What parens? All I see is the program! (an introduction to Clojure)

Trifork A/S, Headquarters
Margrethepladsen 4,
DK-8000 Århus C,
Denmark
info@trifork.com
http://www.trifork.com

JAOO 2010 Geek night
August 31st, 2010,
Trifork, Aarhus

Pop-quiz: consider this class

```
public class HashMapDemo {
      private final static Map<Integer, Integer> m = new HashMap(1);
      public static void write(final int offset) {
            for (int i = 0; i < 10000; i++) {
                  int k = offset+i;
                  m.put(k, k+offset);
            }
      }
      public static void read(final int offset) {
        for (int i = 0; i < 10000; i++) {
            int key = offset+i;
            Integer val = m.get(key);
            if (val != null) {
                  if (val.intValue() != -key) {
                        System.out.println("Key and value don't match...");
                  }
            }
        }
      }
```

- Suppose we have multiple threads calling both read and write without synchronization.
- Now, what can happen?

Non-obvious bug 1

- ArrayIndexOutOfBoundsException?

```
krukow:~/workspaces/trifork/concurrency$ java -cp bin hashmap.HashMapDemo
Exception in thread "Thread-0"
                    java.lang.ArrayIndexOutOfBoundsException: 23
      at java.util.HashMap.get(HashMap.java:301)
      at hashmap.HashMapDemo.read(HashMapDemo.java:17)
      at hashmap.HashMapDemo$1.run(HashMapDemo.java:32)
      at java.lang.Thread.run(Thread.java:637)
WRITE done: j = 1
READ done: j = 2
READ done: j = 4
READ done: j = 6
WRITE done: j = 3
READ done: j = 8
READ done: j = 10
```

Non-obvious bug 2

- Inifinite loop!!

```
krukow:~/workspaces/trifork/concurrency$ java -cp bin hashmap.HashMapDemo
READ done: j = 0
READ done: j = 2
WRITE done: j = 1
WRITE done: j = 3
…
READ done: j = 12
WRITE done: j = 11
READ done: j = 14
WRITE done: j = 15
WRITE done: j = 17
READ done: j = 18
READ done: j = 16
WRITE done: j = 19
^[[A
^[[B
^[[A
^[[A
^C
```

Is this artificial?

- NO! This does happen in “real world” programs
  - *Incorrect optimizations*:
    - “I really can't pay the cost of synchronization (even though I haven't measured it), and in ***this*** particular case a data-race is safe.”
  - *Non-obvious sharing*:
    - “I thought this object wasn't shared between multiple threads.”
  - *Design changes*.
    - In the original design this object wasn't shared.
    - But now it is, for some reason: design change, (bad) optimizations, singleton/caching.
  - *Bad library/framework*.
    - The bug may not even be in your code!
- Still dont believe me? I've seen the “endless loop” happen on production servers *this month*

Apache MyFaces 1.1.8 (JSF impl).

- JSF is a web framework for Java which offers a UI component model for the web development.
  - Componets are placed on pages using e.g. the JSP language with tags for each component.
  - *UIComponentTag*: abstract super class of all JSP tags that represent components.
- The findComponent(..) method

```
/**
   * Return the corresponding UIComponent for this tag, creating it
   * if necessary.
   * <p>
   * If this is not the first time this method has been called, then
   * return the cached component instance found last time.
   * <p>...
```

Code: UIComponentClassicTagBase

```
private UIComponent _componentInstance = null;
protected UIComponent findComponent(FacesContext context)
            throws JspException
    {
        if (_componentInstance != null) return _componentInstance;
        UIComponentTag parentTag = getParentUIComponentTag(pageContext);
        if (parentTag == null)
        {
            //This is the root
            _componentInstance = context.getViewRoot();
            setProperties(_componentInstance);
            return _componentInstance;
        }
      //...
```

- As it turns out, this method can be called by several threads.
- setProperties(..) sets a number of key value pairs in ....
  - A HashMap :)

MyFaces JSF Portlet Bridge (trunk)

- Double checked locking (+ another bug - spot it!)
  - Broken lazy initialization technique
  - Unsound optimization

```
private void initBridge() throws PortletException {
    if (mFacesBridge == null) {
      try {
        // ensure we only ever create/init
        //  one bridge per portlet
        synchronized(mLock) {
          if (mFacesBridge == null)
          {
            mFacesBridge = mFacesBridgeClass.newInstance();
            mFacesBridge.init(getPortletConfig());
          }
        }
      }
      catch (Exception e) {
        throw new PortletException("...", e);
      }
    }
  }
```

More real-life bugs...

- IceFaces: unsound "optimization"
  - *Store a mutable object (SwfLifecycleExecutor) in a map in application scope*
    - *Each requests "initializes" it setting variables*
  - *Design changes: works in 1.8.0. broken 1.8.2*
- Spring WebFlow: ***unintended sharing***
  - Storing non-thread safe object in application scope
  - https://jira.springframework.org/browse/SWF-976

- Goetz: are all stateful webapps broken? http://www.ibm.com/developerworks/library/j-jtp09238.html

This is production code which has gone through QA!

I'm not pointing fingers, but trying to convince you that it is easy to make mistakes and everyone does it.. The list goes on and on...

What can we learn?

- Immutable data saves the day
- Not sharing data (thread confinement or serialization) for mutable data
- Synchronize when accessing shared mutable data
  - BOTH reader and writer must synchronize (visibility)
  - Don't try to be clever and optimize
  - Beware of non-obvious sharing
    - Is this mutable piece of data shared?
  - Can be hard in case of multiple locks...
    - Easy to forget, hard to debug, fail under load
- Use data structures from java.util.concurrent
  - atomics or volatile on JDK1.5+
- Beware with lazy initialization, singletons, caches

What if...

- ... there was a practical language
  - where all objects were immutable by default, even common data structures like hash table and vector (→ immutability saves the day)
  - where all shared mutable state was explicitly marked in the program (→ obvious mutable sharing)
  - where all interaction with mutable state was managed by the language runtime (→ visibility)
  - where synchronization was possible without the complexity of locks (→ atomic multi-operations easy)
- Then many, although not all, concurrency problems become much easier

Clojure is such a language

:)

What else?

