Clojure

A Dynamic Programming Language for the JVM

Rich Hickey

Clojure Fundamentals

- 3 years in development, released 10/2007
- A new Lisp, not Common Lisp or Scheme
- Functional
  - emphasis on immutability
- Supporting Concurrency
  - language-level coordination of state
- Designed for the JVM
  - exposes and embraces platform

Clojure is a Lisp

- Dynamically typed, dynamically compiled
- Interactive - REPL
- Load/change code in running program
- Code as data - Reader
- Small core
- Sequences
- Syntactic abstraction - macros
- Not Object-oriented

Atomic Data Types

- Arbitrary precision integers - `12345678987654`
- Doubles `1.234` , BigDecimals `1.234M`
- Ratios - `22/7`
- Strings - `“fred”` , Characters - `\a \b \c`
- Symbols - `fred ethel` , Keywords - `:fred :ethel`
- Booleans - `true false` , Null - `nil`
- Regex patterns `#“a*b”`

Data Structures

- Lists - singly linked, grow at front
  - `(1 2 3 4 5), (fred ethel lucy), (list 1 2 3)`
- Vectors - indexed access, grow at end
  - `[1 2 3 4 5], [fred ethel lucy]`
- Maps - key/value associations
  - `{:a 1, :b 2, :c 3}, {1 “ethel” 2 “fred”}`
- Sets `#{fred ethel lucy}`
- Everything Nests

Syntax

- You’ve just seen it
- Data structures *are* the code
- Not text-based syntax
  - Syntax is in the interpretation of data structures
- Things that would be declarations, control structures, function calls, operators, are all just lists with op at front
- Everything is an expression

```python
# Norvig’s Spelling Corrector in Python
# http://norvig.com/spell-correct.html

def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))
alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
    n = len(word)
    return set([word[0:i]+word[i+1:] for i in range(n)] +
               [word[0:i]+word[i+1]+word[i]+word[i+2:] for i in range(n-1)] +
               [word[0:i]+c+word[i+1:] for i in range(n) for c in alphabet] +
               [word[0:i]+c+word[i:] for i in range(n+1) for c in alphabet])

def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)

def known(words): return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
    return max(candidates, key=lambda w: NWORDS[w])
```

```clojure
; Norvig’s Spelling Corrector in Clojure
; http://en.wikibooks.org/wiki/Clojure_Programming#Examples

(defn words [text] (re-seq #"[a-z]+" (.toLowerCase text)))

(defn train [features]
  (reduce (fn [model f] (assoc model f (inc (get model f 1))))
          {} features))

(def *nwords* (train (words (slurp "big.txt"))))

(defn edits1 [word]
  (let [alphabet "abcdefghijklmnopqrstuvwxyz", n (count word)]
    (distinct (concat
      (for [i (range n)] (str (subs word 0 i) (subs word (inc i))))
      (for [i (range (dec n))]
        (str (subs word 0 i) (nth word (inc i)) (nth word i) (subs word (+ 2 i))))
      (for [i (range n) c alphabet] (str (subs word 0 i) c (subs word (inc i))))
      (for [i (range (inc n)) c alphabet] (str (subs word 0 i) c (subs word i)))))))

(defn known [words nwords] (for [w words :when (nwords w)]  w))

(defn known-edits2 [word nwords]
  (for [e1 (edits1 word) e2 (edits1 e1) :when (nwords e2)]  e2))

(defn correct [word nwords]
  (let [candidates (or (known [word] nwords) (known (edits1 word) nwords)
                       (known-edits2 word nwords) [word])]
    (apply max-key #(get nwords % 1) candidates)))
```

Java Interop

```
Math/PI
3.141592653589793

(.. System getProperties (get "java.version"))
"1.5.0_13"

(new java.util.Date)
Thu Jun 05 12:37:32 EDT 2008

(doto (JFrame.) (add (JLabel. "Hello World")) pack show)

;expands into:
(let [x (JFrame.)]
   (do (. x (add (JLabel. "Hello World")))
       (. x pack)
       (. x show))
   x)
```

