Are We There Yet?

- **Speaker: Rich Hickey**
- **Conference: JVM Language Summit 2009 - Sept 2009**
- **Video: http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey**

[Time 0:00:00]

```
slide title: Are We There Yet?


A deconstruction of object-oriented time

Rich Hickey
```

So I am going to talk about time today. In particular, how we treat time in object-oriented languages generally, and maybe how we fail to. So I am trying to provoke you today to just reconsider some fundamental things that I just think we get so entrenched with what we do every day, we fail to step back and look at what exactly are we doing.

[Time 0:00:34]

```
slide title: 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?
```

So are we being well served by Object Orientation as commonly embodied? The concept is pretty broad, and there are multiple possible embodiments, but it ends up the ones that we have have a lot of consistent attributes.

Do we all agree this is the best way to write software? Do we think this will continue to be the best way?

[Time 0:00:55]

```
slide title: 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 ...

[ Photo of traffic jam on a highway with mostly similar cars, varying
in color and a little bit in shape. ]
```

Certainly today, this is a really entrenched model. It does not matter which language you are using. Everybody has different languages: Groovy, and whatnot, Scala, and Java, and people use C#, and they love the differences between these languages. And I want you to focus on the *similarities* between these languages, which is they are all single-dispatch, stateful, object-oriented languages.

And they have a lot of the same kinds of things: some notion of classes, inheritance. Fields are an interesting concept. Methods are more interesting, and we will talk about them later. They are all garbage collected, and they have a heritage that goes back to languages like Smalltalk.

[Time 0:01:41]

```
slide title: 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

[ Photo of empty road leading off into the horizon. ]
```

They are not significantly different in some dimensions. They are superficially different. They might have mix-ins. They might have interfaces. Even static and dynamic typing, I think, is not nearly as important as some of the underpinnings that they share. Everybody is so excited because now there are languages without semicolons and other great choices that we have.

But they have more to do with the sensibilities of the programmer than they have to do with significant differences in the programming model. So they are all different cars, but they are all on the same road.

[Time 0:02:17]

```
slide title: 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.
```

Is this the end? Are we done? Are we going to keep making languages that are just very, very slight incremental differences to the things that we know? Certainly, one thing is undeniable: people like Object Orientation.

On the other hand, I think we have gotten increasingly conservative, which makes sense. Of course, you get adopted by large companies. They have big investments. People know how to do it. It is not something you are going to move away from any too readily.

And certainly, I want to emphasize, the purpose of this talk is not to beat up on OO, but to have everybody just take a step back. Just imagine you do not love it, if you do, and think about whether or not it is perfect.

[Time 0:03:09]

```
slide title: 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?

[ Photo of a road sign with wavy arrow indicating "curves ahead". ]
```

When we look at languages and try to think of: "If I could write another language, or if I could fix this language, or if I could add a feature to the next version of the language", what would we do? Why do we add things? What drives us to make changes, or what drives us to change cars to say, "I am going to stop using this language and adopt this other language"? And what things *should* drive us to that? I do not think a lot of people say, "Oh, I am tired of semicolons. I cannot do it anymore. Or curly braces, or something. I am going to switch to something easier."

I think static and dynamic may cause people to switch, but I think there are examples already in our history that show us what causes us to switch.

[Time 0:03:52]

```
slide title: Some Critical Ideas

+ Incidental complexity
+ Time / Process
+ Functions / Value / Identity / State
+ Action / Perception
```

So the things I am going to talk about today are a small subset of the kinds of things that I think you should think about when you look back at the language you are using, and try to decide whether or not you want to do something differently.

I want to talk about complexity today. I want to talk about time. Mostly about time. And then about models we can use to better implement time. And some of the principles that underlie Object Orientation. It is a modeling concept. It is based around we can sort of do things in our programs that are similar to what we see in the world, and that helps us understand our programs.