- The most important part of Clojure is its state management features
  - vars, refs, atoms, agents
  - Functional (pure functions as basic building blocks)
- But there are other important parts
  - Dynamic (flexible, expressive)
  - Meta-programming and DSL facilities (a Lisp, macros)
  - Designed to be hosted; with good interop (JVM, CLR)
  - Programming to contracts (protocols)
    - Sequences and library
  - Fast by default (and with “knobs” for hot-spots)
  - and “goodies” (laziness, metadata, destructuring, multi-methods,...)

The Shameless Plug...

Obviously WAY too much to cover so

Come to JAOO Aarhus 2010 to hear much more

http://jaoo.dk/aarhus-2010/speaker/Rich+Hickey

http://jaoo.dk/aarhus-2010/speaker/Stuart+Halloway

Clojure Tutorial with Stuart!

20% Discount if member of DCUG (www.clojure.dk)
- free choice of tutorial/conference days.
Goto www.clojure.dk and sign up, then email me :) kkr@trifork.com

Also JAOO and the Danish Clojure Users Group is organizing a free event where you can meet Rich Hickey, Stuart Halloway as well as many Clojure users in Denmark

https://secure.trifork.com/aarhus-2010/freeevent/index.jsp?eventOID=2698

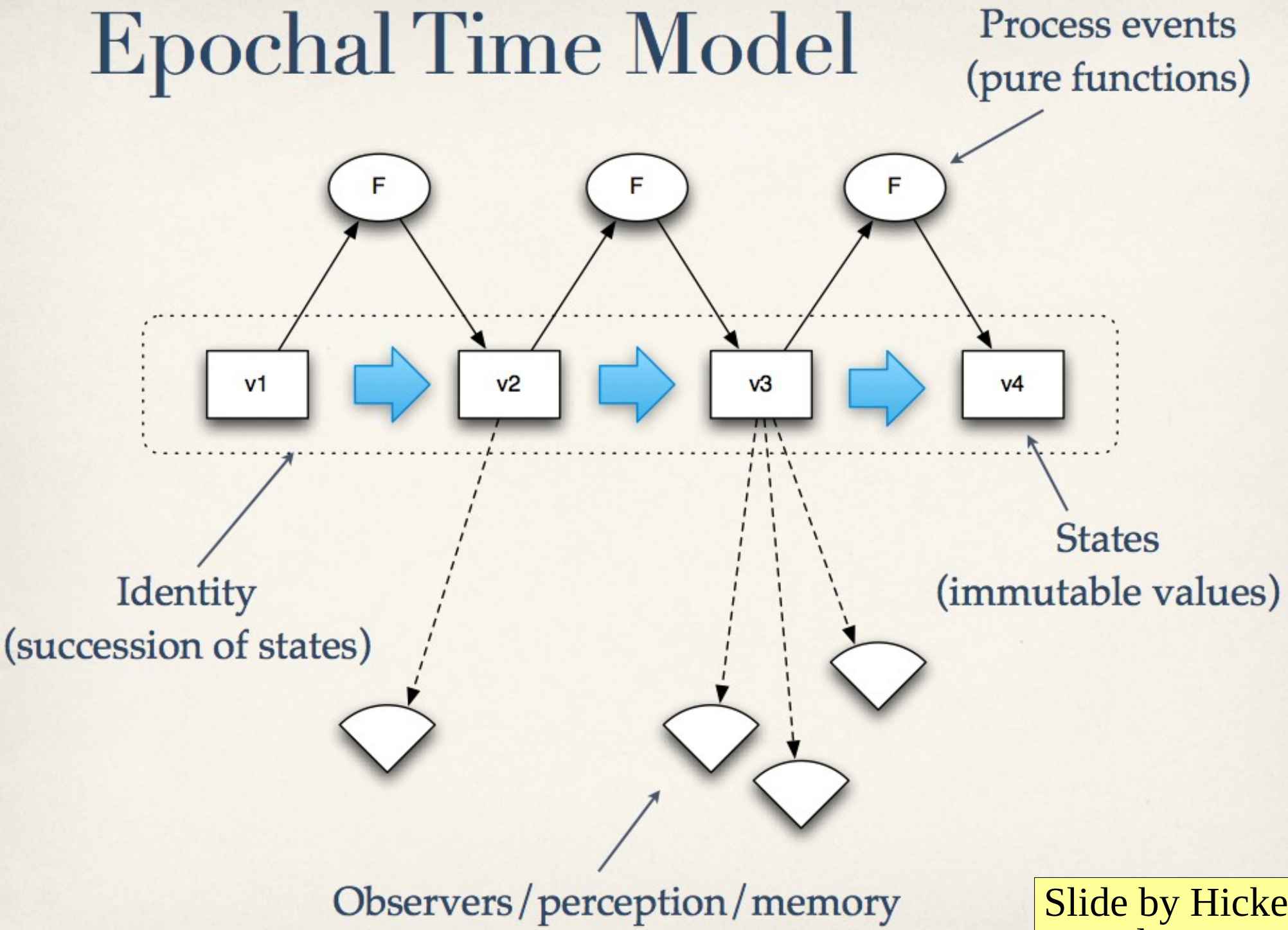
Agenda

- Immutability and functional programming in Clojure
- Managed change and shared data
- Demos if we have time
  - Host interoperability
  - Protocols (circuitbreaker)
  - macros

Value, state, identity: Clojure way

- What is a value?
  - An immutable object
- What is an identity?
  - A *logical* entity that takes on a unique value at any given time
  - Takes on a series of values over time
  - Usually has one or many names
- What is State?
  - The value of an identity at a given time.
  - Snapshot: a value: stable, immutable.
- See refs for more info...

Epochal Time Model



Slide by Hickey
JVM lang summit

Values

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"`

Composite Data Types

- 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

Persistent Data Structures

- In Clojure, generally, all data structures and objects are ***immutable***
  - Sharing among threads - no synch needed
  - Can be saved for later inspection
  - Provides composite ***values***
- There are efficient operations operations for creating ***variants*** of a data structure, for example:
  - If **`M`** is a clojure map, then **`assoc(M,k,v)`** is a clojure map which is like **`M`** except that it maps key **`k`** to value **`v`**
  - (within 1-4x their mutable counterparts, or faster :)
- Further: they are ***persistent*** meaning that the operations are non-destructive, e.g.,
  - Both **`M`** and **`assoc(M,k,v)`** are usable and preserve their performance guarantees.

NO!

... that doesn't entail copying the entire structure(!)

