Are We There Yet?

A deconstruction of object-oriented time

Rich Hickey

Provocation

- Are we being well served by the popular OO languages?
- Have we reached consensus that this is the best way to build software?
  - Is there any evidence that this is so?
- Is conventional OO a known good?
  - or just so widely adopted we no longer have the ability to see its attendant costs or limitations?

A Deeply Entrenched Model

- Popular languages today are more similar than they are different
  - Single-dispatch, stateful OO
  - Classes, inheritance, fields, methods, GC
- Smalltalk, Java, C#, Python, Ruby, Scala...

Not so Different

- Differences are superficial
  - MI/Mixins/Interfaces
  - Static/Dynamic typing
  - Semicolons/indentation/blocks
  - Closures/Inner-classes
- Preferences have more to do with programmer sensibilities and expressivity than core principles
- Different cars, same road

Has OO "Won" ?

- Are we just going to tweak this model for the next few decades?
  - People seem to like it
  - Success has bred increasing conservatism, and slowed the pace of change
- The purpose of this talk is not to beat up on OO
  - Just admit the possibility that not only are we not there, we may be driving on the wrong road.

What are we missing?

- Are we ready for an increasingly complex, concurrent and heterogeneous world, or will we be facing some fundamental impedance mismatch?
- What pressures should drive the adoption of new (and often old) ideas not yet in the mainstream?

Some Critical Ideas

- Incidental complexity
- Time / Process
- Functions / Value / Identity / State
- Action / Perception

“Seek simplicity, and distrust it.”

Alfred North Whitehead

Incidental complexity

- Not the complexity inherent in the problem
- Comes along as baggage in the way we formulate our solutions, our tools or languages
- Worst when a side effect of making things appear simple

C++

- Foo *bar(...); //what's the problem?
  - Simple constructs for dynamic memory
    - Simple? - same syntax for pointers to heap and non-heap things
    - Complexity - knowing when/if to delete
- No standard automatic memory management
  - Presents inherent challenge to C++ as a library language
  - Implicit complexity we are no longer willing to bear

Java

- Date foo(...); //what's the problem?
  - Simple - only references to dynamic memory, plus GC
  - Simple? - same syntax for references to mutable/immutable things
  - Complexity - knowing when you will see a consistent value
    - Not (just) a concurrency problem. Can we 'remember' this value, is it stable? If aliased and mutated, who will be affected?
- No standard automatic time management

Familiarity Hides Complexity

- For too many programmers, simplicity is measured superficially:
  - Surface syntax
  - Expressivity
- Meanwhile, we are suffering greatly from incidental complexity
  - Can’t understand larger programs
  - Can’t determine scope of effects of changes to our programs
  - Concurrency is the last straw

“Civilization advances by extending the number of important operations which we can perform without thinking about them.”

Alfred North Whitehead

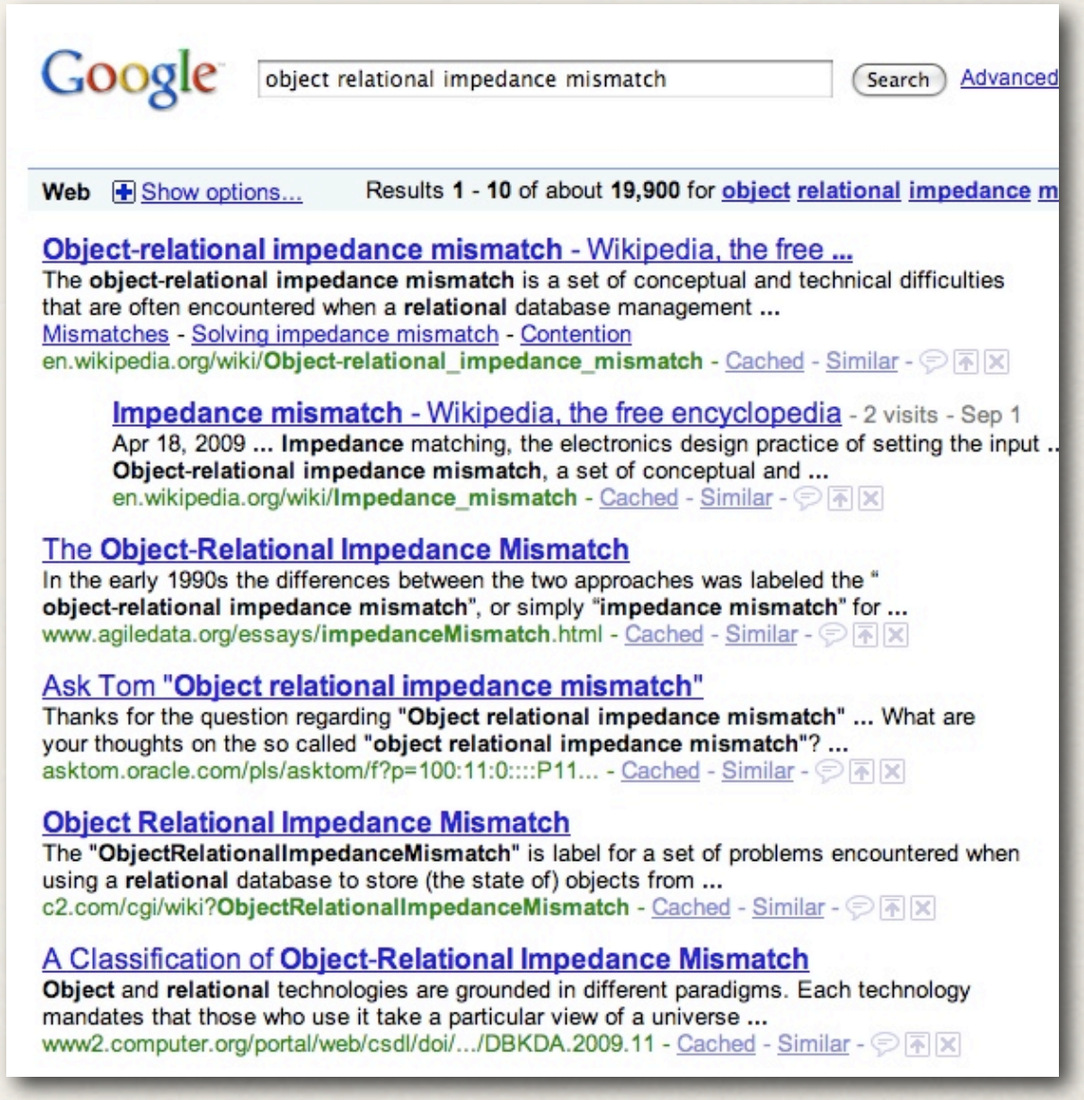
Pure Functions are Worry-Free

- Take/return ***values***
- Local scope
  - No remote inputs or effects
  - No notion of time
- Same arguments, same result
- Easy to understand, change, test, compose

- Huge benefits to using pure functions wherever possible
- In contrast:
  - Objects + methods fail to meet the "without thinking about them" criteria

But - many interesting programs aren't functions

- E.g. - 'google' is not a function
- Our programs are increasingly participants in the world
  - Not idealized timeless mathematical calculations
- Have observable behavior over time
  - get inputs over time
- We are building *processes*



“That ‘all things flow’ is the first vague generalization which the unsystematized, barely analysed, intuition of men has produced.”