[Time 0:04:30]

```
slide:

"Seek simplicity, and distrust it."

Alfred North Whitehead
```

So the hero of the talk today is Alfred North Whitehead. He is the famous guy who with Russell wrote Principia Mathematica. Subsequent to that, he also became a philosopher. And he wrote some great things, and I am just going to put them up here because they are great.

So the first thing is: distrust simplicity. I do not want to talk actually about the complexity of the problems we are trying to solve. We all know we are given increasingly more complex problems to solve, bigger problems, more data, more flexibility. Expectations of people for software will only ever increase.

[Time 0:05:09]

```
slide title: 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
```

The complexity I want to talk about today is the incidental complexity. The complexity that arises from the way our tools work, from the ideas that embody our tools, from the ways our tools do not work, from the ways our approaches do not work. These things all become problems that we have to solve. And we have a certain number of hours in the day we have to solve problems. Are the problems you are solving the problems of the application domain or the problems you have set in front of yourself by choosing a particular language, or tool, or development strategy? So that is incidental complexity. It is coming along for the ride. It is not part of the problem you are trying to solve.

And it is worse, I think. I mean, everybody knows when something is complex and you look at it, it says, "Arrr!!! Complex!" And everybody says, "OK. Well I see that. That is scary. I know that is a danger zone. I know I am going to be careful with that."

The worst kind of incidental complexity is the kind that is disguised as simplicity. "Look how easy this is! There are no semicolons." I do not want to beat up on no semicolons. It is just an easy way to say, "Look at some superficial aspect of the language I am using, this seems easy, this seems familiar". But is there incidental complexity hiding underneath it?

[Time 0:06:25]

```
slide title: 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
```

So this is an example. Again, not to beat up on C++, but I spent more than a decade doing this. So it is not that hard. I mean, if you get into template metaprogramming, it can get hard. But the basics are pretty simple. You can write a function that returns a pointer. What is wrong with that? It is pretty simple. There is new, and delete, and there is pointers, and you can pass them around, and you can dereference them. There is only like five things you need to know. You can learn them in an afternoon.

So is it really simple? For instance, the same syntax is used to refer to things on a heap and things that are not on a heap; these pointers.

But it gets worse. And the *real* problem with that function signature is: what do you do with the thing that you get when you call it? Is it yours? Is it now your responsibility? Do you have to delete it later? Is it even something that can be deleted? Can you hand it to somebody else? Is that allowed? Could you save it?

So the problem there was: there is no standard automatic memory management. There is no garbage collection. And this was, and still is, for people using this language, a big source of incidental complexity. Because managing memory is on you. You do not see that. There is not a sign on the top of

your source code, "do not forget; managing memory is on your head." This is incidental complexity. You have to just know that. It is not in the source code.

And it is a big problem. I think the lack of garbage collection really impeded C++ in one of its design objectives, which is: it was supposed to be a library language. All the original design stuff and any time you heard Stroustrup talk about it, it is like, "C++ is going to be a library language". But it only ever ended up being a parochial library language. Every shop had a library, but there were not a library, and still are not a lot of libraries, that go between places, because of this problem.

And we know that Java, having garbage collection, has this *huge* library infrastructure. So I think people that moved from C++ to Java did so in no small part due to the fact that they were no longer willing to bear this implicit complexity. I do not want to do manual memory management. It is not part of the problem I am trying to solve at all. It is just another problem on my plate every day when I go to work, and I do not want to do it.

[Time 0:08:52]

```
slide title: 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
```

So let us look at Java. It is easier; there is no asterisk. This is like even better.

[Audience laughter]

So what is the problem with this? It is simpler. It is definitely simpler. Now we only have references to managed memory, and we have automatic memory management. We have garbage collection. This is much, much better. It is much easier.

Except, again, we have this hidden complexity. Is this a mutable thing or not? When will I see a consistent value? If I look at this right now and I walk through its fields, will the sum of the things I have seen represent a consistent value? This is not just a concurrency problem. There *is* a concurrency problem, and it is a big one.