Clojure is Functional

- All data structures immutable
- Core library functions have no side effects
  - Easier to reason about, test
  - Essential for concurrency
    - Functional by convention insufficient
- let-bound locals are immutable
- loop/recur functional looping construct
- Higher-order functions

Sequences

```
(drop 2 [1 2 3 4 5]) -> (3 4 5)

(take 9 (cycle [1 2 3 4]))
-> (1 2 3 4 1 2 3 4 1)

(interleave [:a :b :c :d :e] [1 2 3 4 5])
-> (:a 1 :b 2 :c 3 :d 4 :e 5)

(partition 3 [1 2 3 4 5 6 7 8 9])
-> ((1 2 3) (4 5 6) (7 8 9))

(map vector [:a :b :c :d :e] [1 2 3 4 5])
-> ([:a 1] [:b 2] [:c 3] [:d 4] [:e 5])

(apply str (interpose \, "asdf"))
-> "a,s,d,f"

(reduce + (range 100)) -> 4950
```

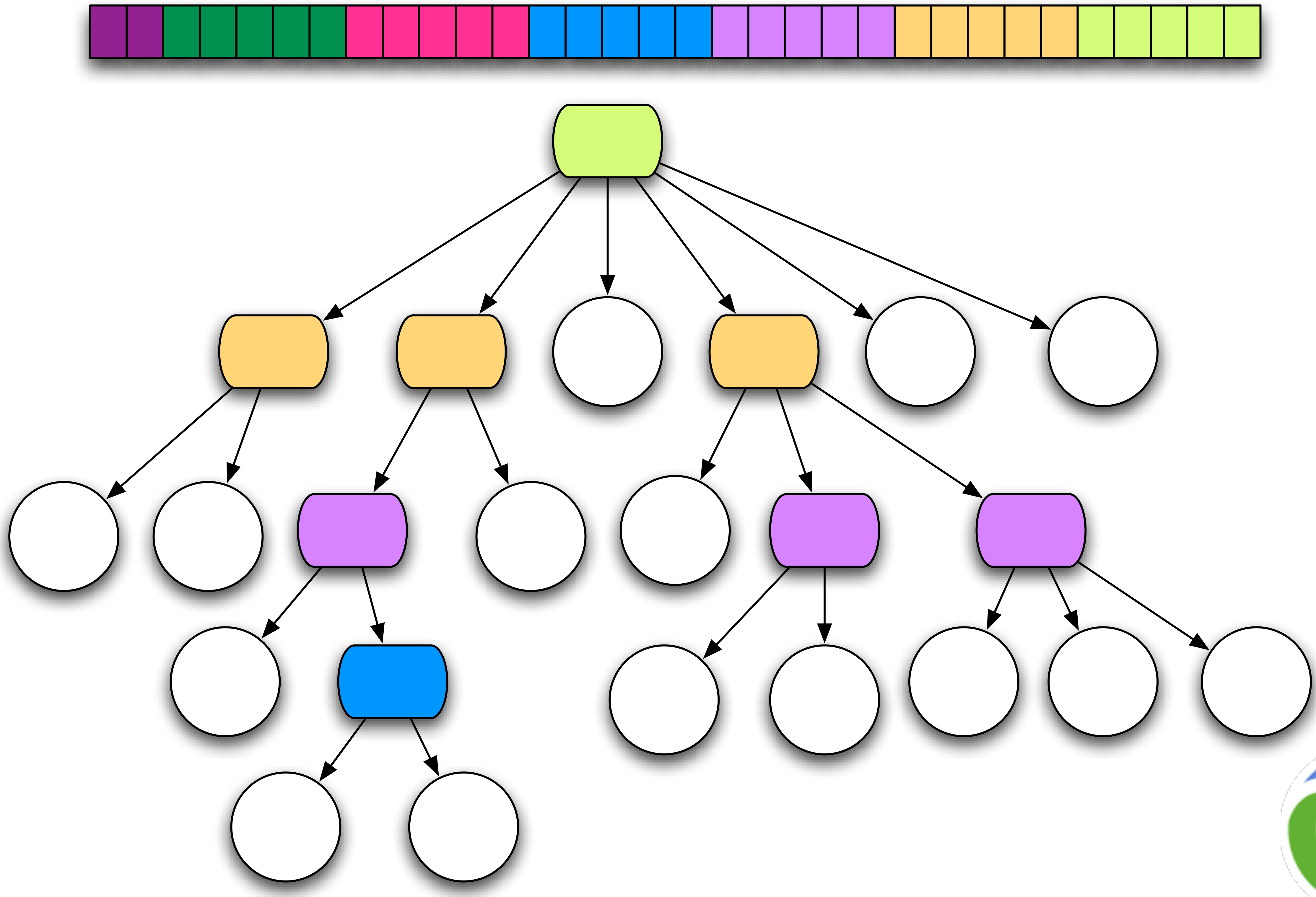
Maps and Sets

```
(def m {:a 1 :b 2 :c 3})

(m :b) -> 2 ;also (:b m)

(keys m) -> (:a :b :c)

(assoc m :d 4 :c 42) -> {:d 4, :a 1, :b 2, :c 42}

(merge-with + m {:a 2 :b 3}) -> {:a 3, :b 5, :c 3}

(union #{:a :b :c} #{:c :d :e}) -> #{:d :a :b :c :e}

(join #{{:a 1 :b 2 :c 3} {:a 1 :b 21 :c 42}}
      #{{:a 1 :b 2 :e 5} {:a 1 :b 21 :d 4}})

-> #{{:d 4, :a 1, :b 21, :c 42}
     {:a 1, :b 2, :c 3, :e 5}}
```