See my blog for explanation:

http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/
http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice/
http://blog.higher-order.net/2010/08/16/assoc-and-clojures-persistenthashmap-part-ii/

Also

http://blog.higher-order.net/2010/06/11/clj-ds-clojures-persistent-data-structures-for-java/

Sequence library and pure functions

Sequences: example of programming to abstractions

- Lift the first/rest abstraction off of concrete lists
- Function seq
  - (seq coll) gives **nil** if empty otherwise a seq on coll
  - **first**, calls seq on arg if not already a seq
    - returns first item or **nil**
  - **rest**, calls seq on arg if not already a seq
    - returns the next seq, if any, else **nil**
- Most library functions are fully lazy
- Vast library works on: all Clojure DS, Java: Strings, arrays, collections, iterables
  - All ***pure functions***

Sequence library examples

```
drop 2, [1, 2, 3, 4, 5] → (3, 4, 5)

cycle [1, 2, 3] → [1, 2, 3, 1, 2, 3, …]

partition 3, [1, 2, 3, 4, 5, 6] → ((1 2 3) (4 5 6))

interpose \, "Fred" → (\F \, \r \, \e \, \d)

map inc [1, 2, 3] → (2, 3, 4)

reduce +
   range 100 → 4950
```

Examples: Map functions

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

m :b → 2      (maps are functions)

:b m → 2    (keywords are functions)

keys m → (:a, :b, :c)

assoc m, :d 4 → {:a 1, :b 2, :c 3, :d 4} ;;m is unchanged

dissoc m, :a → {:b 2, :c 3} ;;m is unchanged

map :name [{:name "Fred"}, {:name "Ethel"}]
                    → ("Fred", "Ethel")

merge-with +, m, {:a 2, :b 3} → {:a 3, :b 5, :c 3}
```

Records: another kind of object

- Records aren't like objects in traditional OO languages (think Java), but there are similarities
  - Represent “domain types”
  - Store data in fields; can have associated “methods”
  - Can implement host interfaces
- But there are also differences with traditional objects
  - Records are immutable ('setters' create new records)
  - Automatic impl. of hashCode, equals, toString
  - Extensible and map-like behaviour: implementents interfaces clojure.lang.IPersistentMap and java.util.Map
  - No inheritance
  - Methods & polymorphism only via interfaces and protocols (more on that)

If it were Java a record would be similar to this

```
public class PersonRecord implements IPersistentMap, java.util.Map // and more...
{
        public final String firstName;
        public final Integer age;
        public final AddressRecord address;
        public PersonRecord(String firstName, Integer age, AddressRecord address) {
                this.firstName = firstName;
                this.age = age;
                this.address = address;
        }

        public Object get(Object key) {
                if (isFirstnameKey(key)) {
                        return firstName;
                } //else {}
                //...
                return null;
        }

        public IPersistentMap assoc(Object key, Object val) {
                //create a map which is like this one
                //but has also key -> value
                return null;//
        }
        //... more methods...
}
```

```
defrecord Person,
        [f rstname, age,  address]
```

Map-like nature of Records

```
defrecord Person :
        [f rstname, age,  address]
```

- The map-like nature of records means
  - Records can be extended as a map

```
def p
  Person. "Karl", 42, nil
```

```
assoc p, :lastname, "Krukow"
```

→

```
#:Person{:f rstname "Karl",
         :age 42,
         :address nil,
         :lastname "Krukow"}
```

  - In general, any function that works on maps works on records

```
keys p
```

→

```
(:f rstname, :age, :lastname)
```

```
second,
      vals p
```

→

```
42
```

Protocols: another kind of interface

- Protocols aren't like interfaces in traditional OO languages (think Java), but there are similarities
  - Like interfaces, protocols are a grouping of “method” sigs.
  - The “methods” are polymorphic: depending on the type of the receiver an impl. is selected
- But there are also differences with traditional interfaces
  - There is no type hiearchy
  - Protocols are extensible: any protocol can dynamically be extended to reach any type (but you should own the either the type or the protocol)
  - Protocols “methods” aren't methods they really are ***functions*** (their first argument can be thought of as the 'receiver')

If it were Java a protocol would be similar to this

```
public interface Indexed {
        Object at(int i);
        Object atWithDef(int i, Object notFound
}
public interface TopScored {
        Object getTopScore();
}
```

```
defprotocol Indexed,
  at [this, i] "return el at ith pos",
  atwithdef [this, i, d] "...."

defprotocol TopScored,
  topscore [this] "return topscore"
```

```
public class HighscoreRecord implements Indexed,
{
        public final IPersistentVector scores;
        public HighscoreRecord(IPersistentVector
                this.scores = scores;
        }
        public Object getTopScore() {
                return nth(0);
        }
        public Object at(int i) {
                return this.scores.nth(i);
        }
        public Object atWithDef(int i, Object no
                return this.scores.nth(i, notFou
        }
//and much more
}
```

```
defrecord Highscores [scores],
  Indexed,
    at [this i],
          nth scores, i
    atwithdef [this, i, d],
          nth scores, i, d

  TopScored,
    topscore [this],
          at this, 0
```

```
def h
  Highscores. [42 22 2]

topscore h ;; gives 42
```

Extensibility of procotols

- As mentioned, protocols can be extended to all types.
  - for example it is possible to extend our Indexed protocol to Java native type String

```
defprotocol Indexed,
  at [this, i] "return el at ith pos",
  atwithdef [this, i, d]
```

```
extend-protocol Indexed,
   String,
     at [str i],
       . str charAt i
     atwithdef [str i d],
       ;define implementation on string here...
```