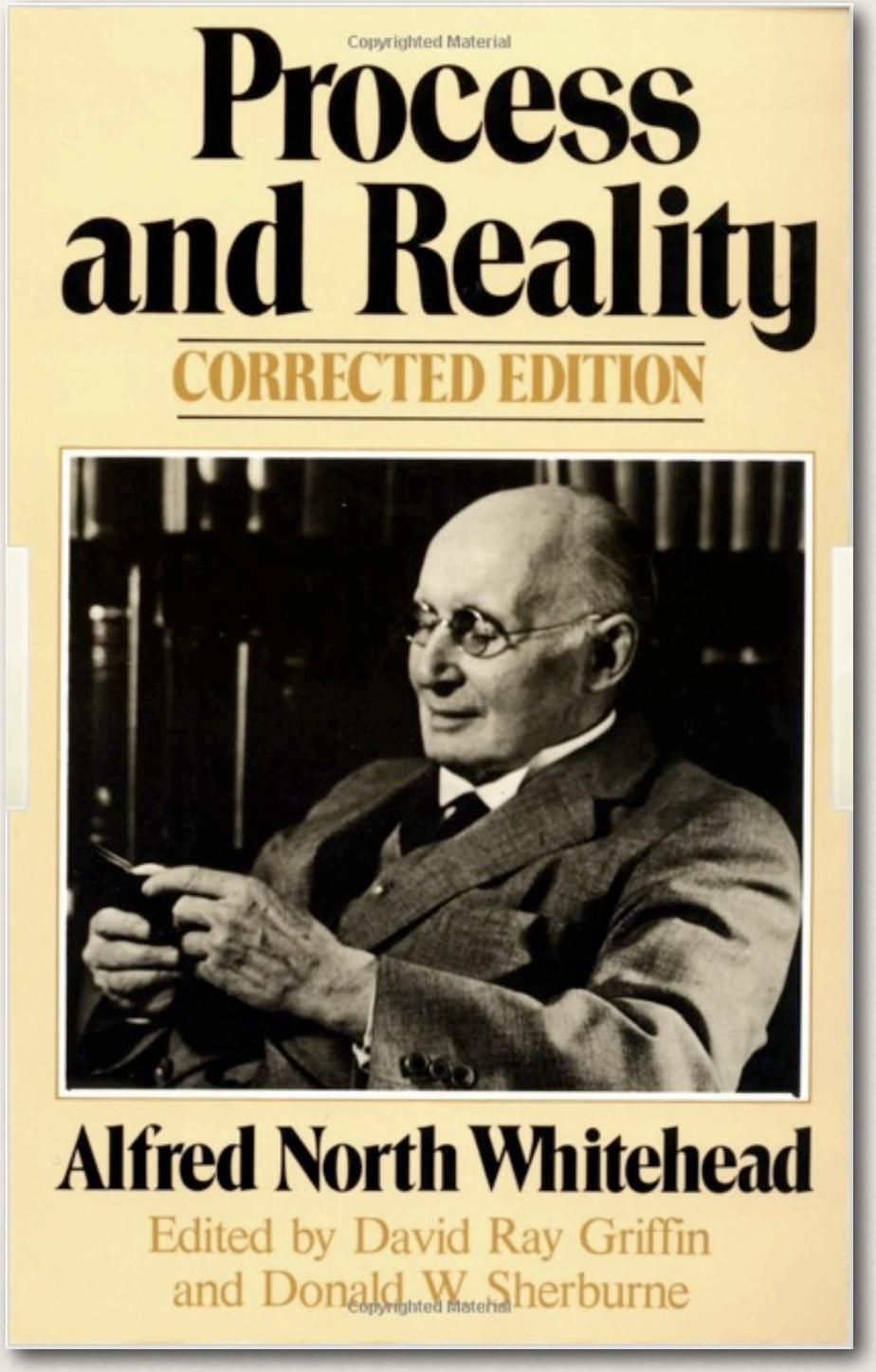
Alfred North Whitehead

OO and "Change"

- Object systems are very simplistic models of the real world
- Most embody some notion of "behavior" associated with data
- Also, no notion of time
  - Or, presume a single universal shared timeline
    - When concurrency makes that not true, breaks badly
    - Locking an attempt to restore single timeline
- No recipe for perception/memory - call clone()?

We have gotten this wrong!

- By creating objects that could ‘change’ in place
- ... objects we could 'see' change
- Left out time *and* left ourselves without values
- Conflated symbolic reference (identity) with actual entities
- Perception is fragile



“No man can cross the same river twice.”

Heraclitus

Oops!

- Seemed to be able to change memory in place
- Seemed to be able to directly perceive change
  - Thus failed to associate values with points in time
- New architectures forcing the distinctions more and more
  - Caching
  - Multiple versions of the value associated with an address
- Maintaining the illusion is getting harder and harder

A Simplified View (apologies to A.N.W.)

- Actual entities are atomic immutable values
- The future is a function of the past, it doesn't change it
  - Process creates the future from the past
- We associate ***identities*** with a series of causally related values
  - This is a (useful) psychological artifact
  - Doesn't mean there ***is*** an enduring, changing entity
- Time is atomic, epochal succession of process events

“There is a becoming of continuity, but no continuity of becoming”

