Clojure

A Dynamic Programming Language for the JVM

Concurrency Support

Rich Hickey

Agenda

- Introduction
- Feature Tour
- Shared state, multithreading and locks
- Refs, Transactions, and Agents
- Walkthrough - Multithreaded ant colony simulation
- Q&A

Introduction

- Who are you?
  - Know/use Lisp?
  - Java/C#/Scala?
  - ML/Haskell?
  - Python, Ruby, Groovy?
  - Clojure?
- Any multithreaded programming?

Clojure Fundamentals

- Functional
  - Immutable, persistent data structures
  - No mutable local variables
- Lisp
  - *Not* CL or Scheme
- Hosted on, and embracing, the JVM
- Supporting Concurrency
- Open Source

Clojure Features

- Dynamic development
  - REPL, reader, on-the-fly compilation to JVM bytecode
- Primitives - numbers, including arbitrary-precision integers & ratios, characters, strings, symbols, keywords, regexes
- Aggregates - lists, maps, sets, vectors
  - read-able, persistent, immutable, extensible
- Abstract sequences + library

Clojure Features

- Metadata
- First-class functions (*fn*), closures
- Recursive functional looping
- Destructuring binding in *let/fn/loop*
- List comprehensions (*for*)
- Macros
- Multimethods
- Concurrency support

Clojure Features

- Java interop
  - Call methods, access fields, arrays
  - Proxy interfaces/classes
  - Sequence functions extended to Java strings, arrays, Collections
  - Clojure data structures implement Collection/Callable/Iterable/Comparable etc where appropriate
- Namespaces, zippers, XML and more!

State - You're doing it wrong

- Mutable objects are the new spaghetti code
  - Hard to understand, test, reason about
  - Concurrency disaster
  - Terrible default architecture (Java/C#/Python/Ruby/Groovy/CLOS...)
- Doing the right thing is very difficult
  - Languages matter!

Concurrency

- Interleaved/simultaneous execution
- Must avoid seeing/yielding inconsistent data
- The more components there are to the data, the more difficult to keep consistent
- The more steps in a logical change, the more difficult to keep consistent
- Opportunities for automatic parallelism
    - Emphasis here on coordination

Explicit Locks

- lock/synchronized(coll){...}
- Only one thread can have the lock, others block
- Requires coordination
  - All code that performs non-atomic access to coll must put that in a lock block
  - Synchronized handles single-method jobs only

Single Lock Problems

Single Lock Problems

- Can’t enforce coordination via language/code

Single Lock Problems

- Can’t enforce coordination via language/code
  - This is not a small problem

Single Lock Problems

- Can’t enforce coordination via language/code
  - This is not a small problem
- Even when correct, can cause throughput bottleneck on multi-CPU machines

Single Lock Problems

- Can’t enforce coordination via language/code
    - This is not a small problem
- Even when correct, can cause throughput bottleneck on multi-CPU machines
    - Your app ***<u>is</u>*** running on a multi-CPU machine

Single Lock Problems

- Can’t enforce coordination via language/code
  - This is not a small problem
- Even when correct, can cause throughput bottleneck on multi-CPU machines
  - Your app ***is*** running on a multi-CPU machine
  - Readers block readers

Enhancing Read Parallelism

Enhancing Read Parallelism

- Multi-reader/single-writer locks
  - Readers don't block each other
  - One writer at a time
  - Writers wait for reader(s)

