Persistent Data Structures and Managed References

Clojure’s approach to Identity and State

Rich Hickey
Agenda

• Functions and processes
• Identity, State, and Values
• Persistent Data Structures
• Clojure’s Managed References
• Q&A
Clojure Fundamentals

- Dynamic
- Functional
  - emphasis on immutability
- Supporting Concurrency
- Hosted on the JVM
  - Compiles to JVM bytecode
- Not Object-oriented
- Ideas in this talk are not Clojure-specific
Functions

• Function
  • Depends only on its arguments
  • Given the same arguments, always returns the same value
  • Has no effect on the world
  • Has no notion of time
Functional Programming

• Emphasizes functions
• Tremendous benefits
• But - most programs are not functions
  • Maybe compilers, theorem provers?
    • But - They execute on a machine
  • Observably consume compute resources
Processes

• Include some notion of change over time
• Might have effects on the world
• Might wait for external events
• Might produce different answers at different times (i.e. have state)
• Many real/interesting programs are processes
• This talk is about one way to deal with state and time in the local context
State

• Value of an identity at a time

• Sounds like a variable/field?
  • Name that takes on successive ‘values’

• Not quite:
  • i = 0
  • i = 42
  • j = i
  • j is 42? - depends
Variables

- Variables (and fields) in traditional languages are predicated on a single thread of control, one timeline
- Adding concurrency breaks them badly
  - Non-atomicity (e.g. of longs)
  - volatile, write visibility
- Composite operations require locks
- All workarounds for lack of a time model
Time

- When things happen
- Before/after
- Later
- At the same time (concurrency)
- Now
- Inherently relative
Value

- An immutable magnitude, quantity, number... or composite thereof
- 42 - easy to understand as value
- But traditional OO tends to make us think of composites as something other than values
  - Big mistake
    - aDate.setMonth("January") - ugh!
- Dates, collections etc are all values
Identity

• A logical entity we associate with a series of causally related values (states) over time

• Not a name, but can be named
  • I call my mom ‘Mom’, but you wouldn’t

• Can be composite - the NY Yankees

• Programs that are processes need identity
State

• Value of an identity at a time

• Why not use variables for state?
  • Variable might not refer to a proper value
  • Sets of variables/fields never constitute a proper composite value
  • No state transition management
    • I.e., no time coordination model
Philosophy

• Things don't change in place

• Becomes obvious once you incorporate time as a dimension

• Place includes time

• The future is a function of the past, and doesn’t change it

• Co-located entities can observe each other without cooperation

• Coordination is desirable in local context
Race-walker foul detector

- Get left foot position
  - off the ground
- Get right foot position
  - off the ground
- Must be a foul, right?
• Snapshots are critical to perception and decision making

• Can’t stop the runner/race (locking)

• Not a problem if we can get runner’s value

• Similarly don’t want to stop sales in order to calculate bonuses or sales report
Approach

• Programming with values is critical

• By eschewing morphing in place, we just need to manage the succession of values (states) of an identity

• A timeline coordination problem
  • Several semantics possible

• Managed references
  • Variable-like cells with coordination semantics
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

<table>
<thead>
<tr>
<th>HashMap</th>
<th>int count</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>INode root</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HashMap</th>
<th>int count</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>INode root</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coordination Methods

• Conventional way:
  • Direct references to mutable objects
  • Lock and worry (manual/convention)

• Clojure way:
  • Indirect references to immutable persistent data structures (inspired by SML’s \texttt{ref})
  • Concurrency semantics for references
    • Automatic/enforced
    • \textbf{No locks in user code!}
Typical OO - Direct references to Mutable Objects

- Unifies identity and value
- Anything can change at any time
- Consistency is a user problem
- Encapsulation doesn’t solve concurrency problems
Clojure - Indirect references to Immutable Objects

- Separates identity and value
- Obtaining value requires explicit dereference
- Values can never change
- Never an inconsistent value
- Encapsulation is orthogonal
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’

- New value is function of old
- Shares immutable structure
- Doesn’t impede readers
- Not impeded by readers
Atomic State Transition

- 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}

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

(dosync (commute 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
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

• 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)
Agents in Action

```
(def foo (agent {:a "fred" :b "ethel" :c 42 :d 17 :e 6}))
@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}
(send foo assoc :a "lucy")
@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}
... time passes ...
@foo -> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}
```
Atoms

- Manage independent state
- State changes through `swap!`, using ordinary function `(state=>new-state)`
- Change occurs *synchronously* on caller thread
- Models compare-and-set (CAS) spin swap
- Function may be called more than once!
  - Guaranteed atomic transition
  - Must avoid side-effects!
Atoms in Action

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

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

(swap! foo assoc :a "lucy")

@foo -> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}
Uniform state transition

;refs
(dosync
  (commute foo assoc :a "lucy"))

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

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

- Immutable values, a feature of the functional parts of our programs, are a critical component of the parts that deal with time.
- Persistent data structures provide efficient immutable composite values.
- Once you accept immutability, you can separate time management, and swap in various concurrency semantics.
- Managed references provide easy to use and understand time coordination.
Thanks for listening!

http://clojure.org

Questions?