```
at "Karl" 2

;result is → \r
```

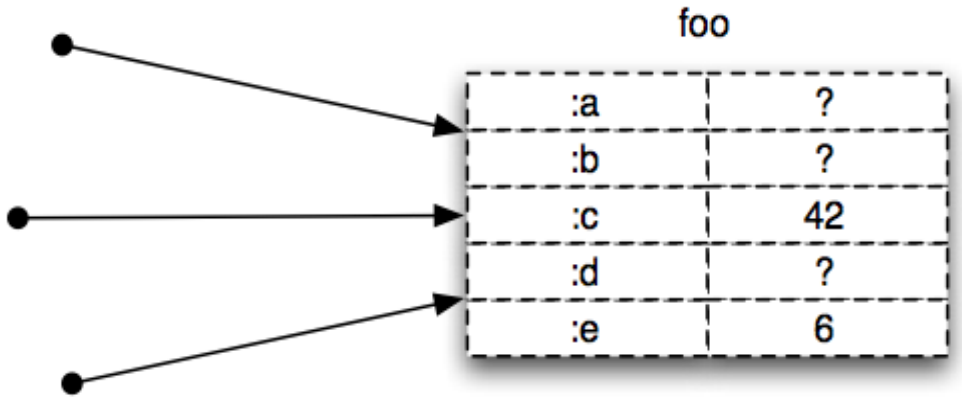
- Can be done at any point in the program

Managing change

Clojure philosophy

- Clojure supports programming with pure functions and persistent data structures
  - this even extends to user defined types & interops with well with host
- Philosophy: Most parts of most programs could and should be functional
  - Yet we recognize that most real-world programs are not functions; they are processes.
- The future is concurrent; locks and mutable objects make complex, intractable programs
- In Clojure everything is pure functional, except
  - ”Reference types” which have a concurrency-aware semantics for change. (vars, atoms, agents, refs)