Persistent Data Structures

- Immutable, + old version of the collection is still available after 'changes'
- Collection maintains its performance guarantees
  - Therefore new versions are not full copies
- Structural sharing - thread safe, iteration safe
- All Clojure data structures are persistent
  - Hash map/set and vector based upon array mapped hash tries (Bagwell)
  - Practical - much faster than O(logN)

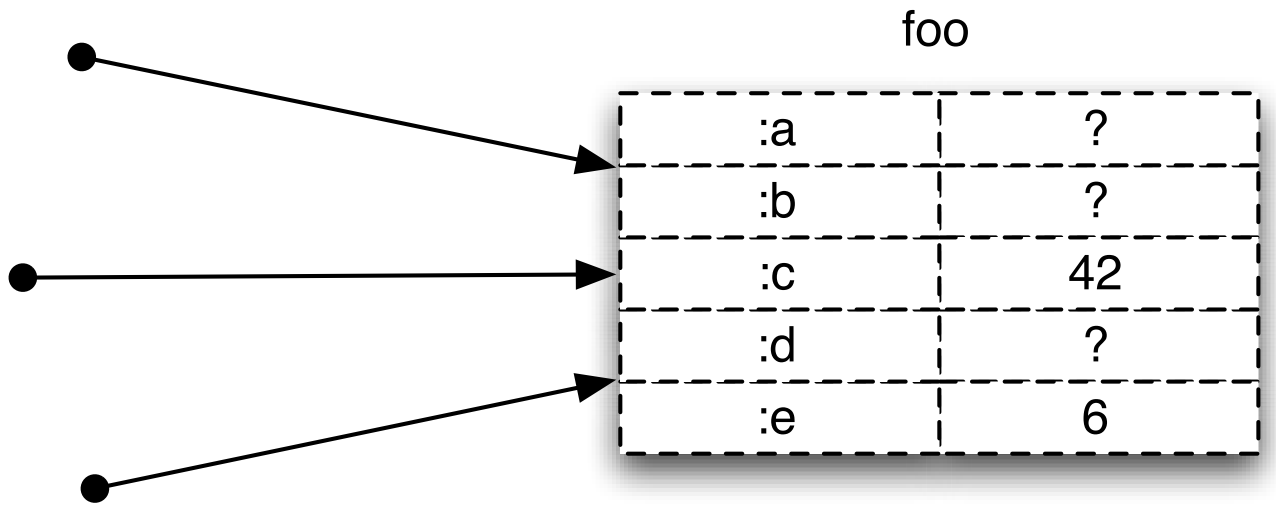
Bit-partitioned hash tries



Concurrency

- 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 `ref`)
  - Concurrency semantics for references
    - Automatic/enforced
    - <u>No locks in user code!</u>