But even before we had threads and all that part, this is a big source of incidental complexity in programs, because we do not know when we have a stable value. Can I store this date off and look at it later, and know I am going to see what I saw when I was handed it? You do not know.

In addition, if you hand a Date or some mutable thing – I know the mutable things have been all deprecated. They are fixing Date or whatever. I am not trying to beat up on Date. But if you hand a mutable thing to somebody, and they may hand it to other people, and then you need to change it, who is going to be affected by that? You have no idea.

So this looks really easy. Now this is true. This is not just Java. This is every single language I listed that allows for mutable objects has this problem. And there is no way to fix it.

So what is the problem here? I am going to say the problem here is: we do not have any standard time management. That may be a really confusing thing. Hopefully, it will not be as we go along.

[Time 0:10:55]

```
slide title: 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
```

So this is kind of a little bit of a reiteration of the points I was making before. I think that because we are so familiar with this, we are absolutely, completely blind to it. And when we choose languages, or when people choose different languages, a lot of times they make the decisions on very superficial differences like the syntax, or perhaps sort of this makes me feel good, expressivity differences, which I admit completely are real and valid, but they are somewhat emotional.

In the meantime, our systems are getting very, very hard to build, maintain, and make correct. And in no small part, that is due to this incidental complexity. We cannot understand big programs. We have these giant test suites. And we run them every time we change any little thing. Because we do not know if we change something over here, that it is not going to break something over there. And we cannot know.

And I think for me, and I think for many people, we are going to find concurrency just is the straw that breaks the camel's back in this area.

[Time 0:12:08]

```
slide:

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

Alfred North Whitehead
```

So we are programmers. We do not use assembly language anymore. We have languages. Each time we build a new language, or we use a new language, we are expecting some benefits in this area. We want to hide chunks of stuff, name them, encapsulate them, get them out of our way so we do not have to think about them, and we can build something on top of that.

I mean, somebody who is building houses out of bricks does not need to worry about the inside of bricks. They have certain properties, they have certain expectations. And I think it is one of the selling points of Object Orientation, that this is a way to make these kinds of units that we can combine to make programs that are easier to understand. Because we understand the pieces. And we put the pieces together, and we get something we can understand.

[Time 0:13:06]

```
slide title: Pure Functions are Worry-Free

+ Take / return _values_           + Huge benefits to using pure
+ Local scope                        functions wherever possible
  + No remote inputs or effects    + In contrast:
  + No notion of time                + Objects + methods fail to
```

```
+ Same arguments, same result          meet the "without thinking
+ Easy to understand, change,          about them" criteria
  test, compose
```

It ends up that they are really not the best unit for that. The best unit for that are functions. And in particular, pure functions. If you want something you do not have to worry about, you should love the pure function. The pure function takes immutable values. It does something with them. That stuff it does has no effect on the world, and no connection on the rest of the outside world. Then it returns another immutable thing.

So the entire scope of its activity is local. It has no notion of time. That is going to become important later. But it is definitely easy to understand. It is easy to change. There is some signature. That is the *only* thing about it anybody else knows. When we change the insides, nobody cares. Pure functions are, and should be, the bricks that we use, because they are the things we can use without worrying about them most readily. There are definitely huge benefits from doing this. I think you could easily do it in object-oriented languages, but people do not.

In contrast, objects and methods do not have this property. They do not have the "I do not need to think about them" property. They definitely do not. And we are going to see why in a minute.

[Time 0:14:22]

```
slide title: _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_

[ Image of Google search results web page. ]
```

On the other hand, as great as functions are as building blocks, our programs in general are not functions. There are programs that are functions. There are compilers and theorem provers. Take this stuff and convert it or whatever.

But a lot of programs run indefinitely long, and people have an expectation of being able to see their behavior, to have inputs as the program runs, and get something different every time.

I do not want Google to return the same result every time I type the same word into it. Google would not work for me then. If Google was a function, it would be no good. Google is a process that is connected to the rest of the world. It is scouring pages and integrating them and forming algorithms, which hopefully also should change. As a whole, it feels much more like a participant in the world than a function any more. It is not an idealized calculation.

So we can say that the entire program has this behavior we can observe over time, although you will see I do not like the word "behavior". So most programs that most people work on in industry are processes.

[Time 0:15:34]

```
slide:
```

```
"That 'all things flow' is the first vague generalization which the
unsystematized, barely analysed, intuition of men has produced."

Alfred North Whitehead
```

So maybe we have not seen the value of functions. I certainly do not think we have. But we also have seen the limitations. Object Orientation was a way to say, "Well functions are great, and they are great for calculations and all those stuff. But then I see the real world, and there are objects, and there are windowing systems, and there are things." And Object Orientation was a way to say, "All right. Well how do we take our mental model for the processes we see in the world, and embody them in some kind of programming model?"

And so the essence of the Object-Oriented programming model is not encapsulation, blah, blah, blah. It is really that behavior, that flow-like thing. We have these entities we see doing things in the world. We should have entities that do things in our programs.

[Time 0:16:17]

```
slide title: 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()?
```

So the first thing we should realize: that any programming model that tries to model the real world is essentially going to be a simplistic thing. But again, there is that "beware of simplicity". Is this thing too simple to do the job correctly? One of the problems with object-oriented time is that we talk about behavior and state and things like that really, really loosely. These terms are almost completely meaningless.

And in addition, even though objects putatively are about process, there is no notion, no concrete notion of time in objects. No more so than there are in functions. But at least functions are not pretending to play with time. Functions say, "There is no time. There are my inputs and my outputs. I am not pretending to deal with time."

Objects are pretending to deal with time. And yet, our object systems do not have any reified notion of time. There is nothing you can talk about explicitly, because most of them were born in a day when your program ruled the computer. You had a single monotonic execution flow and it just did what it wanted; do this, do that. There was a single universal process controlling everything.

Now that that is no longer true, we try to use locks to restore that vision of the world. But that vision of the world was *never* correct. And you can tell in one key way: because we still, even with all the locks and everything else, we still do not really have a concrete representation we can use for perception. Can I look at something and see it be stable? Or memory. Can I remember that? These objects are all live. They are time bombs. We have gotten this wrong. The object-oriented model has gotten time wrong.

[Time 0:18:16]

```
slide title: 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

[ Image of cover of the book "Process and Reality", Corrected Edition,
by Alfred North Whitehead ]
```

And we have done so in a couple of ways. The first is: we have made objects that can change in place, and we have made objects that we could *see* change in place. As I have said, we left out any concrete notion of time. And, there is no proper notion of values. You can fabricate values. You can make a class that has all immutable components, and that would constitute a value. But there is no proper notion of value in a lot of these languages.

The biggest problem we have is we have conflated two things. We have said: the idea that I attach to this thing that lasts over time, *is* the thing that lasts over time. And that is not actually true. In addition, as I said before, our ability to perceive is fragile.

So I have the hero of the day, Whitehead, up here. Who, subsequent to doing all the Principia Mathematica stuff, as I said, became a philosopher. And he tried to concern himself with: how does the world actually work, informed by the current knowledge – which this was back in the '20s – of quantum mechanics and relativity. And one of the things that he came up with was the fact that time must be atomic and move in chunks. And in fact, time is not actually a real thing you can touch, but it is something that you derive from seeing these epochal transitions.

[Time 0:20:00]

```
slide:

"No man can cross the same river twice."

Heraclitus
```

So I am going to explain that more, but this is a great quote. "No man can ever cross the same river twice." Because what is a river? I mean, we love this idea of objects; like there is this thing that changes. There is no river. There is water there at one point in time. And at another point in time, there is other water there. River; river is all in here [the mind].