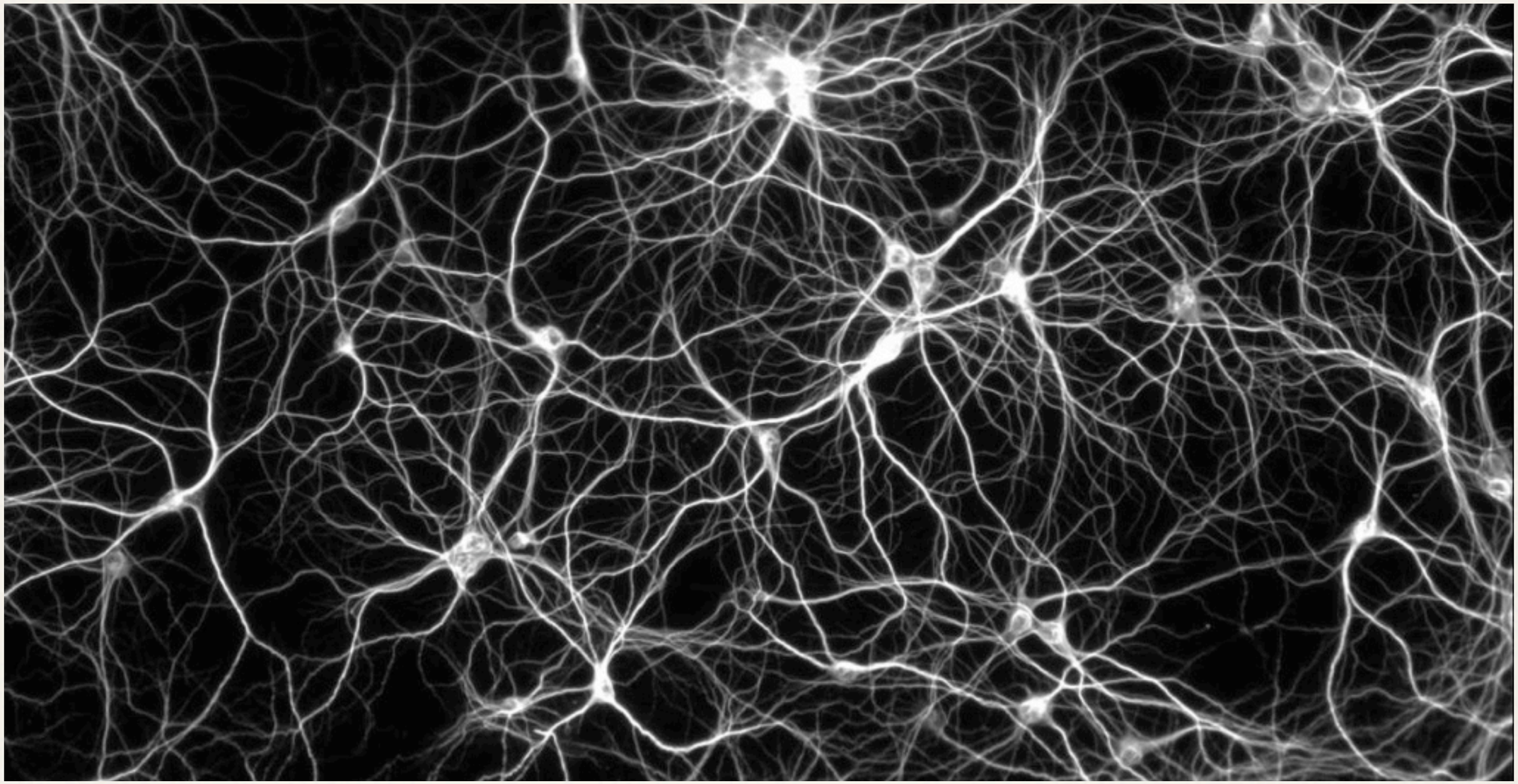
Alfred North Whitehead

Terms (for this talk)

- **Value**
  - An **immutable** magnitude, quantity, number... ***or*** ***immutable composite thereof***
- **Identity**
  - A putative entity we associate with a series of causally related values (states) over time
- **State**
  - Value of an identity at a moment in time
- **Time**
  - Relative before/after ordering of causal values

Why should we care?

- Our programs need to make decisions
- Making decisions means operating on stable values
- Stable values need to be:
  - Perceived
  - Remembered
- We need identity to model things similarly to the way we think about them
  - ***while getting state and time right***



We don't make decisions about things in the world by taking turns rubbing our brains on them.

Nor do we get to stop the world when we want to look around

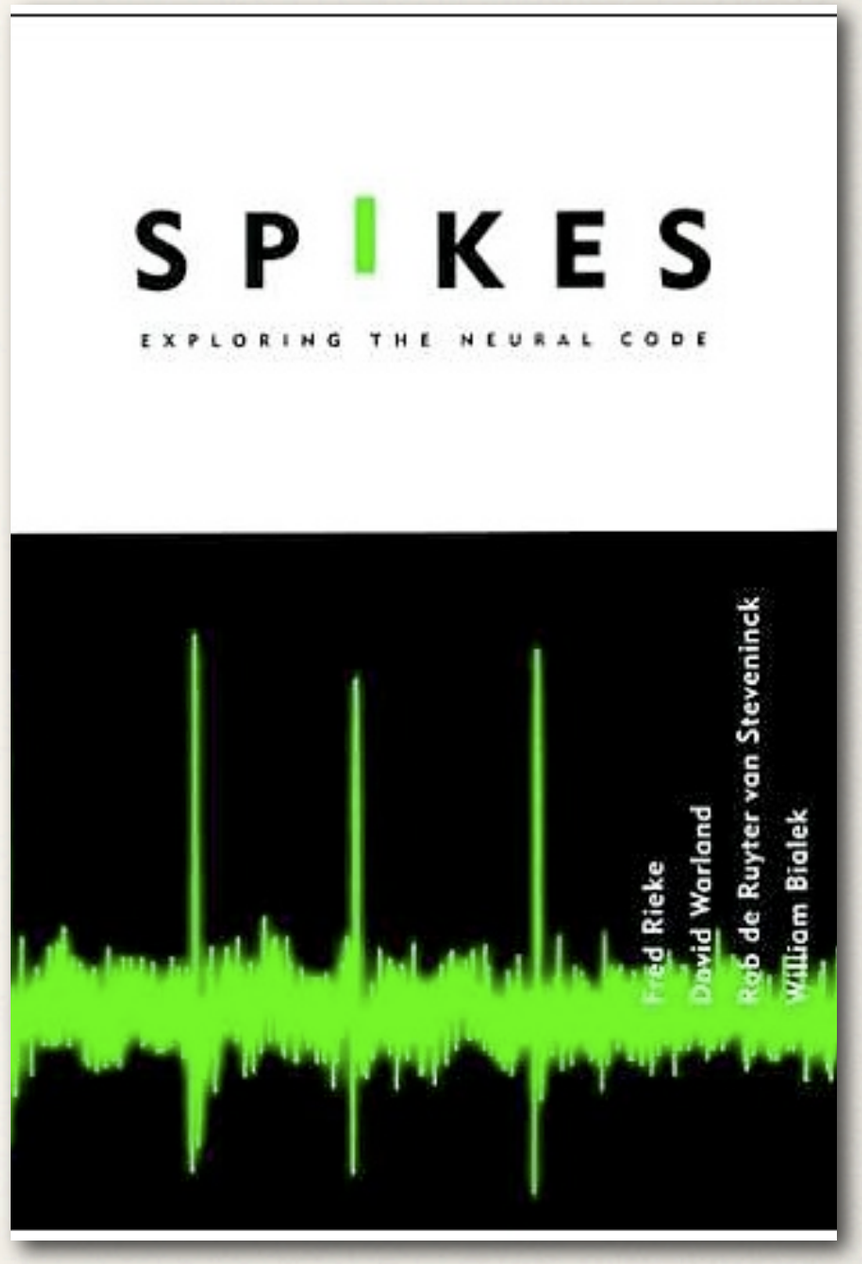
Perception is massively parallel and requires no coordination

This is not message passing!