Typical OO - Direct references to Mutable Objects

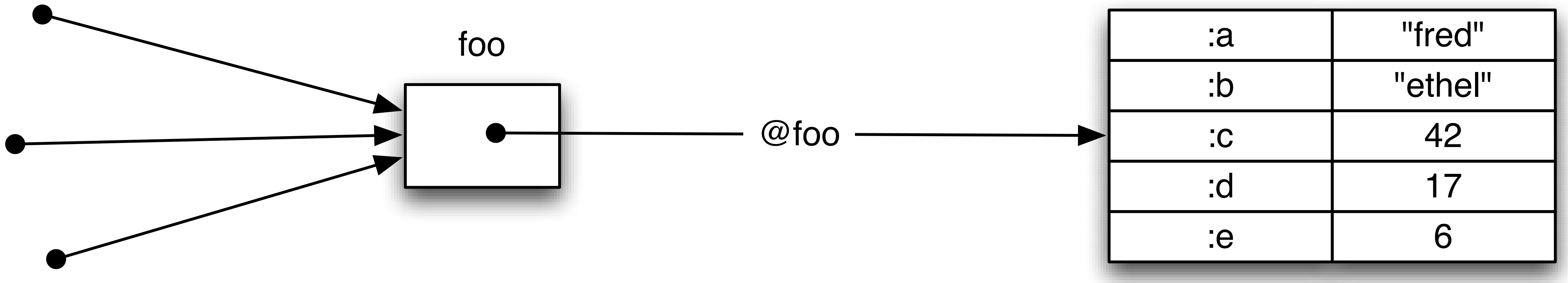


foo

:a	?
:b	?
:c	42
:d	?
:e	6

- Unifies identity and value
- Anything can change at any time
- Consistency is a user problem