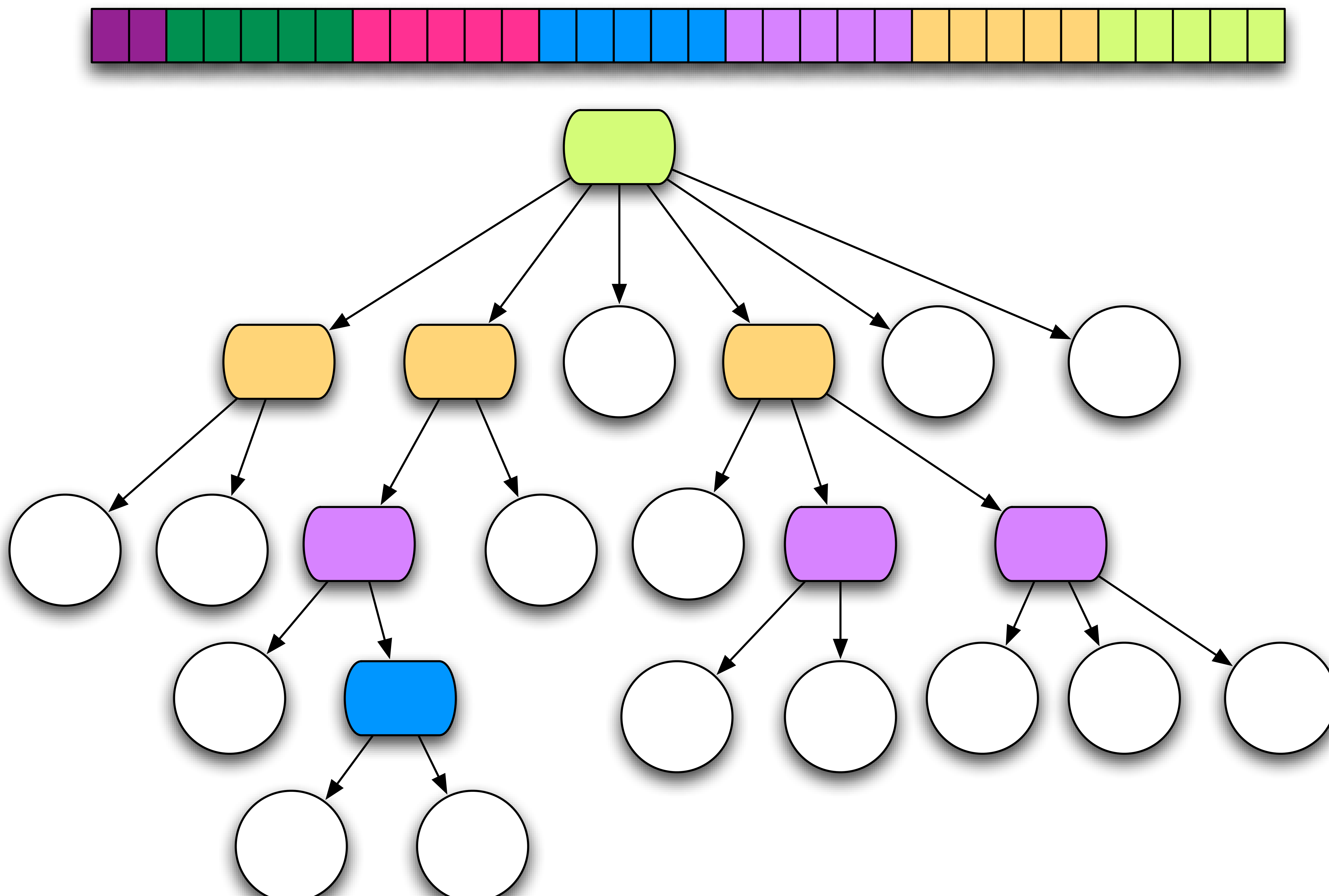
CopyOnWrite Collections

- Reads get a snapshot
- Lock-free reading
- Atomic writes
- Internally, copy the representation and swap it
  - Writes can be expensive (copying)
- Multi-step writes still require locks