[Time 0:20:27]

```
slide title: 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
```

So how did we make this mistake? What is the real nature of this mistake? It looked like we could change memory in place. We were doing it. There is PEEK and POKE and it looked like we could see that. We could read. But there was nothing about what we were putting in memory that had any correlation to time. It was live again.

And now we are finding, "Well, look at these new computer architectures. Where is the variable?" Well, there is one version over here from one point in time. And another one over here. And that is on its way to a place, that this over there might see at some point. It is live. Now we see the problem. There are no changing values. There are values at points in time and all you are ever going to get is the value for a point in time. And values do not change.

So the biggest key insight of Whitehead was: there is no such thing as a mutable object. We have invented them. We need to uninvent them. And Whitehead's model, which I am *grossly* oversimplifying. I do not even understand it. The book is completely daunting, but it is full of really cool insights. And what is he has built is a model that says: there is this immutable thing. Then there is a process in the universe that is going to create the next immutable thing.

And entities that we see as continuous are a superimposition we place on a bunch of values that are causally related. We see things happen over time and we say, "Oh, that is Fred!" or "Oh, that is the river outside the back of my house" or "That is a cloud". We know you can look at a cloud for enough time, and all of a sudden it is like, well, not it is three clouds, or the cloud disappeared. There is no cloud changing. You superimpose the notion of cloud on a series of related cloud values.

[Time 0:22:34]

```
slide title: 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
```

So, here are the rules. Again, I am not restating Whitehead. I am making this up now. Actual entities are immutable. When you have a new thing, it is a function – in that pure functional sense that I just talked about – of the past. So the future is a function of the past. And the notion of process is what creates the future from the past.

Identities are mental constructs. We call it a cloud. We call it a river. We call him Fred. It is an extremely useful psychological artifact. That is why we have object-oriented languages. This is useful to us. It helps us understand things. But we have to make sure we understand that objects are not things that change over time.

We superimpose object on a set of values we saw over time. That is an object. So just because we like to think of it this way, because it is important to us to understand the causality. You know, lion, lion, lion, lion, lion ... I better go. That does not mean there is a lion that is changing. There is not. And then, time then is strictly, again, a derivative of this series of events.

[Time 0:24:04]

```
slide:

"There is becoming of continuity,
```

```
but no continuity of becoming"

Alfred North Whitehead
```

So Whitehead's great quote, which is extremely confusing, but I think it is something that you could try to get right now and remember as I keep going, is that there is a becoming of continuity. There is this process in the universe that is creating successive values. And that allows us to say, "Oh, continuity. Great." It is not the other way around.

[Time 0:24:27]

```
slide title: 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
```

So now we are completely out of Whitehead terms. He has a whole bunch of his own terms. But these are the terms I want to use to talk about the rest of this problem.

The first is the notion of a "value". We need a very proper notion of a value. We tend to have a decent notion of a value when we say "42". We have a *much* weaker notion of a value when we talk about dates.

So the key characteristic of a value is that it is immutable. It could be a magnitude, it could be something like that, or any composite of those things that is also immutable is a value. These are extremely important to us.

Then we have "identity". Identity, again, is the psychological construct. We are going to see a succession of values whose causation is related. One was caused from the previous that was caused from the previous. And we are going to say Fred. Fred, again, is a label. The important thing is this identity, which is just a construct we use to collect the time series.

A "state" is not something you can change. The state is a snapshot. This entity has this value at this point in time. That is state. So, the concept of mutable state, it makes no sense. Mutable objects, they make no sense.

And finally, we have "time". Time is a completely relative thing. All time can ever tell you is: this thing happened before or after that other thing, or at the same point. It is not a measurable thing. It does not have dimension.