Perception

- We are always perceiving the (unchanging!) past

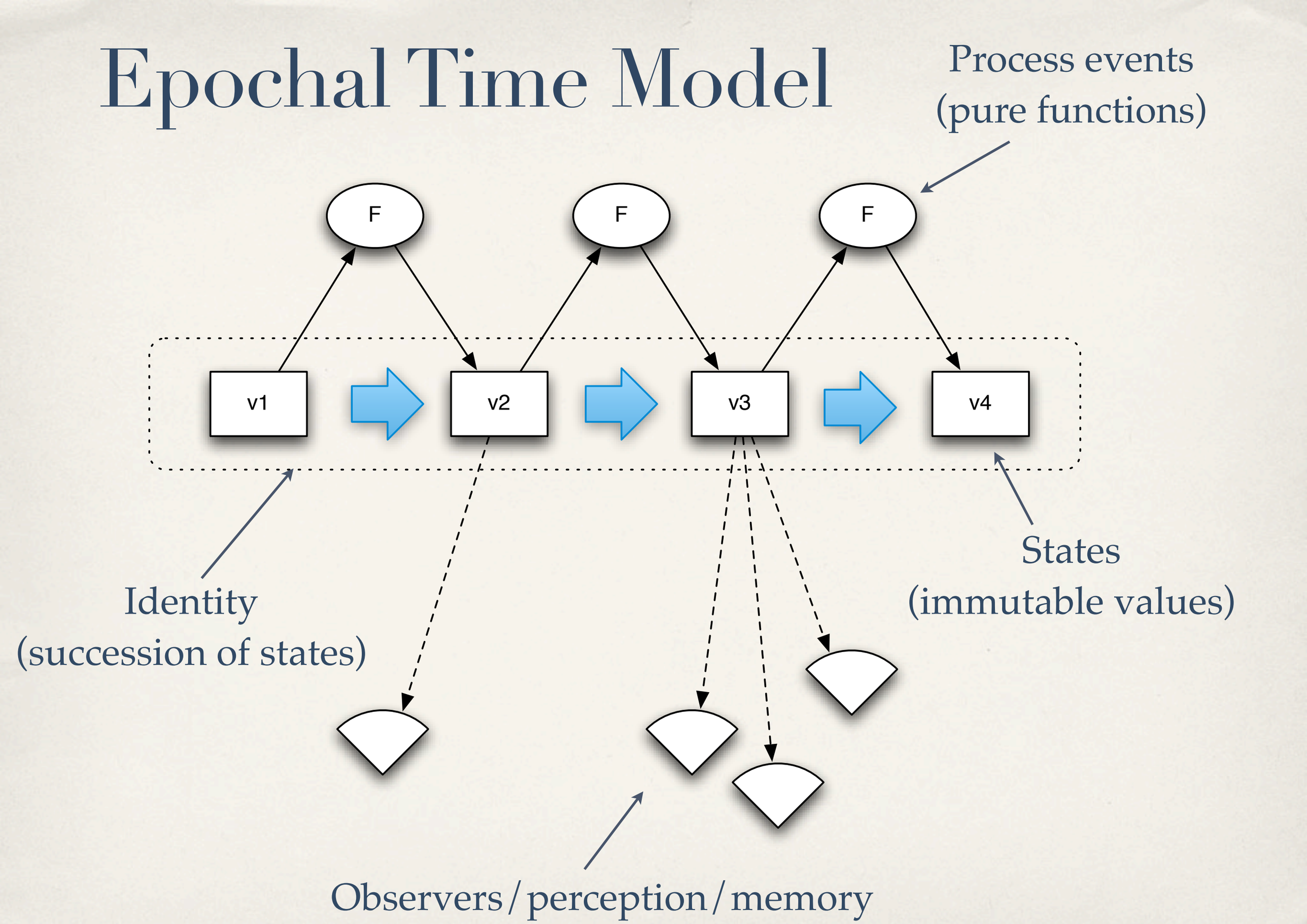
- Our sensory/neural system is oriented around:
  - Discretization
  - Simultaneity detection
- Ignoring feedback, we like snapshots



Action, in a place, must be sequential

Action and perception are different!

Epochal Time Model



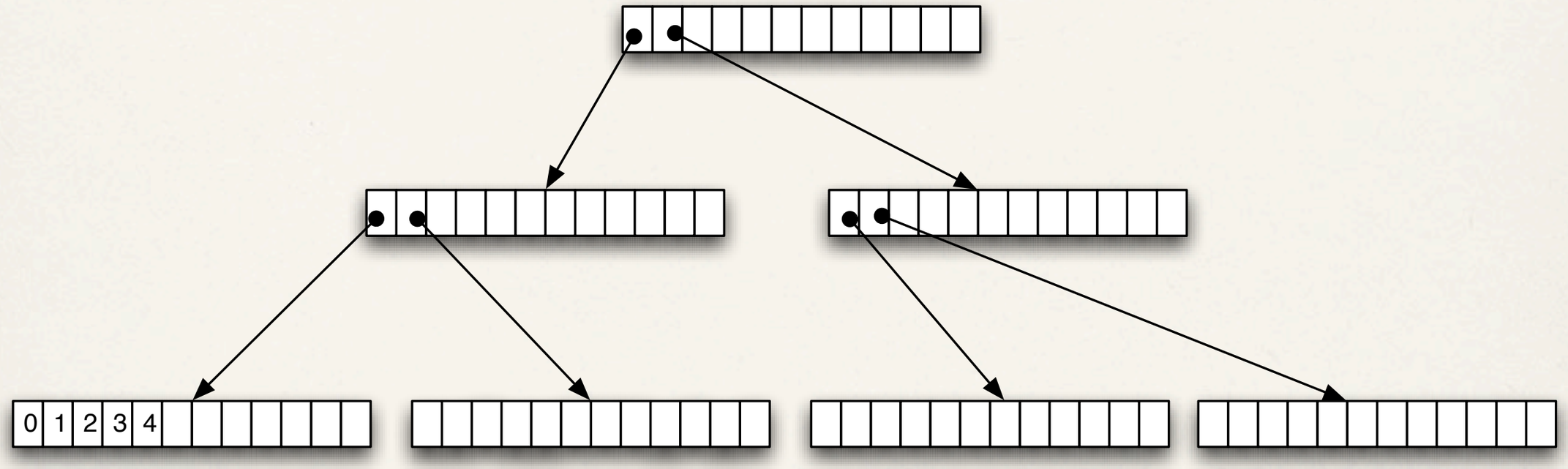
Implementation ideas

- We need language constructs that will let us efficiently:
  - Represent values. Create and share.
  - Manage value succession/causation/obtention
- We need *coordination* constructs to moderate value succession
  - Can also serve as identities
- We can (must?) consume memory to model time!
  - Old value -> pure function -> new value
  - Values can be used as perceptions/memories
  - GC will clean up the no-longer-referenced 'past'

