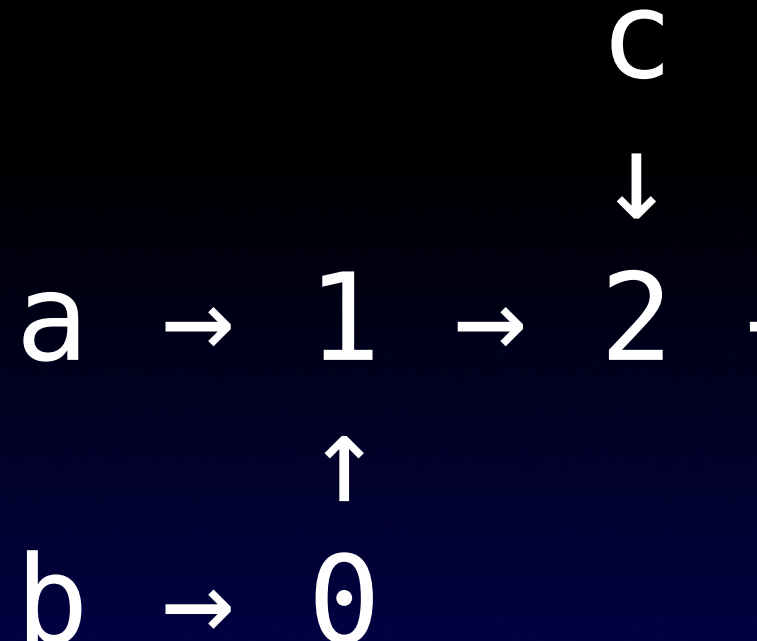
```
(def a (list 1 2 3))
```

a → 1 → 2 → 3

```
(def a (list 1 2 3))
(def b (conj a 0))
```



```
(def a (list 1 2 3))
(def b (conj a 0))
(def c (rest a))
```



Mostly efficient!

Not always.

TIMTOWTDI time!

```
(into #{} (range 5))
;= #{0 1 2 3 4}
```

Cheating: transient collections

You can mark a vector or hash as transient.

Then it is mutable.

Until its marked persistent.

Then it will persist.

```
(into #{} (range 5))
;= #{0 1 2 3 4}

(defn naive-into [coll source]
  (reduce conj coll source))

(defn faster-into [coll source]
  (persistent! (reduce conj! (transient coll) source)))
```

State: Delays, Futures, Promises

Delay

Evaluate on demand, when dereferenced.

```
user=> (def f (delay (+ 30 3)))
#'user/f

user=> f
#<Delay@601c9cb9: :pending>

user=> @f
33
```

Futures

Run in a thread, behave like a delay.

```
user=> (def f (future (Thread/sleep 10000) (println "done") 100))
#'user/f

user=> f
#<core$future_call$reify__6267@4fc64df3: :pending>

user=> done

user=> f
#<core$future_call$reify__6267@4fc64df3: 100>

user=> @f
100
```

pmap

Like map, but uses futures.

Lots of futures.

```
; Before...
(map inc [0 1 2 3 4 5])

; After
(pmap inc [0 1 2 3 4 5])
```

Promises

One time use concurrent mailboxes.

State: Atoms, Refs, Agents

Concurrency / Parallelism

Shared state between threads

Perl: Implicit private, Explicit shared

Clojure: Shared. Scoped. Mostly.

Atoms

Synchronous, uncoordinated, atomic

compare-and-set

Block until modification is complete

```
user=> (def x (atom 7))
#'user/x

user=> @x
7

user=> (reset! x 8)
8

user=> @x
8
```

Kinda rude!

Ignores other work :(

```
user=> (swap! x + 3)
11

user=> @x
11
```

Runs over and over until x is not touched by someone else during execution.

Refs: Software Transactional Memory

alter: like swap! re-runs until nobody else bothers it. Blocks.

commute: applied once in transaction,
non-blocking, then again at the end, non-blocking.

ref-set: ignore current value. like reset!

THE END

References:

Emerick, Carper, Grand "Clojure Programming" O'Reilly

"Clojure for the Brave and True"
http://www.braveclojure.com/

http://clojure.org/state

State in Clojure

TIMTOWTDI -- Choose Your Poison Carefully

DCBPW 2014
Brock Wilcox
awwaiid@thelackthereof.org

BONUS SLIDES

State: Actors

Uncoordinated, Asyncronous

Across processes / machines

Queue of actions. Kinda Erlangy.

Groups actions into memory transacton commits!