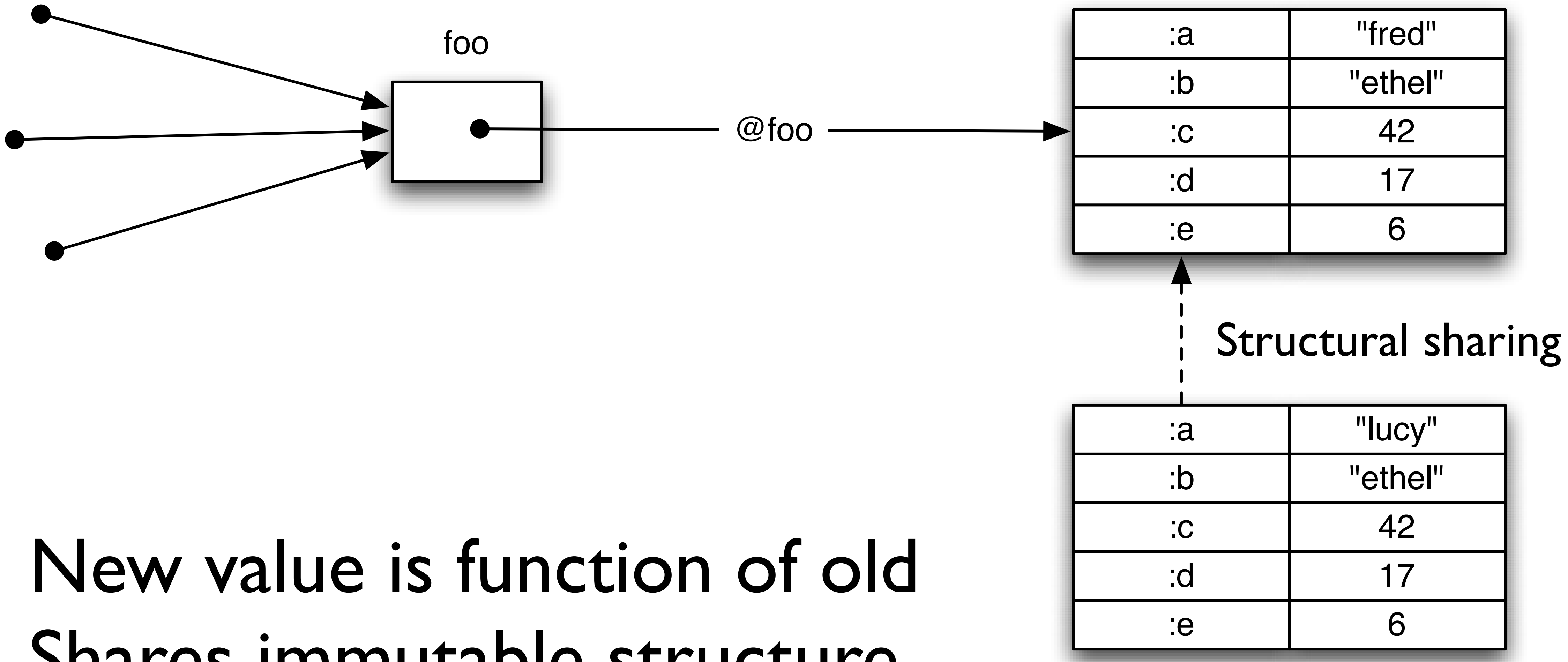
Clojure - Indirect references to Immutable Objects



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

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

Persistent ‘Edit’