Persistent data structures

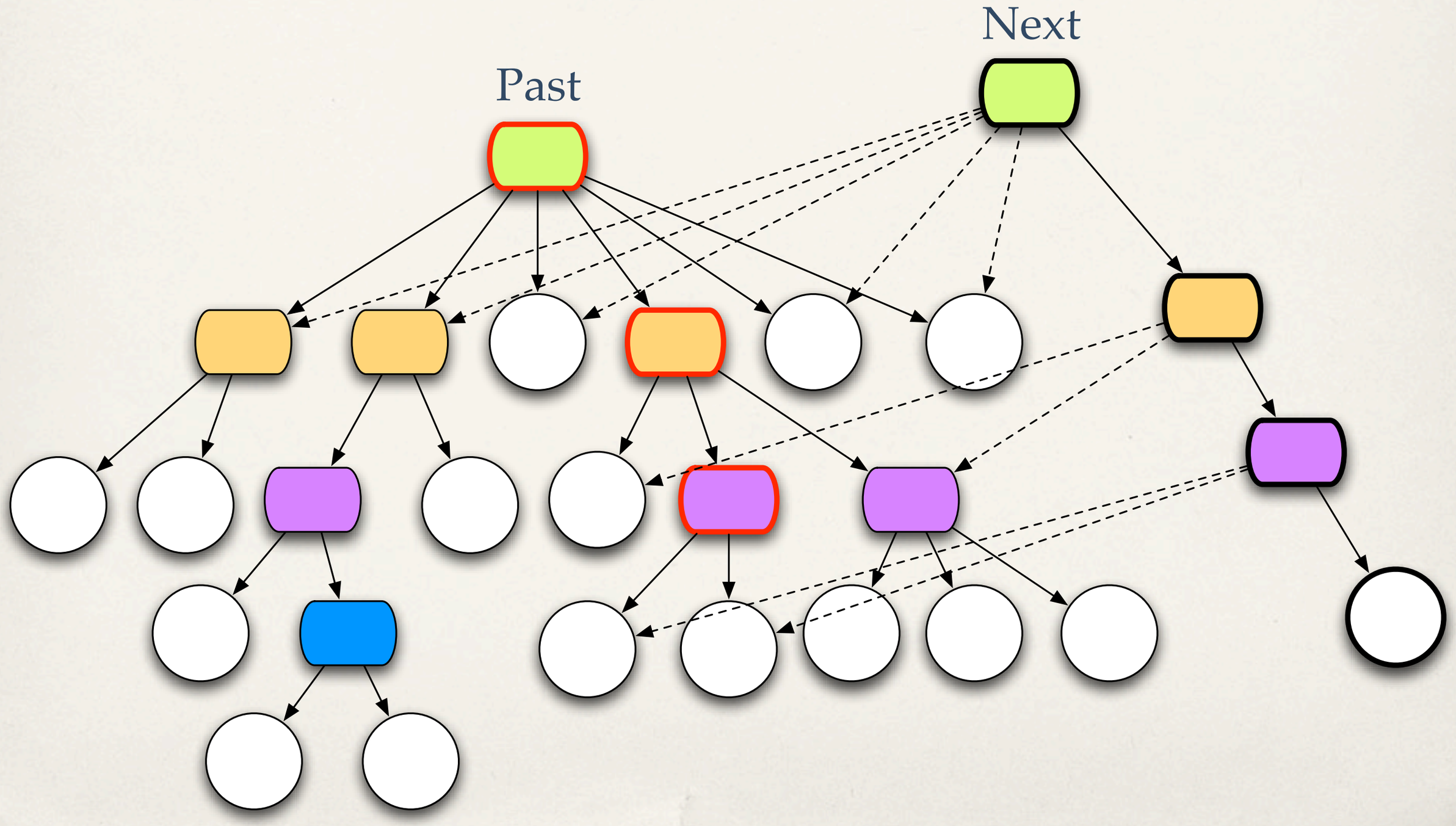
- Immutable
  - Ideal for states, snapshots and memories
  - Stable values for decision making and calculation
  - Never need synchronization!
- 'Next' values share structure with prior, minimizing copying
- Creation of next value never disturbs prior, nor impedes perceivers of prior
- Substantial reduction in complexity:
  - APersistentStructure foo();
  - Alias freely, make modified versions cheaply
  - Rest easy, stay sane

Trees!



- Shallow, high branching factor
- Nodes use arrays
- Can implement vectors and hash maps/sets etc

Structural Sharing



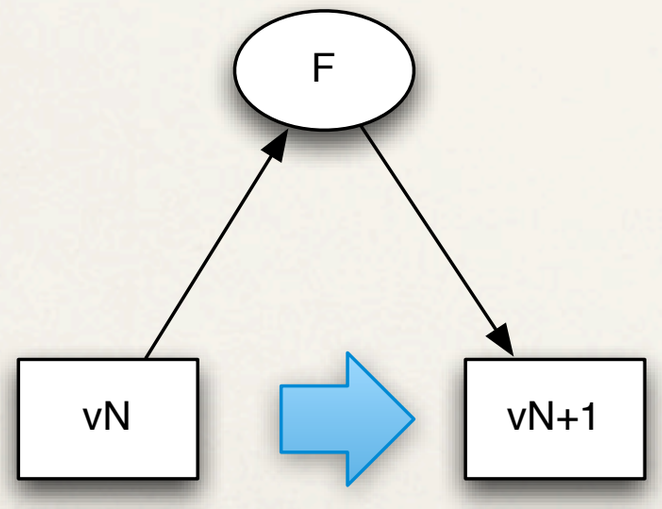
Declarativeness and Parallelism

- Performance gains in the future will come from parallelism
- Parallel code needs to be declarative - no loops!
  - map/reduce etc
- Parallel code is easier when functional
  - else will get tied up by coordination
- Tree-based persistent data structures are a perfect fit
  - Already set up for divide and conquer and composable construction
- IMO - These should be the most common data structures in use, yet almost unused outside of FP

“It’s the performance, stupid!”