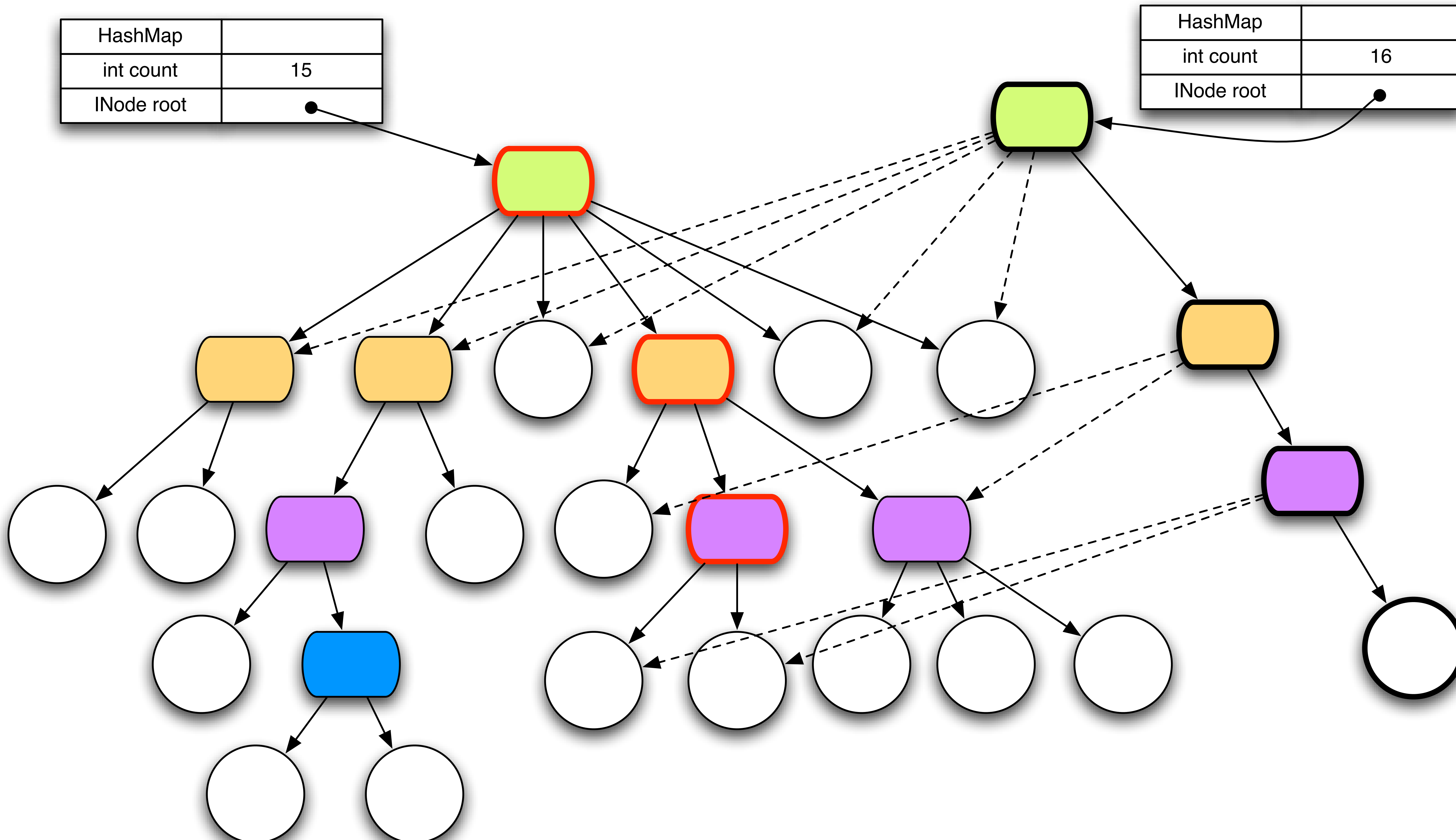
Persistent Data Structures

- Immutable, + old version of the collection is still available after 'changes'
- Collection maintains its performance guarantees for most operations
  - Therefore new versions are not full copies
- All Clojure data structures persistent
  - Hash map and vector both based upon array mapped hash tries (Bagwell)
  - Sorted map is red-black tree