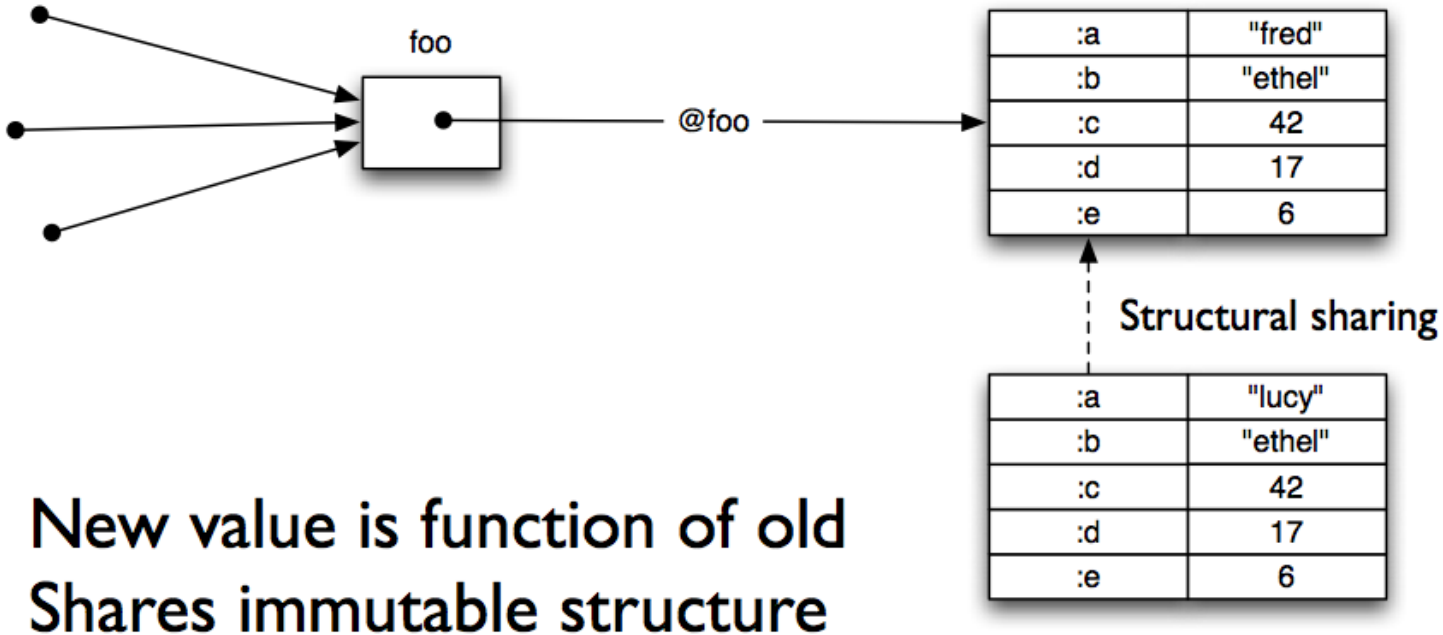
Traditional OO approach: Direct References to Mutable Objects



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

Slide by Hickey

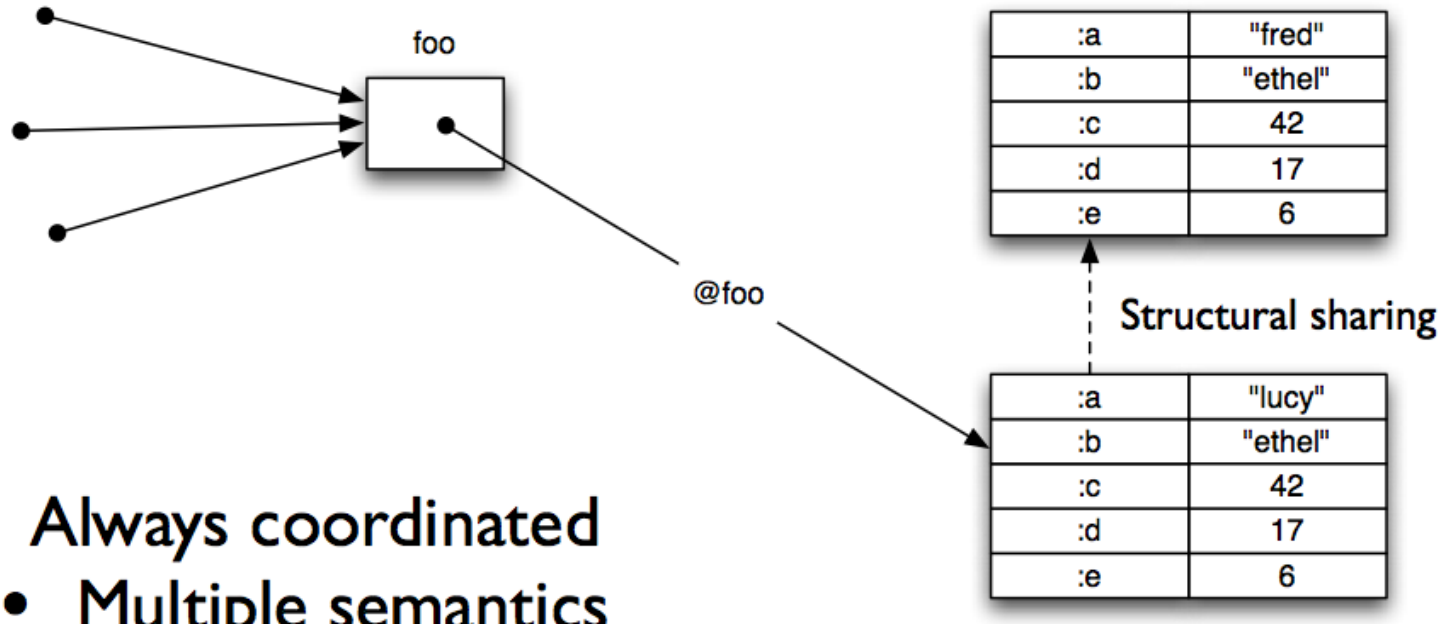
Clojure approach: indirect references to values



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

Slide by Hickey

Atomic update



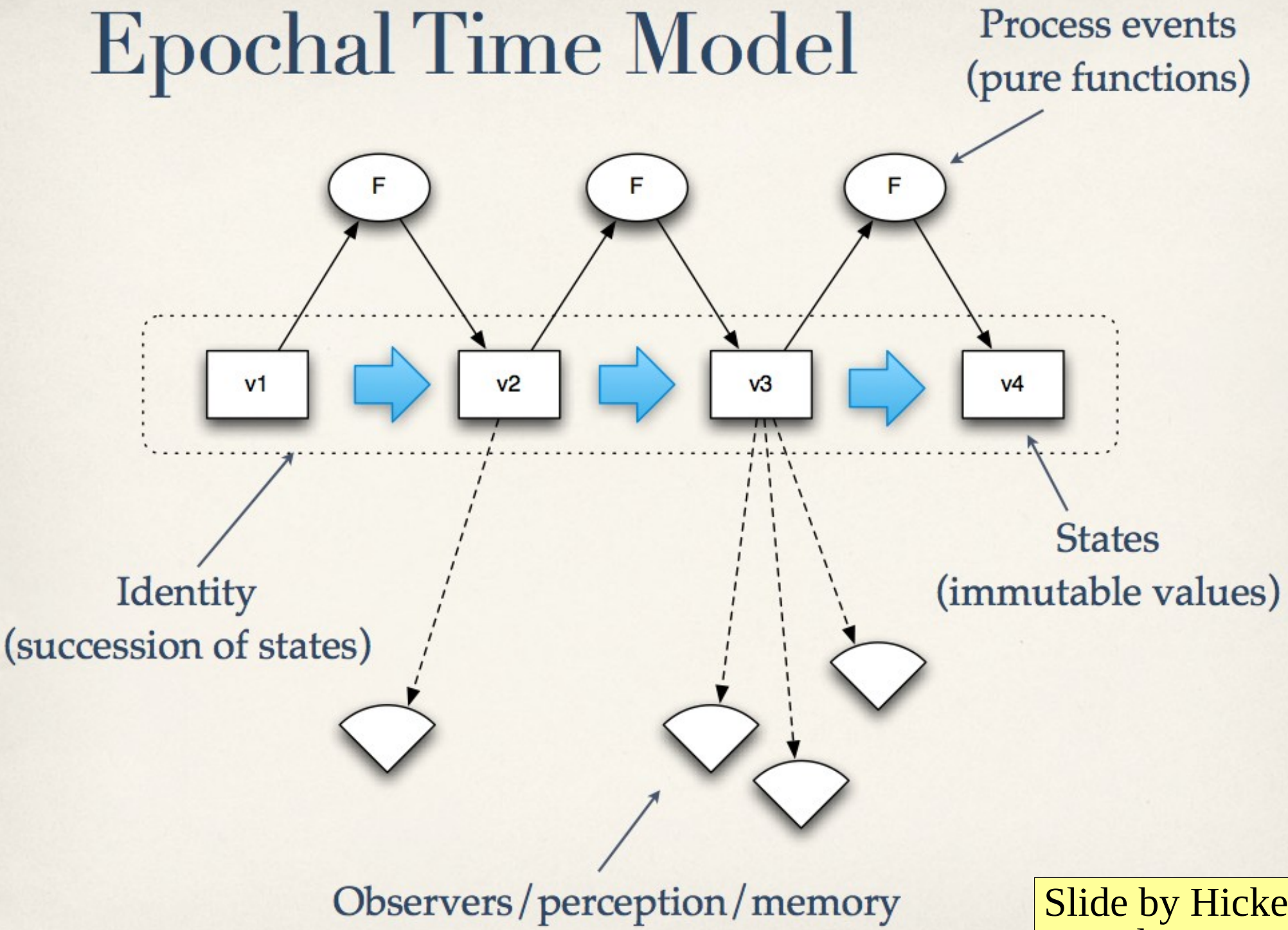
- Always coordinated
  - Multiple semantics
- Next dereference sees new value
- Consumers of values unaffected

Slide by Hickey

Clojure References

- The last slides show an abstract model of change
  - Only references mutate: in a controlled way
  - Allows for multiple semantics
- Currently 4 types of references, all with concurrency semantics:
  - Vars: shared root binding, isolate changes in thread
  - Refs: synchronous, coordinated
  - Atoms: synchronous, independent
  - Agents: asynchronous, independent
- **deref** or reader-macro @ to get value
- Different mutator functions for each type
  - Same syntactic form: **function** reference-type opt-args

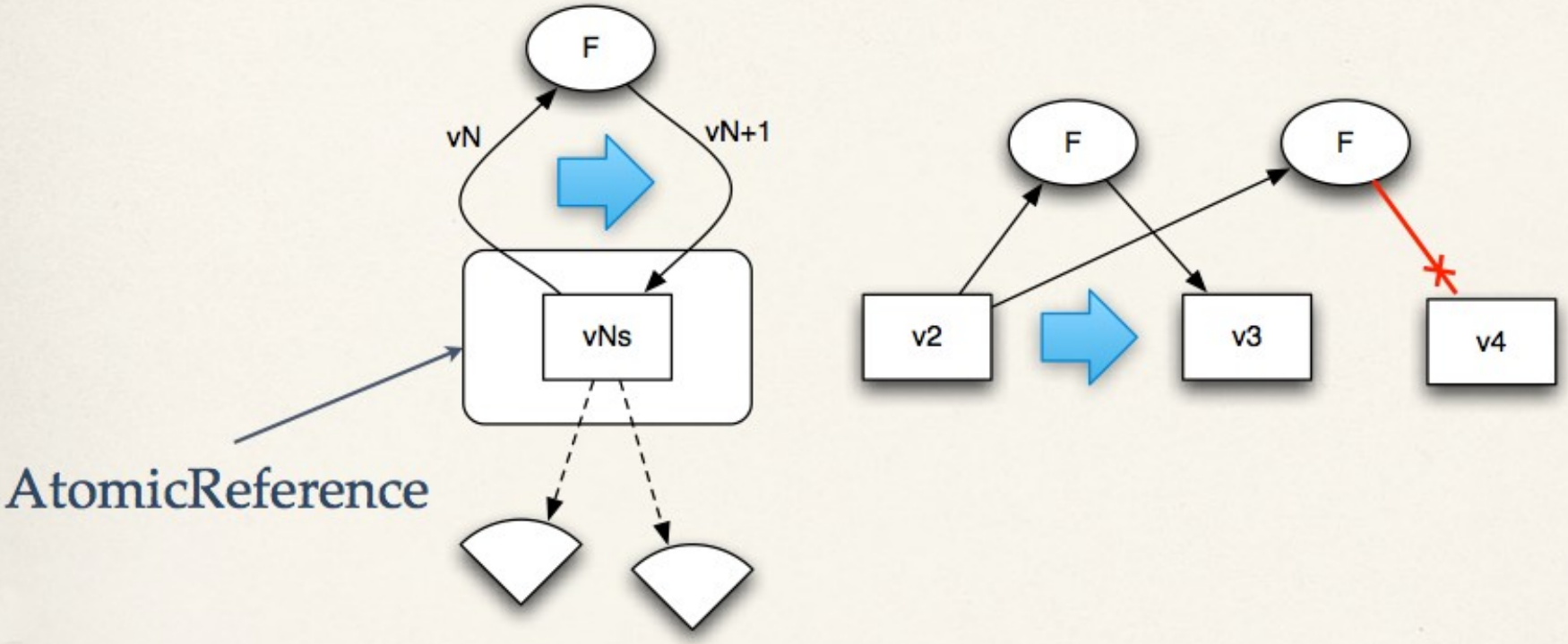
Epochal Time Model



Slide by Hickey
JVM lang summit

CAS as Time Construct

* Slide by Rich Hickey, Jvm Lang. Summit 2009 Keynote



- swap(aRef, f, args)
  - f(vN, args) *becomes* vN+1
  - can automate spin
- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception

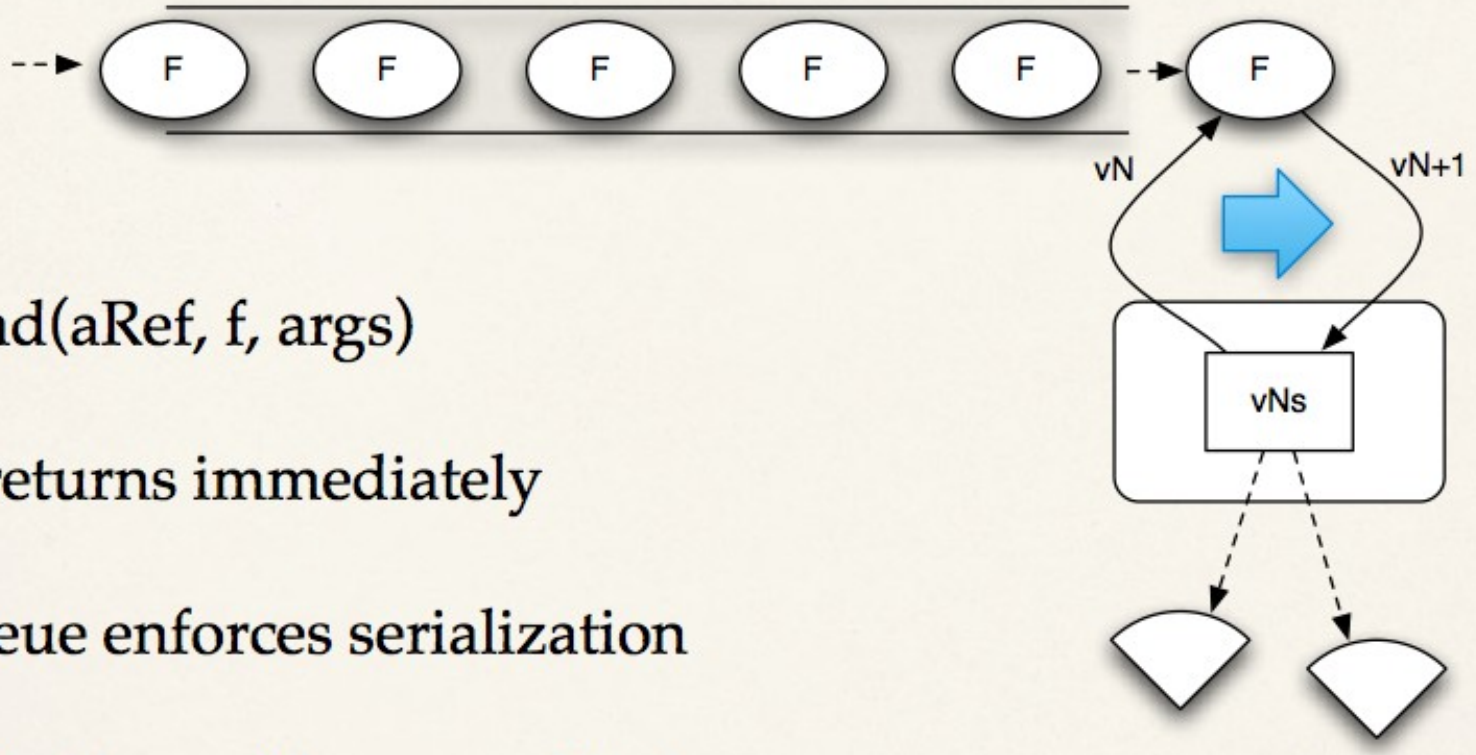
Examples in code

- Atom