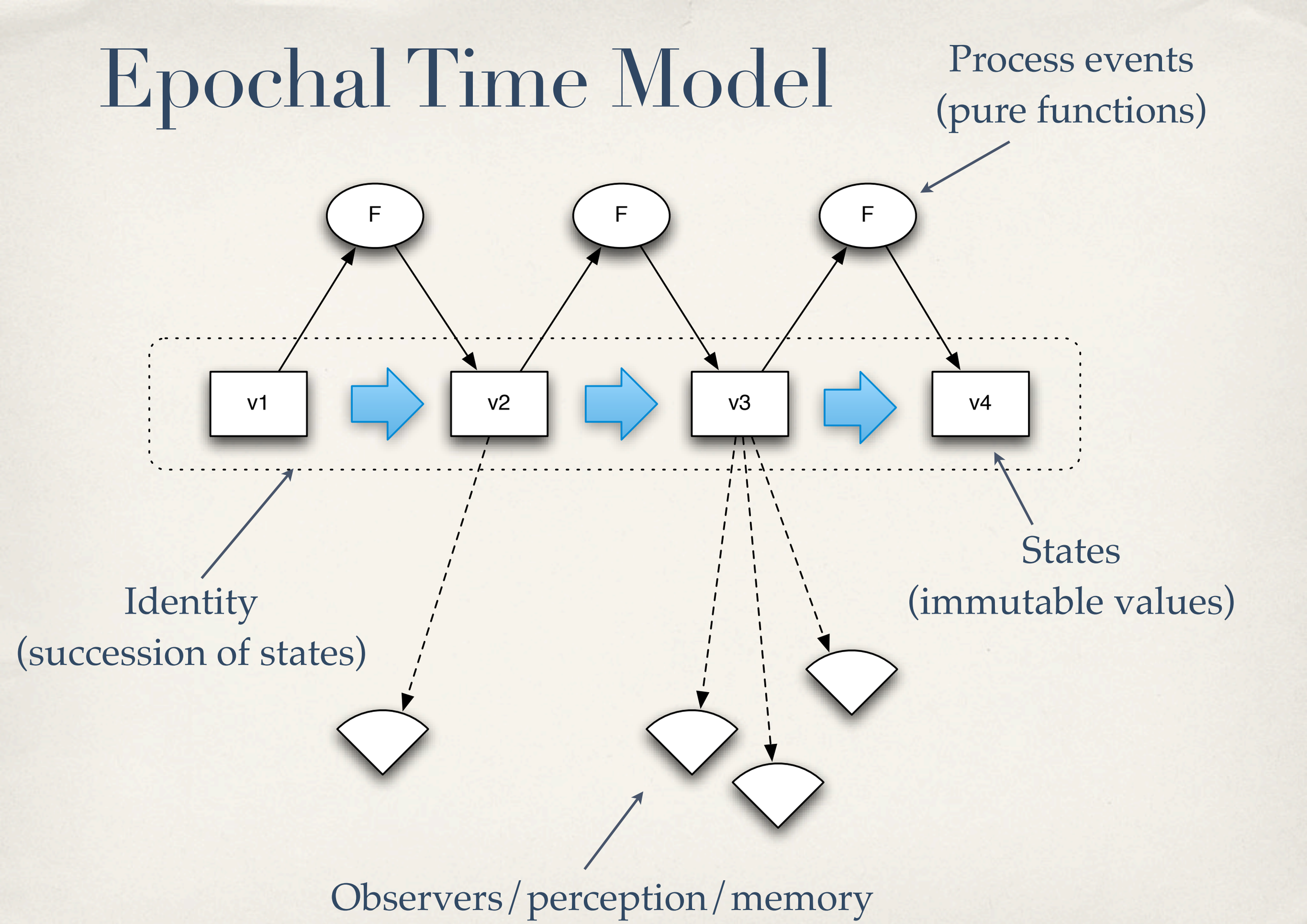
the Audience

- Persistent data structures ***are*** slower in sequential use (especially ‘writing’)
- **<u>But</u>** - no one can see what happens inside F



- I.e. the ‘birthing process’ of the next value can use our old (and new) performance tricks:
  - Mutation and parallelism
- Parallel map on persistent vector same speed as loop on j.u.ArrayList on quad-core
- Safe ‘transient’ versions of PDS possible, with O(1) conversions between persistent/transient

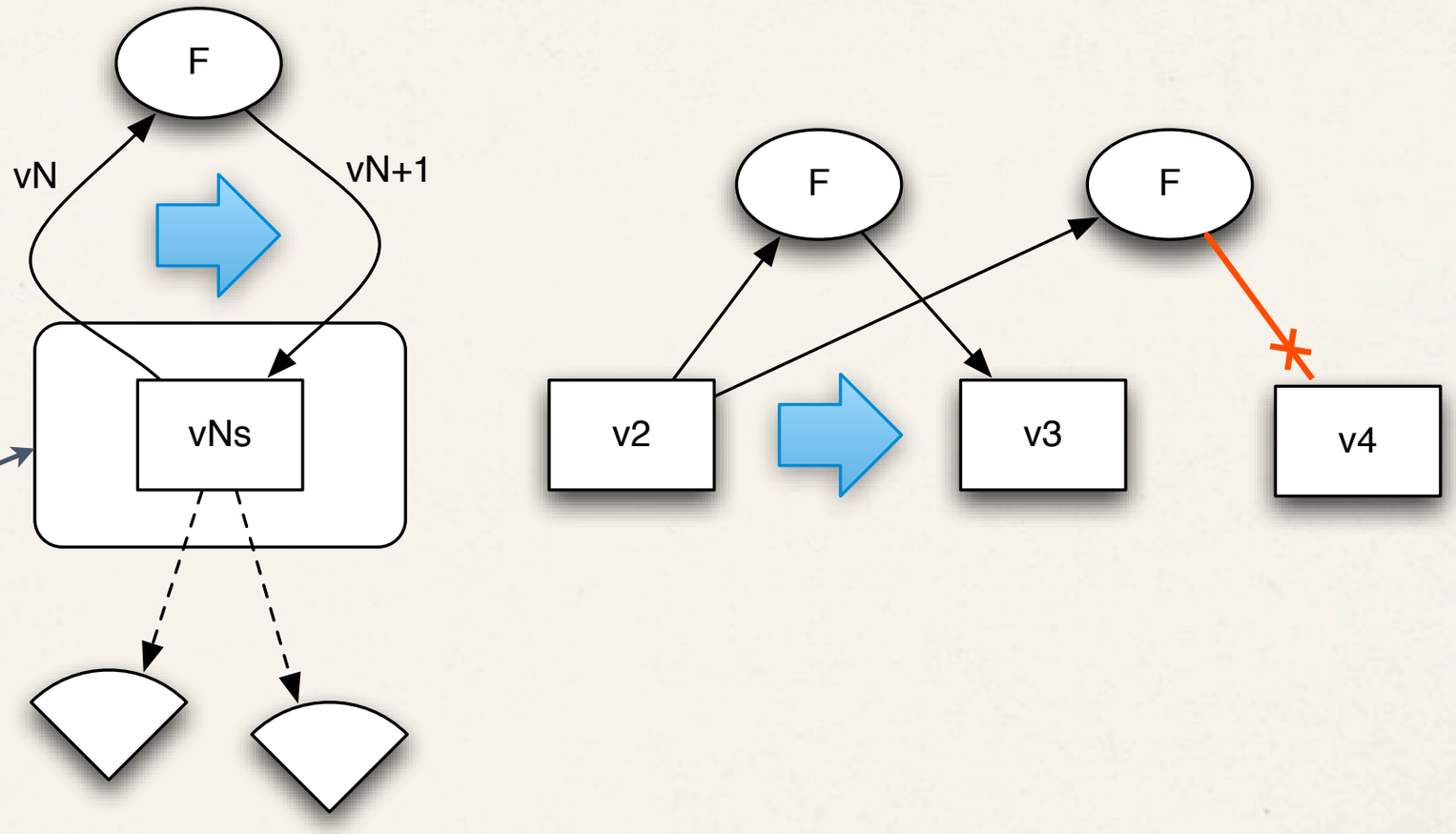
Epochal Time Model



Time constructs

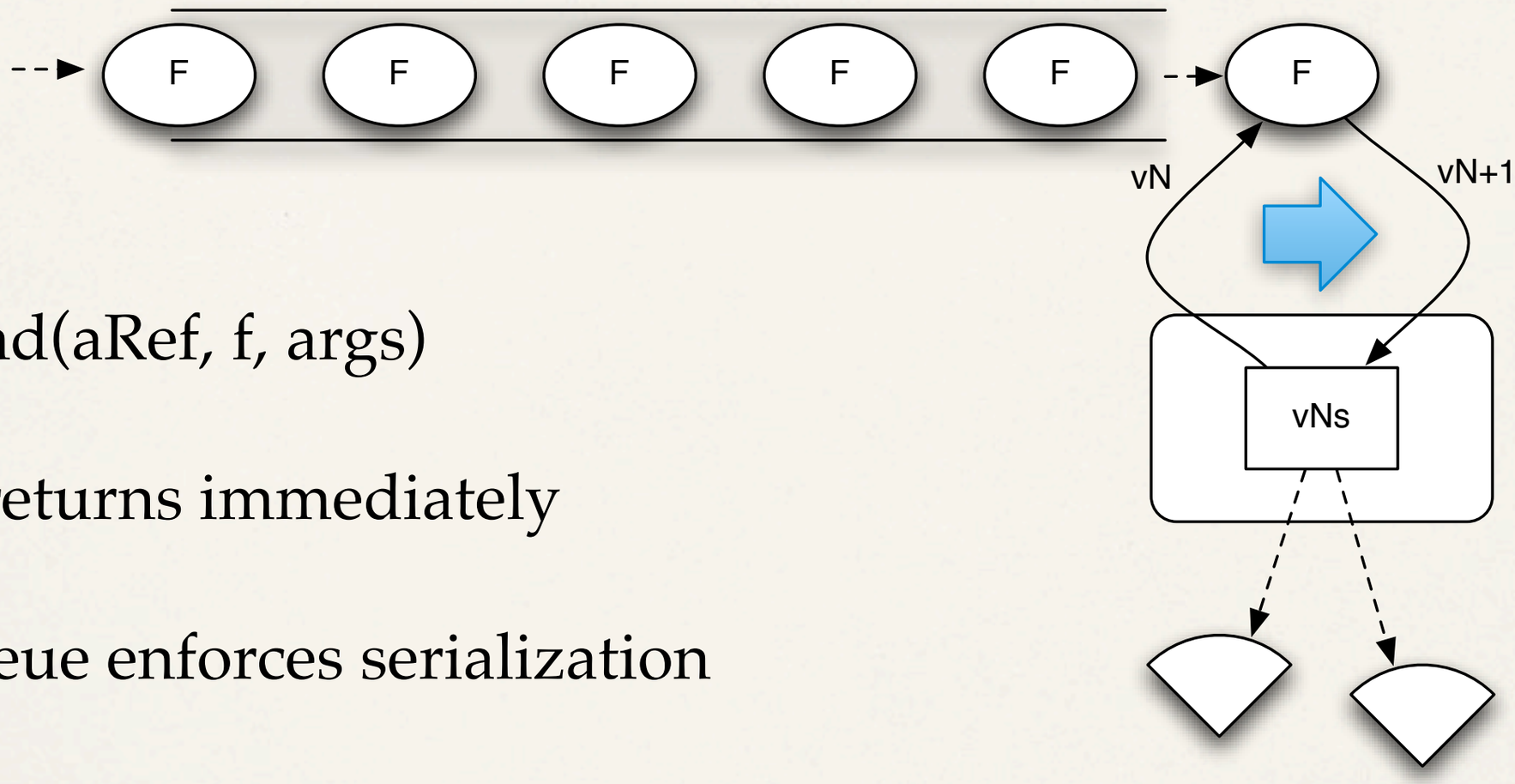
- Need to ensure atomic state succession
- Need to provide point-in-time value perception
- Multiple timelines possible (and desirable)
- Many implementation strategies with different characteristics/semantics
- CAS - uncoordinated 1:1
- Agents - uncoordinated, async. (Like actors, but local and observable)
- STM - coordinated, arbitrary regions
- Maybe even ... locks?
  - coordinated, fixed regions

CAS as Time Construct



- 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

Agents as Time Construct

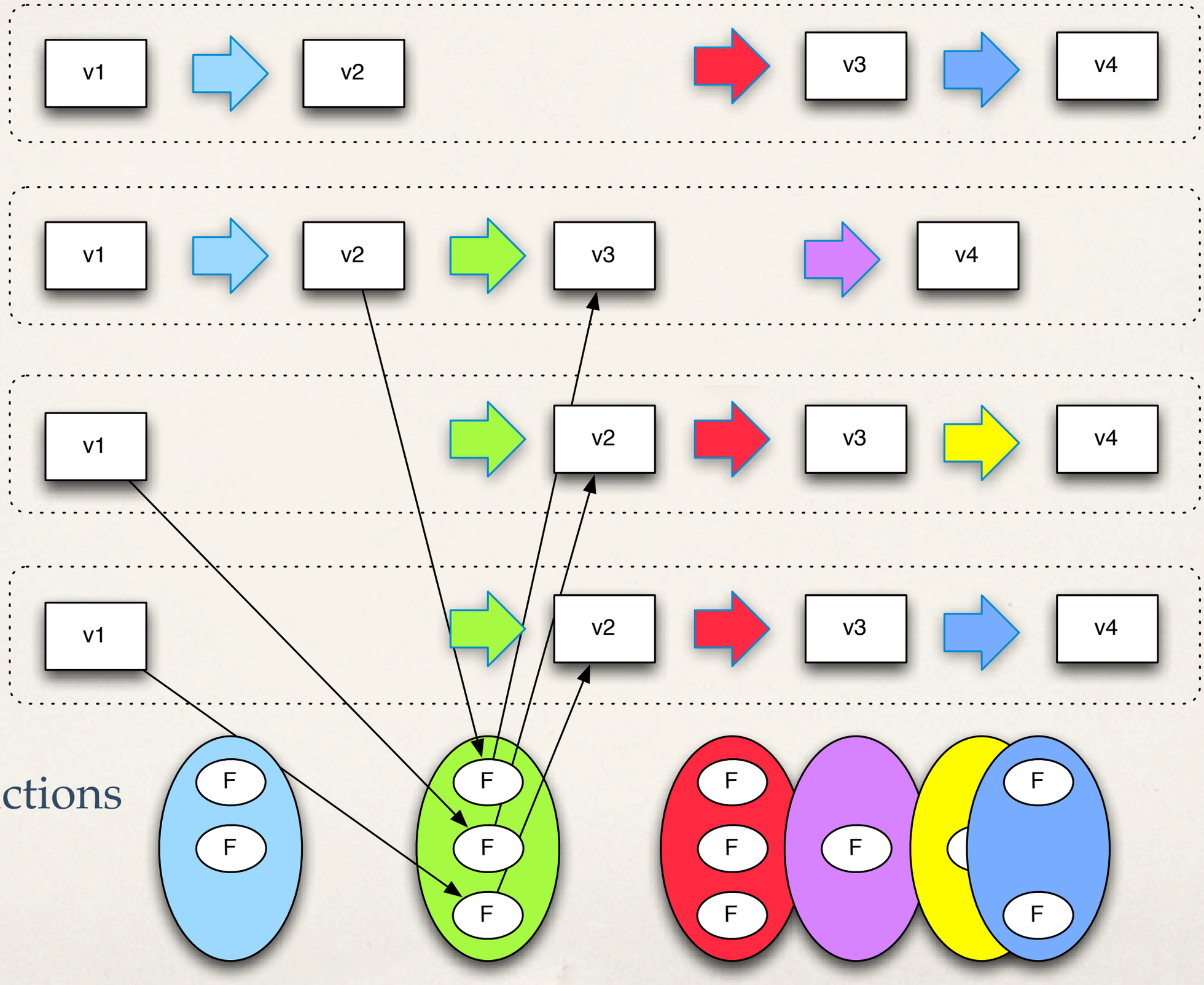


- send(aRef, f, args)
  - returns immediately
- queue enforces serialization
  - f(vN, args) *becomes* vN+1
  - happens asynchronously in thread pool thread
- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception

STM

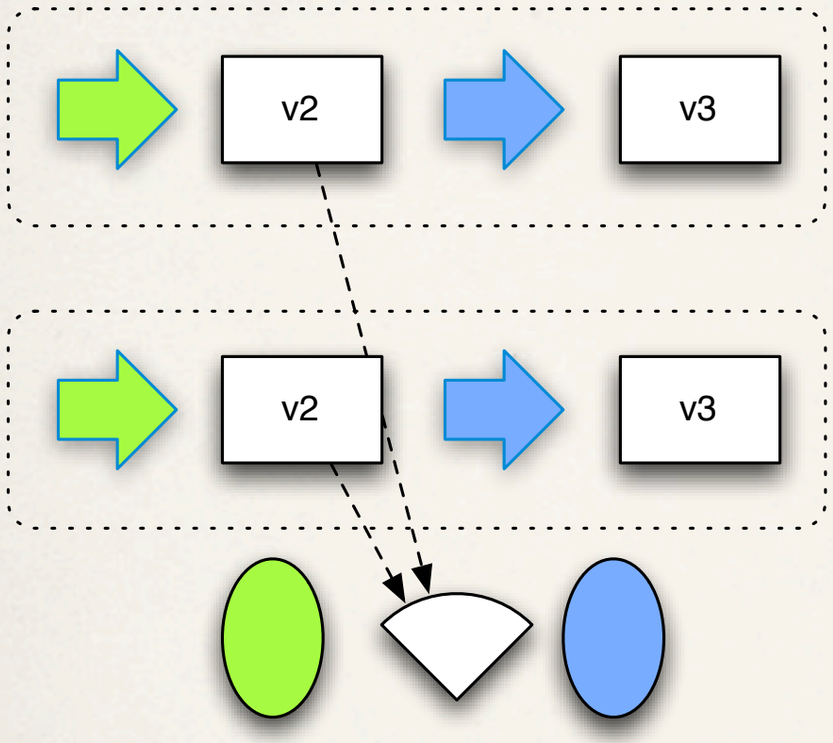
- Coordinates action in (arbitrary) regions involving multiple identities/places
- Multiple timelines intersect in a transaction
- ACI properties of ACID
- Individual components still follow functional process model
  - f(vN, args) *becomes* vN+1

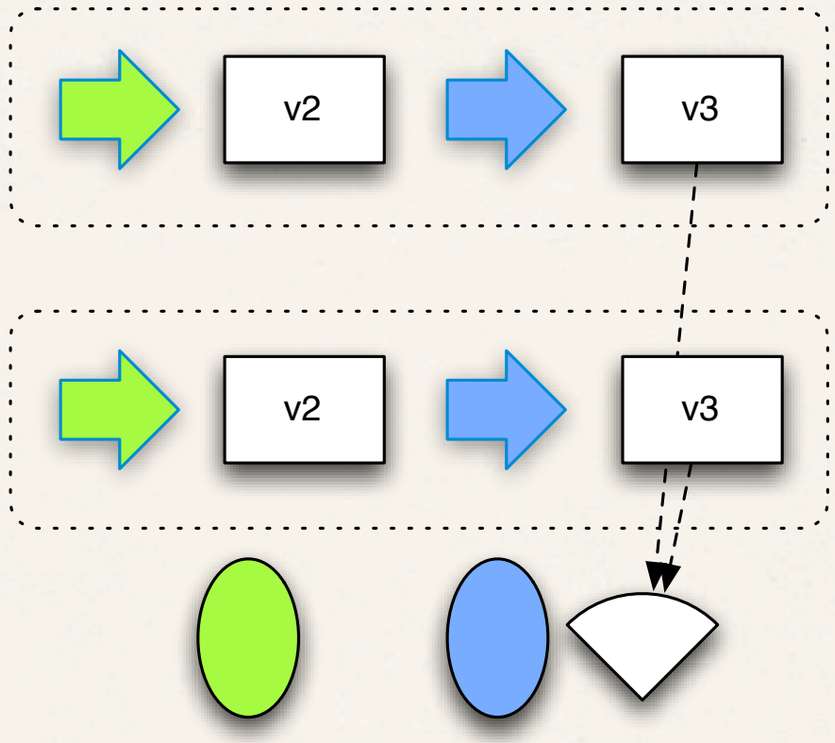
STM as Time Construct



Transactions

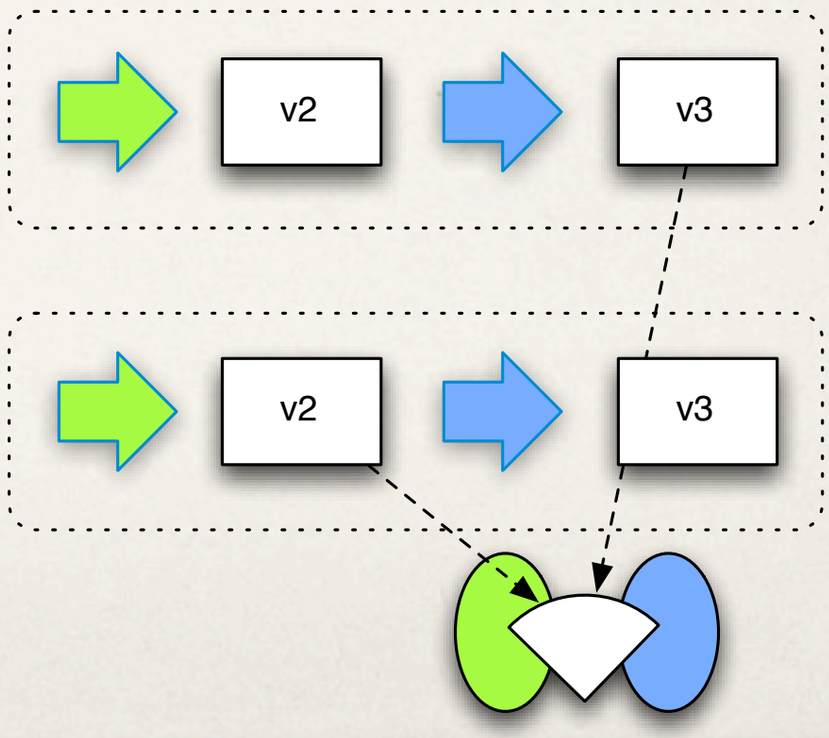
Perception in (MVCC) STM





Transactional snapshots

Non-transactional scans



Multiversion concurrency control

- No interference with processes
- Models light propagation, sensory system delay
- By keeping some history
- Persistent data structures make history cheap
- Allows observers/readers to have timeline
  - Composite snapshots are like visual glimpses, from a point-in-time in the transaction universe
  - Free reads are like visual scans that span time

STMs differ

- Without MVCC you will either be:
  - limited to scans
  - back to “stop the world while I look at it”
- Granularity matters!
  - STMs that require a transaction in order to see consistent values of individual identities are not getting time right, IMO

Conclusions

- Excessive implicit complexity begs for (and sometimes begets) change
- The conflation of behavior, state, identity and time is a big source of implicit complexity in current object systems
- We need to be explicit about time
- We should primarily be programming with pure functions and immutable values
- Epochal time model a general solution for the local process
- Current infrastructures (JVM) are sufficient for implementation

Future Work

- Coordinating internal time with external time
  - Tying STM transactions to I/O transactions
    - e.g. transactional queues and DB transactions
- Better performance, more parallelism
- More data structures
- More time constructs
- Reconciling epochal time with OO - is it possible?

"It is the business of the future to be dangerous; and it is among the merits of science that it equips the future for its duties."

Alfred North Whitehead