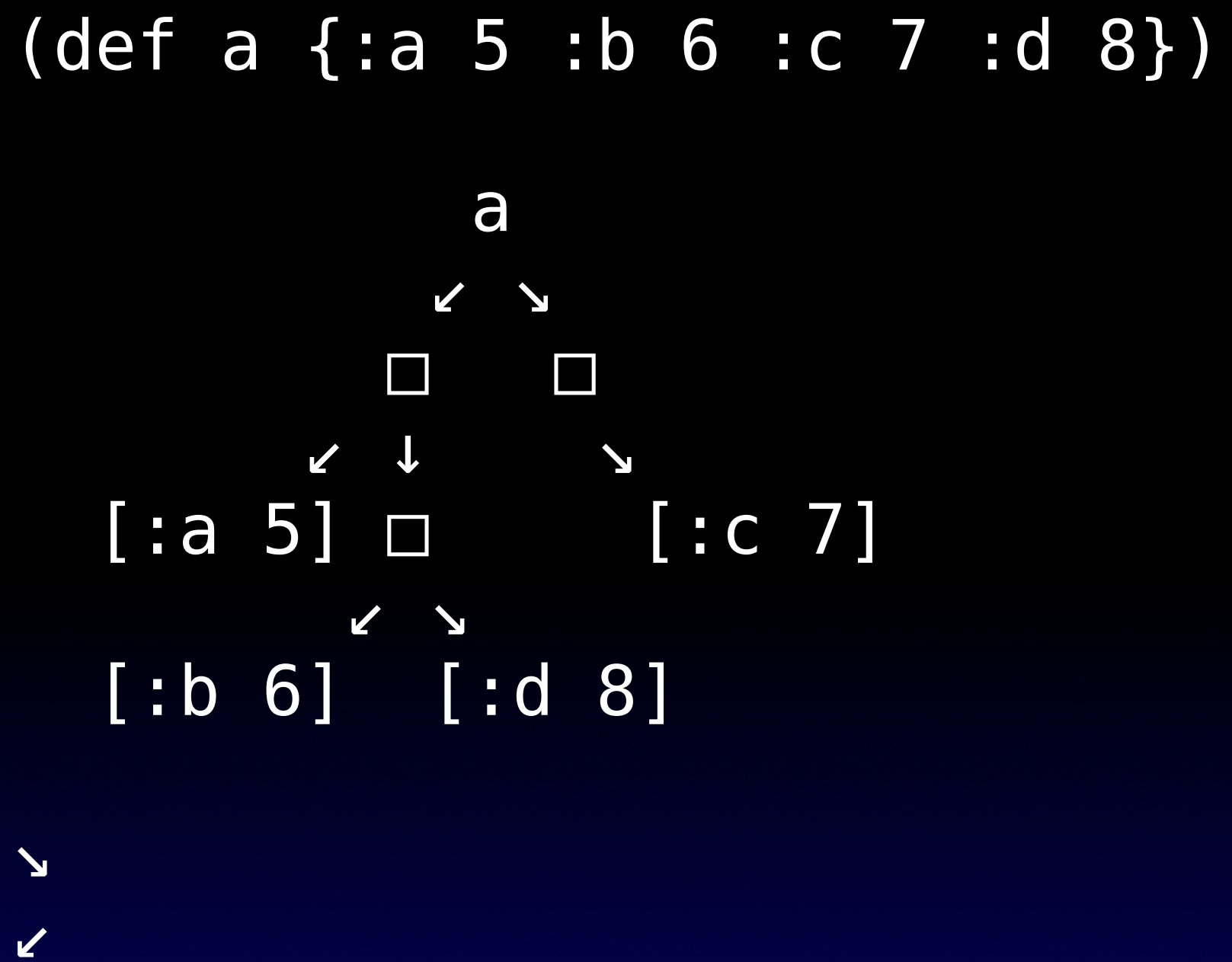
State: Validators, Watches

Validators ... er ... validate stuff

Validators run before modification

Watches ... er ... watch stuff

Watches run after modification


(def a {:a 5 :b 6 :c 7 :d 8})
a
[:a 5]
[:c 7]
[:b 6]
[:d 8]

State in a Multithreaded World

```
use threads;
use threads::shared;

my ($laser) :shared;
```

Perl: Implicit private, Explicit shared

Clojure: Shared.

Note that perl does implicitly share subs
and modules

Clojure shares everything -- defs, namespaces, etc

atom

atom: swap!

atom: @val

ref

agent

State in a Mulitprocess World