```
user> def a
        atom 1
#'user/a
user> @a
1
user> swap! a, inc
2
user> @a
2
```

* Slide by Rich Hickey, Jvm Lang. Summit 2009 Keynote

Agents as Time Construct



- send(aRef, f, args)
  - returns immediately
- queue enforces serialization
  - f(vN, args) *becomes* vN+1
  - happens asynchronously in thread pool thread

- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception

Examples in code

- Agent

```
user> def ag
        agent 1
#'user/ag
user> @ag
1
user> send ag dec
#<Agent@54c21095: 1>
user> @ag
0
```

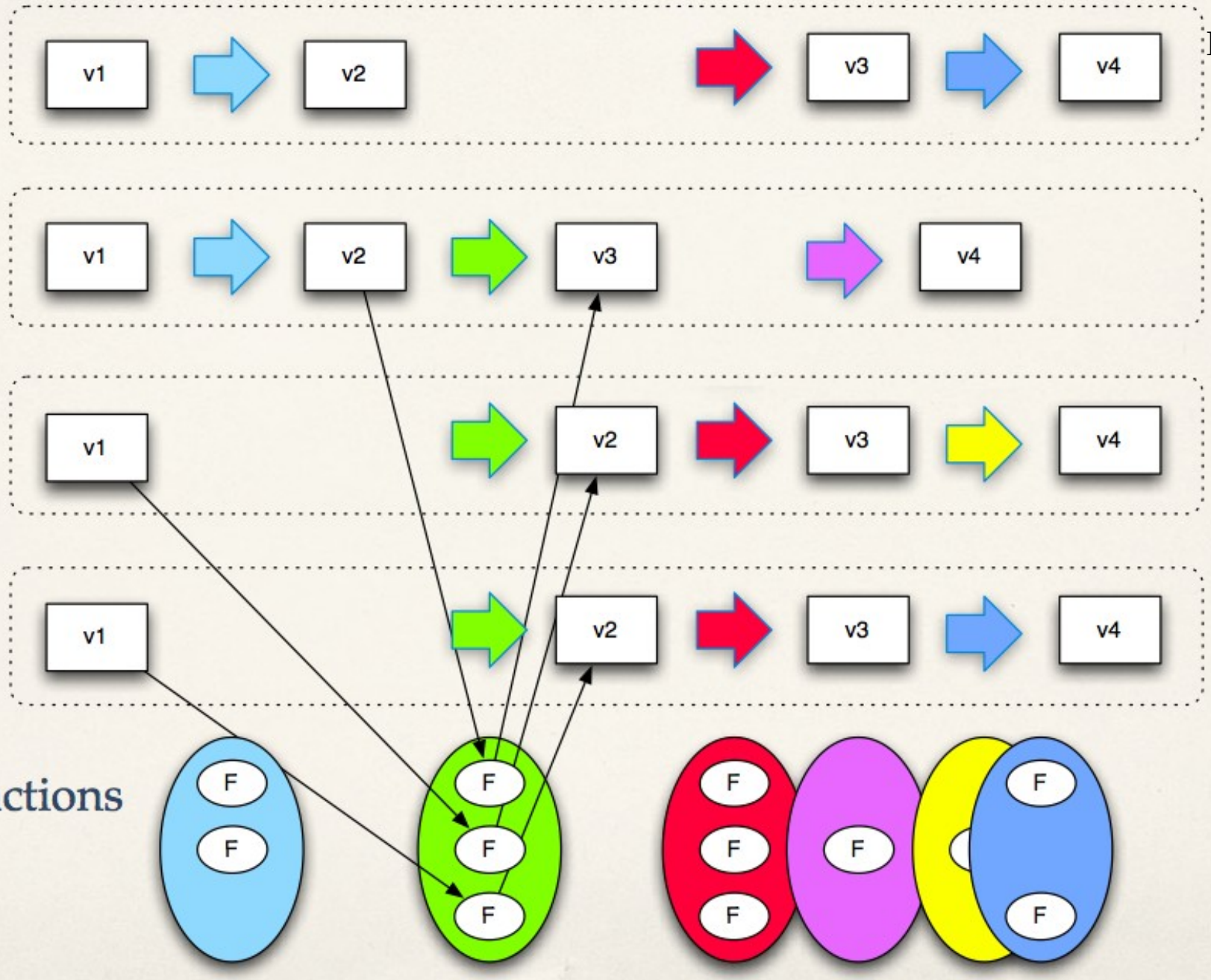
STM

* Slide by Rich Hickey, Jvm Lang. Summit 2009 Keynote

- Coordinates action in (arbitrary) regions involving multiple identities/places
- Multiple timelines intersect in a transaction
- ACI properties of ACID
- Individual components still follow functional process model
  - f(vN, args) *becomes* vN+1

STM as Time Construct

* Slide by Rich Hickey, Jvm Lang. Summit 2009 Keynote

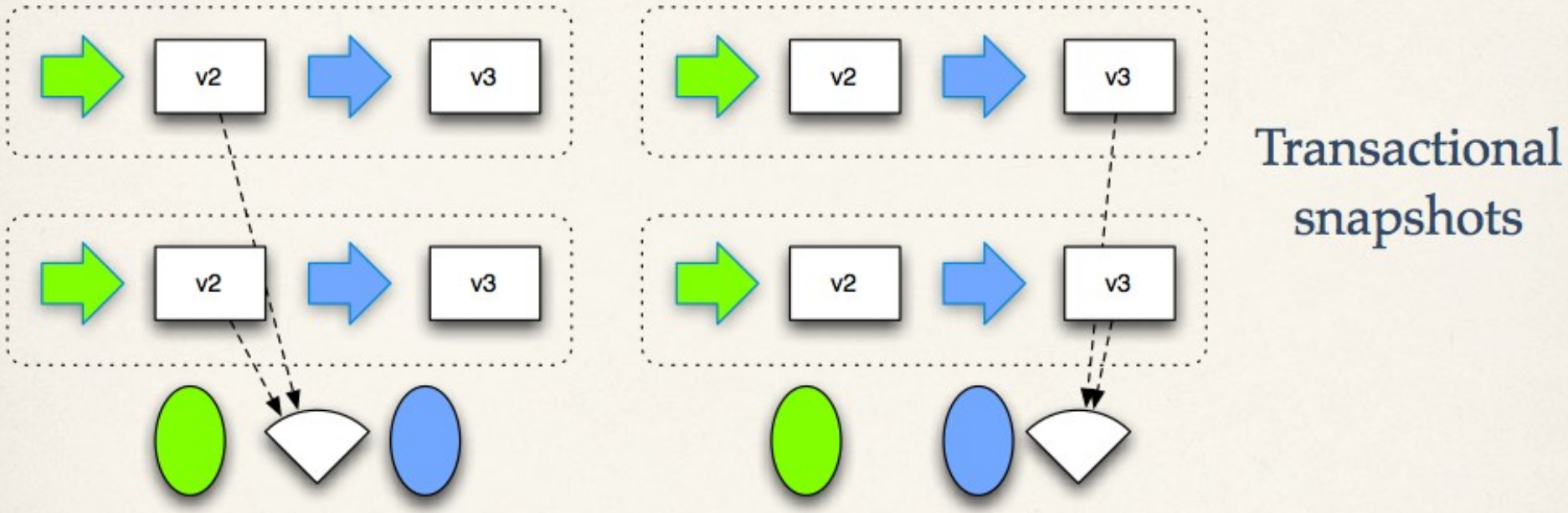


Examples in code

- Ref

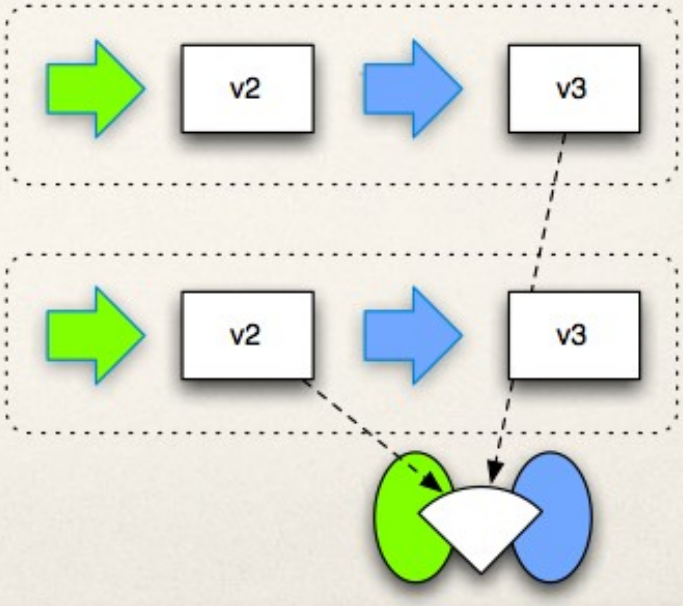
```
user> def k
        ref 1
user> def mk
        ref -1
user> [@k,@mk]
[1 -1]
user> dosync
       let [x, alter k inc]
       ref-set mk, - x
user> [@k,@mk]
[2 -2]
```

Perception in (MVCC) STM



Transactional snapshots

Non-transactional scans



* Slide by Rich Hickey, Jvm Lang. Summit 2009 Keynote

Examples in code

- Transactional scan: consistent snapshot