[Time 0:26:09]

```
slide title: 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_
```

This all sounds kind of high falutin'. Why do we care about this? We care about it because we are trying to make programs that make decisions. We have logic in our programs. You cannot have logic on top of rivers that can change. You can only have logic on top of values. So we need stable values.

And we need to collect them from other parts of our program. We need to see stable values. We need to be able to remember them. So I am using the word "perceived". I understand completely, perception is an incredibly intricate and unresolved mental phenomenon. But I like it better than just "observe" because I can observe the entire room, but perception really is kind of that division into entities. It is a little bit finer.

On the other hand, I do think we need identity. I mean, I think that the appeal of Object Orientation is valid. We care about this because it is the way we are thinking about the world all the time. If I have to change completely the way I am thinking about the world in order to write a program, my life is going to be hard. If I can somehow carry over from the way I think about the world, something to the way I write my program, it will be easier.

But, we cannot screw up time and state the way we have and have it still be easier, because it is now wrong. So it looks, "Oh, I understand those objects. I understand". But it is not right.

[Time 0:27:45]

```
[ Greatly magnified image of neurons in human brain. ]
```

So I saw this great talk at JavaOne where the people who wrote "Head First Java", which is a fantastic book, talked about... Well the guy talked about – I forget his name, I am sorry – "You should put a slide of a lion in your talk, because it will get everybody scared, and then they will be more receptive". So this is my lion.

[Audience laughter]

So let us just try to pull that theoretical mumbo jumbo down to something we can use to write programs.

[Time 0:28:18]

```
slide:

[ Photo of woman wearing surgical gloves holding a human brain,
showing it to a nearby man.  Maybe a scene from a movie? ]

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

The first thing we need to understand is: we do not make decisions about the world by direct cognition. We do not take our brains and rub it on the table. We do not rub it on Fred. There is a disconnect

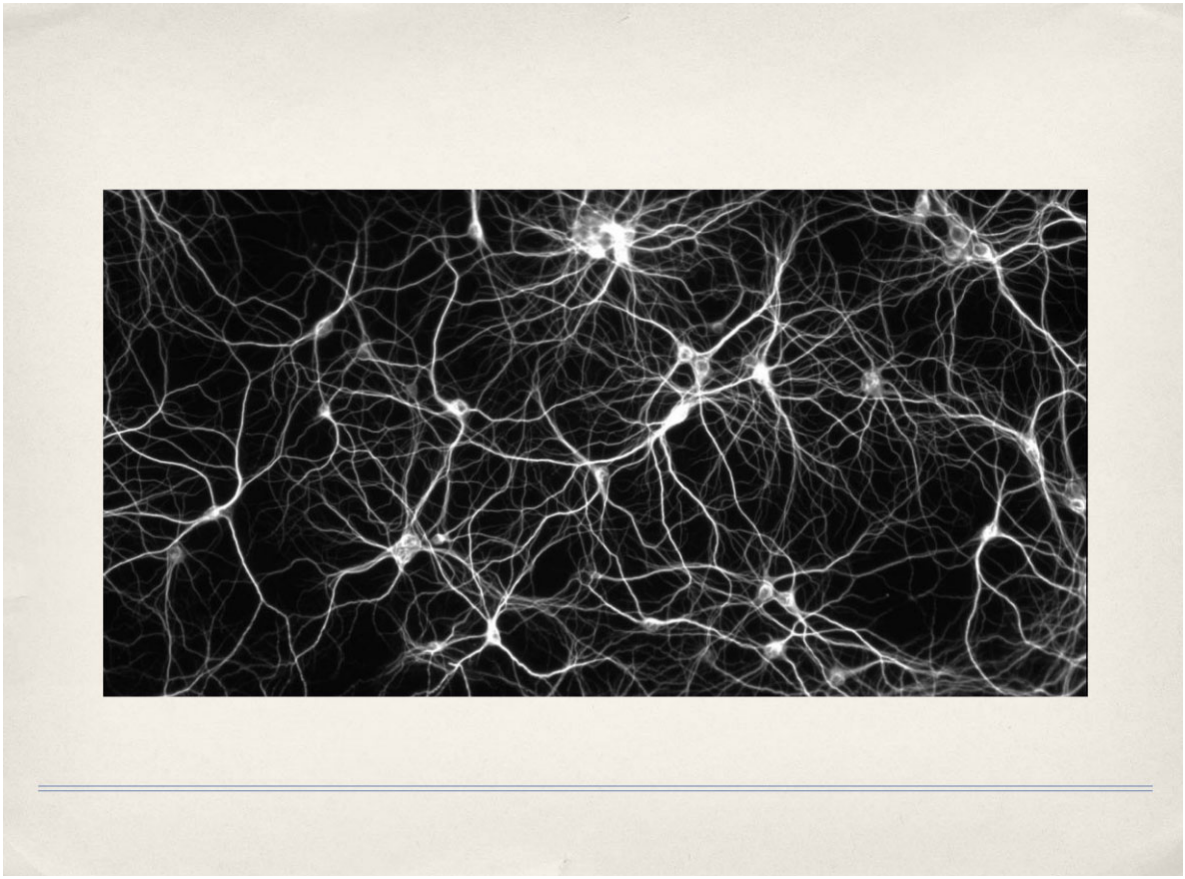

Figure 1: 00.27.45 scary graphic

between our logical system and the actual world. It is not live. This whole liveness we have from "I can see memory", that is not how it works.

[Time 0:28:46]