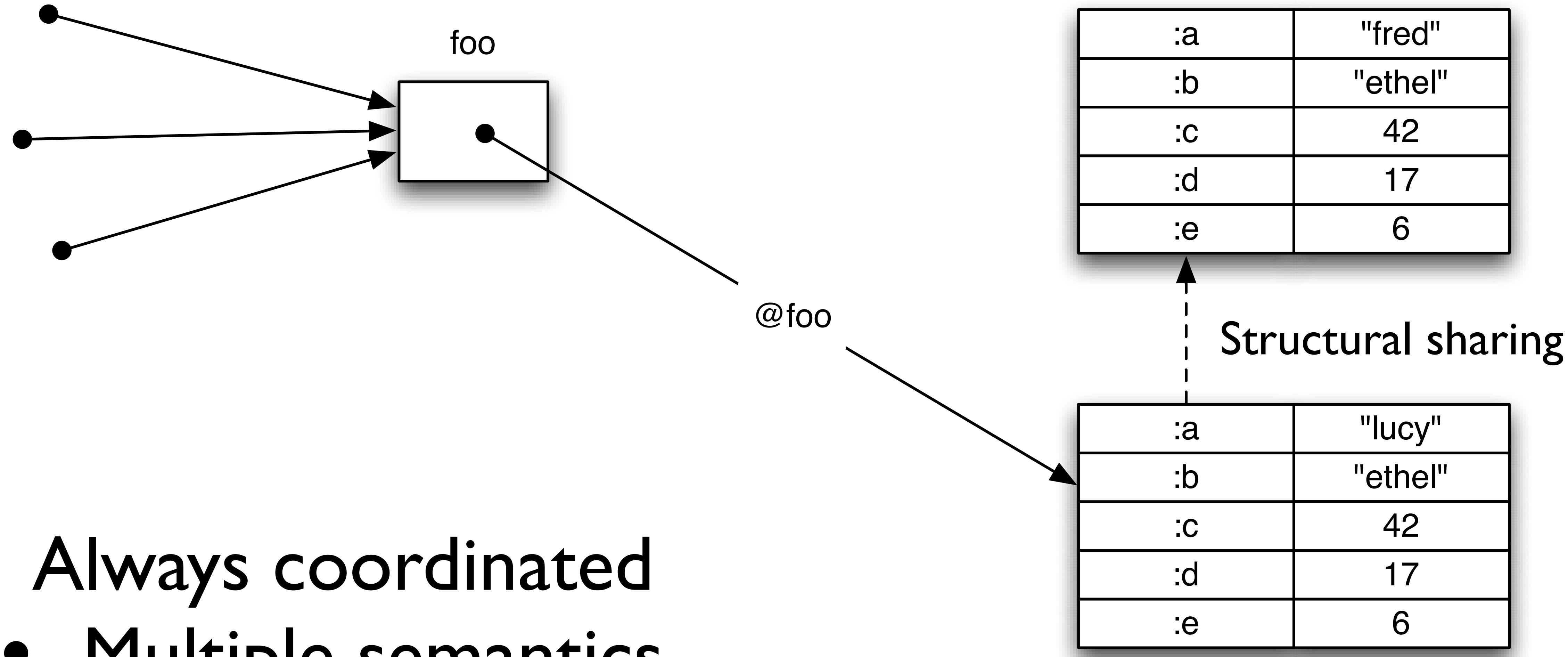


:a	"fred"
:b	"ethel"
:c	42
:d	17
:e	6

: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 Update



: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

Clojure References

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

Refs and Transactions

- Software transactional memory system (STM)
- Refs can only be changed within a transaction
- All changes are Atomic, Consistent 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
  - User 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 impede writers/readers, writers never impede readers, supports commute

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}
```

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)

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}

(await foo)

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

Java Integration

- Clojure strings are Java Strings, numbers are Numbers, collections implement Collection, fns implement Callable and Runnable etc.
- Core abstractions, like seq, are Java interfaces
- Clojure seq library works on Java Iterables, Strings and arrays.
- Implement and extend Java interfaces and classes
- Primitive arithmetic support equals Java's speed.

Implementation - Functions

- Dynamically compiles to bytecode in memory
  - Uses ASM
  - No AOT compilation at present
- Every function is new Class
  - Implements IFn interface
  - Set of invoke methods, overloaded on arity
  - All signatures take/return Objects
  - Variadics based on sequences

Implementation - Calls

- Function calls are straight Java method calls
  - No alternate type system, thunks etc
  - No special extra args
- Calls to Java are either direct or via reflection
  - No wrappers, caches or dynamic thunks
  - Type hints + inference allow direct calls
  - Very few hints needed to avoid reflection
    - compiler flag can generate warnings

Implementation - Primitives

- Locals can be primitives, arrays of primitives
- Math calls inlined to primitive-arg static methods
- HotSpot finishes inlining to primitive math
- Result is same speed as Java

```
(defn foo [n]
  (loop [i 1]
    (if (< i n)
      (recur (inc i))
      i)))

(time (foo 100000))
"Elapsed time: 1.428 msecs"
100000
```

```
(defn foo2 [n]
  (let [n (int n)]
    (loop [i (int 0)]
      (if (< i n)
        (recur (inc i))
        i)))))

(time (foo2 100000))
"Elapsed time: 0.032 msecs"
100000
```

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
- java.util.concurrent is still right tool for caches/queues

Pain Points

- No tail call optimization
  - Important for some functional idioms
  - Major point of criticism for choice of JVM from functional circles
- Use Java's boxed Numbers + own Ratio
  - Integer, Long, BigInteger etc
  - Slow generic math, numbers on heap
  - Would love tagged fixnums and/or standard high performance boxed math lib

Conclusion

- Very happy with the JVM
  - Good performance, facilities, tools, libraries
- Clojure fills a niche
  - Dynamic + functional + JVM
- Lots of interest in first 11 months:
  - 500+ user mailing list, 500+ messages/month
  - 10,000+ SVN reads/month
  - Active community

Thanks for listening!

http://clojure.org