```
user> def k
        ref 1
user> def mk
        ref -1
user> dosync
        [@k,@mk]
[1, -1]
```

- Non-transactional/panning scan

```
;;run in parallel
;;dosync
;;       let [x, alter k inc]
;;     ref-set mk, - x

user> [@k, @mk]
[724452, -724453]
```

Examples in code

- Atom

```
user> def a
        atom 1
#'user/a
user> @a
1
user> swap! a, inc
2
user> @a
2
```

- Agent

```
user> def ag
        agent 1
#'user/ag
user> @ag
1
user> send ag dec
#<Agent@54c21095: 1>
user> @ag
0
```

- Ref

```
user> def k
        ref 1
user> def mk
        ref -1
user> [@k,@mk]
[1 -1]
user> dosync
       let [x, alter k inc]
       ref-set mk,
                  - x
user> [@k,@mk]
[2 -2]
```

- var

```
user> def x 42
user> def y x
user> binding [x, "KARL: "]
       str x, y
"KARL: 42"
user> x
42
```

Some Clojure answers...

- Immutable data saves the day
- Thread confinement/serial
- Synchronize when accessing shared mutable data
  - Can be hard in case of multiple locks...
- Beware of non-obvious sharing
  - Is this mutable piece of data shared?
- Use data structures from java.util.concurrent
- Beware with lazy initialization, singletons, caches

- immutable by default
- vars confine, and agents serial
- System manages synchronization with atom, ref
  - No user-code synch, Non-interferring reads
- All reference types are marked
  - Data is always immutable in Clojure :)
- Use data structures from java.util.concurrent
  - Not needed as often
- Beware with lazy initialization, singletons, caches
  - Immutability helps!

Confession...

I've not presented Clojure syntax completely accurately...

Clojure syntax

- Clojure compiler is defined in terms of data structures, not text.
- There are textual representations of all the data structures and objects (),[],{},#{},...
  - There are some parens – but not too many, actually

```
(def k (ref 1))

(dosync
  (let [x (alter k inc )]
  (ref-set mk  (- x)))
```

```
(defprotocol Indexed
  (at [this i] "return el at ith pos")
  (atwithdef [this i d] "...."))
(defrecord Highscores [scores]
  Indexed
    (at [this i] (nth scores i))
    (atwithdef [this i d] (nth scores i d)))
```

- Clojure is at least as compact as Ruby/Python...

Syntactic weight: Ruby 1.9 vs Clojure

Note semantics is vastly different although syntax is similar

```ruby
m = {:name => "Karl", :age => 42}
m[:name]


class Person
  attr :name,:age
  attr_writer :name,:age
end

p = Person.new
p.name="Karl";p.age=42;

[p,p,p].map &:name
=> ["Karl", "Karl", "Karl"]

([p,p,p].map &:age).reduce(&:+)
=> 126
```

```clojure
(def m {:name "Karl", :age 42})

(m :name)
(:name m)


(defrecord Person[name age])
(def p (Person. "Karl" 42))




(map :name [p p p])
=> ("Karl" "Karl" "Karl")

(reduce + (map :age [p p p]))
=> 126

(->> [p p p] (map :age)(reduce +))
=> 126
```

References

- JAOO 2010 http://jaoo.dk/aarhus-2010 :), DCUG: www.clojure.dk
- http://clojure.org/
- http://clojure.blip.tv/
- Stuart Halloway: Programming Clojure
  - intro
- Chris Houser, Michael Fogus: The Joy of Clojure
  - Deeper
- Mark Volkmann, STM article
  - http://java.ociweb.com/mark/stm/article.html
- Rich Hickey
  - Are we there yet? http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey
    Value, state, identity http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey