Clojure Workshop Slides

JVM Language Summit 2009

Rich Hickey

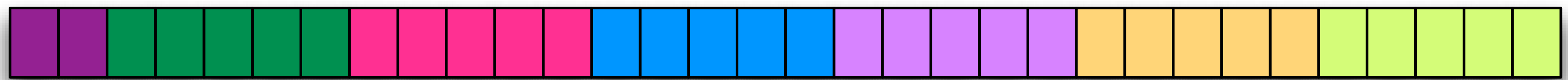
Notes

- These were the slides shown during the Clojure workshop
- Not a complete talk

Persistent Data Structures

- Composite values - immutable
- ‘Change’ is merely a function, takes one value and returns another, ‘changed’ value
- Collection maintains its performance guarantees
  - Therefore new versions are not full copies
- Old version of the collection is still available after 'changes', with same performance
- Example - hash map/set and vector based upon array mapped hash tries (Bagwell)

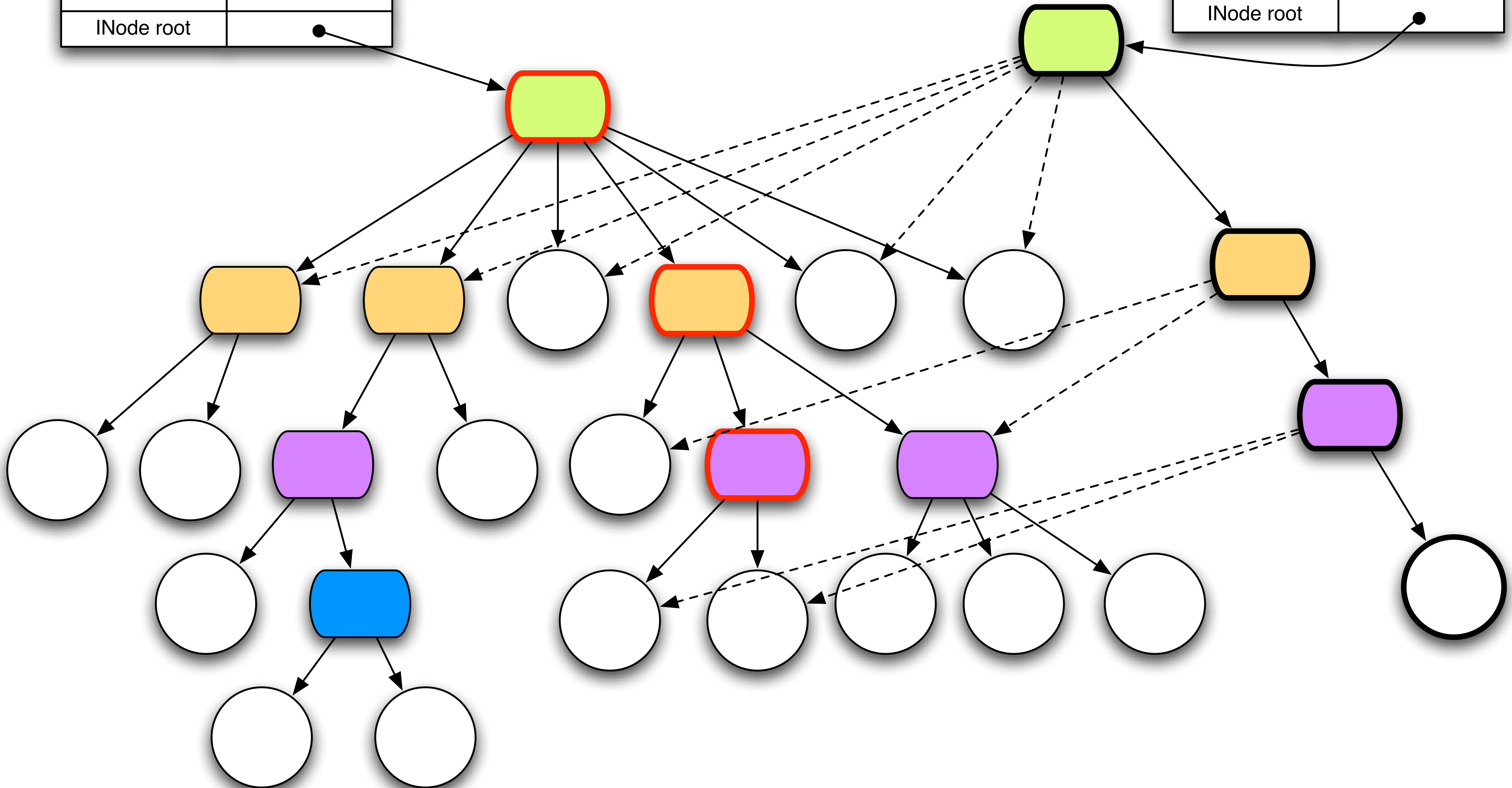
Bit-partitioned hash tries



Structural Sharing

- Key to efficient ‘copies’ and therefore persistence
- Everything is immutable so no chance of interference
- Thread safe
- Iteration safe

Path Copying



Transients

If a pure function mutates some local data in order to produce an immutable return value, is that ok?

Rich Hickey

`assoc` mutates internally

- Creates new array and bangs on it
- Necessary for performance
- No one can see it
- Once published, never changes
- assoc is still pure

What about larger changes?

- Multi-step, multi-collection units of work
- Still functional at the endpoints
- Seen often in user code
- How to offer same write-once capability?

Usage and properties

- Write your code functionally first
- Create a transient data structure from a persistent one - `(transient coll)`
    - O(1) - not a full copy
- Change assoc, conj etc to assoc!, conj!
- When finished, call `(persistent! trans-coll)`
    - O(1) - not a full copy
- Subsequent use of transient will throw

More properties

- vectors, hash-maps supported
  - i.e., not lists, no benefit
- Doesn't modify source, still safe and immutable
- read-only interface still supported, but assoc etc will throw
- assoc!, conj! do same things as assoc conj
  - must be used in a single path of control
  - not change-in-place

Same code structure

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

(defn vrange2 [n]
  (loop [i 0 v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))))

(time (def v (vrange 1000000)))
"Elapsed time: 297.444 msecs"

(time (def v2 (vrange2 1000000)))
"Elapsed time: 34.428 msecs"
```

Transients enforce thread isolation

- Any use from a thread other than the creating thread will throw
- Composite operations are ok, and require no locks
  - contrast with java.util.Vector
- Multi-collection operations are ok, and require no locks
  - contrast with any other j.u.Collection
- Aliasing or leakage will be caught

Parallelism in core structures

```
(defn pvmap [f #^PersistentVector v]
  (let [new-node #(PersistentVector$Node. (.. v root edit) %)
        new-root (fjvtree v
                   #(new-node (to-array %))
                   #(new-node
                     (amap (.array #^PersistentVector$Node %) i a
                           (f (aget a i)))))
        new-tail (to-array (map f (.tail v)))]
    (PersistentVector. (.cnt v) (.shift v) new-root new-tail)))
```

```clojure
(defn- fjvtree [#^PersistentVector v combine-fn leaf-fn]
  (let [tfn (fn tfn [shift #^PersistentVector$Node node]
              (fjtask
               (let [nodes (remove nil? (.array node))]
                 (if (= shift PersistentVector/SHIFT)
                   (let [lts (reduce #(cons (doto (fjtask (leaf-fn %2)) .fork) %1)
                                     () nodes)]
                     (combine-fn (reduce (fn [rets #^FJTask t]
                                           (cons (if (.tryUnfork t)
                                                   (.compute t)
                                                   (do (.join t) (.get t))) rets))
                                         () lts)))
                   (let [tasks (map #(tfn (- shift PersistentVector/SHIFT) %) nodes)]
                     (ForkJoinTask/invokeAll #^Collection tasks)
                     (combine-fn (map #(.get #^ForkJoinTask %) tasks)))))))
        task #^ForkJoinTask (tfn (.shift v) (.root v))]
    (if (ForkJoinTask/getPool) ;nested task
      (.invoke task)
      (.invoke pool task))))
```

Clojure References

- The only things that mutate are references themselves, in a controlled way
- 4 types of mutable references, with different semantics:
  - Refs - shared/synchronous/coordinated
  - Agents - shared/asynchronous/autonomous
  - Atoms - shared/synchronous/autonomous
  - Vars - Isolated changes within threads

Uniform state transition model

- ('change-state' reference function [args*])
- function will be passed current state of the reference (plus any args)
- Return value of function will be the next state of the reference
- Snapshot of 'current' state always available with deref
- No user locking, no deadlocks

Persistent 'Edit'

foo

@foo

:a	"fred"
:b	"ethel"
:c	42
:d	17
:e	6

:a	"lucy"
:b	"ethel"
:c	42
:d	17
:e	6

- New value is function of old
- Shares immutable structure
- Doesn't impede readers
- Not impeded by readers

Atomic State Transition

foo

@foo

:a	"fred"
:b	"ethel"
:c	42
:d	17
:e	6

:a	"lucy"
:b	"ethel"
:c	42
:d	17
:e	6

- Always coordinated
  - Multiple semantics
- Next dereference sees new value
- Consumers of values unaffected

Refs and Transactions

- Software transactional memory system (STM)
- Refs can only be changed within a transaction
- All changes are Atomic and Isolated
  - Every change to Refs made within a transaction occurs or none do
  - No transaction sees the effects of any other transaction while it is running
- Transactions are speculative
  - Will be retried automatically if conflict
  - Must avoid side-effects!

The Clojure STM

- Surround code with (dosync ...), state changes through *alter/commute*, using ordinary function (state=>new-state)
- Uses Multiversion Concurrency Control (MVCC)
- All reads of Refs will see a consistent snapshot of the 'Ref world' as of the starting point of the transaction, + any changes it has made.
- All changes made to Refs during a transaction will appear to occur at a single point in the timeline.

Refs in action

```
(def foo (ref {:a "fred" :b "ethel" :c 42 :d 17 :e 6}))

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(assoc @foo :a "lucy")
-> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(alter foo  assoc :a "lucy")
-> IllegalStateException: No transaction running

(dosync (alter foo  assoc :a "lucy"))
@foo -> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}
```

Implementation - STM

- Not a lock-free spinning optimistic design
- Uses locks, wait/notify to avoid churn
- Deadlock detection + barging
- One timestamp CAS is only global resource
- No read tracking
- Coarse-grained orientation
  - Refs + persistent data structures
- Readers don't impede writers/readers, writers don't impede readers, supports commute

STM - commute

- Often a transaction will need to update a jobs-done counter or add its result to a map
- If done with `alter`, update is a read-modify-write, so if multiple transactions contend, one wins, one retries
- If transactions don't care about resulting value, and operation is commutative, can instead use `commute`
- Both transactions will succeed without retry
- Always just an optimization

STM - ensure

- MVCC is subject to *write-skew*
  - Where validity of transaction depends on stability of value unchanged by it
  - e.g. one of two accounts can go negative but not both
- Simply reading does not preclude modification by another transaction
- Can use ensure for values that are read but must remain stable
- More efficient than dummy write