Bit-partitioned hash tries



Path Copying



Structural Sharing

- Key to efficient 'copies' and therefore persistence
- Everything is final so no chance of interference
- Thread safe
- Iteration safe

Multi-component change

- Preceding was the easy part
- Many logical activities involve multiple data structures/multiple steps
- Two locking options
  - Coarse granularity locks
  - Fine granularity locks

Coarse Granularity Locking

- Create external Lock representing a set of data structures
- Clients must obtain a lock to manipulate **<u>any</u>** of the structures
- Each multi-part logical operation requires only one lock

Coarse Granularity Locking

Coarse Granularity Locking

- ✓ Safest

Coarse Granularity Locking

✓ Safest

✴ Can be confusing as to what constitutes the set(s), what needs to be locked

- X needs a/b/c, Y needs b/c/d

Coarse Granularity Locking

✓ Safest

✴ Can be confusing as to what constitutes the set(s), what needs to be locked

- X needs a/b/c, Y needs b/c/d

✴ Least throughput

- Possible needless blocking

Coarse Granularity Locking

✓ Safest

✴ Can be confusing as to what constitutes the set(s), what needs to be locked

- X needs a/b/c, Y needs b/c/d

✴ Least throughput

- Possible needless blocking

✴ Should reads lock?

Fine Granularity Locking

- Use locks on data structures themselves
- Clients must obtain a lock on each of the structures
- A multi-part logical operation may require several locks

Fine Granularity Locking

Fine Granularity Locking

- Dangerous

Fine Granularity Locking

* Dangerous
* Locking order is critical
    * X locks a/b, Y locks b/a - deadlock possible
    * Very difficult to enforce locking order

Fine Granularity Locking

- ✴ Dangerous
- ✴ Locking order is critical
  - X locks a/b, Y locks b/a - deadlock possible
  - Very difficult to enforce locking order
- ✓ Best throughput
  - Minimal blocking

Fine Granularity Locking

- Dangerous
- Locking order is critical
  - X locks a/b, Y locks b/a - deadlock possible
  - Very difficult to enforce locking order
- ✓ Best throughput
  - Minimal blocking
- Should reads lock?

Concurrency Methods

- Conventional way:
  - Direct references to mutable objects
  - Lock and pray (manual/convention)
- Clojure way:
  - Indirect references to immutable persistent data structures
  - Concurrency semantics for references
    - Automatic/enforced
    - <u>No locks!</u>

Clojure References

- The only things that mutate are references themselves, in a controlled way
- 3 types of mutable references
  - Vars - Isolate changes within threads
  - Refs - Share synchronous coordinated changes between threads
  - Agents - Share asynchronous independent changes between threads

Vars

- Like Common Lisp's special vars
  - dynamic scope
  - stack discipline
- Shared root binding established by *def*
  - root can be unbound
- Can be changed (via *set!*) but only if first thread-locally bound using *binding*
- Functions stored in vars, so they too can be dynamically rebound
  - context/aspect-like idioms

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 ...)
- 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.
- Readers never block writers/readers, writers never block readers, supports commute

Agents

- Manage independent state
- State changes through actions, which are ordinary functions (state=>new-state)
- Actions are dispatched using *send* or *send-off*, which return immediately
- Actions occur asynchronously on thread-pool threads
- Only one action per agent happens at a time

Agents

- Agent state always accessible, via deref/@, but may not reflect all actions
- Can coordinate with actions using *await*
- Any dispatches made during an action are held until *after* the state of the agent has changed
- Agents coordinate with transactions - any dispatches made during a transaction are held until it commits
- Agents are not Actors (Erlang/Scala)

Walkthrough

- Ant colony simulation
- World populated with food and ants
- Ants find food, bring home, drop pheromones
- Sense pheromones, food, home
- Ants act independently, on multiple real threads
- Model pheromone evaporation
- Animated GUI
- < 250 lines of Clojure

Thanks for listening!

http://www.clojure.org