```
slide:

[ Image of Neo from the movie "The Matrix" holding up his hand and
causing bullets to stop in mid air before him. ]

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

The other thing we do not get to do in the real world – if we are going to model the real world – we do not get to do this [points to image of Neo stopping the bullets]. Wait!

We do not get to stop the world, especially not to observe it. But what do we do in our programs all the time? Stop! Wait! Stop! Wait! Hold on! Everybody is trying to stop the world so they can control it completely.

As we get more concurrent, we are going to need to learn to live in a world that is going to proceed in spite of our intention or desire or best wishes that it would not, because it would be a lot easier for us if it would not. It is going to. We are not going to achieve the degrees of parallelism and the concurrency we want until we can accept this and embrace it.

So we need to look more carefully. Well how does perception actually work? We do not rub our brains on it. We do not stop the world.

[Time 0:29:38]

```
slide:

[ Photo of thousands of people in a stadium watching a baseball
game. ]

Perception is massively parallel and requires no coordination

This is not message passing!
```

It is incredibly parallel. There is umpteen thousand people in this stadium. They can all watch the game. They do not say, "Whoa, whoa, whoa! Let me look at you." They do not say, "Hang on! Let me take a picture." They do not need to. They can take a picture and the game can keep going.

So the first thing is: perception is uncoordinated. It is massively parallel. It is not message passing. There is no communication between the people who want to see the game, and the game.

So, we can again look again. We are trying to model reality, so we can look at reality a little bit. How do we do it? How does the wetware do it?

[Time 0:30:19]

```
slide title: Perception

+ We are always perceiving the (unchanging!) past
+ Our sensory / neural system is oriented around:
  + Discretization
```

```
  + Simultaneity detection
+ Ignoring feedback, we like snapshots

[ Cover image of the book "Spikes: Exploring the Neural Code", by Fred
Rieke, David Warland, Rob de Ruyter van Steveninck, and William Bialek ]
```

Well it ends up that the first thing you have to realize is: we are *always* considering the past. We are never perceiving the present. There is the propagation of light. It hits my sensory system. It is an incredibly slow system that carries that to my brain. By the time I am making the decision about anything, I am using the past. I am always calculating with the past, because we are not able to impede time. We cannot stop the world. And so, the world has absolutely continued.

It seems instantaneous. I see the person in the front row here, but they could leave. Depending on how much time, and how much distance, because light is pretty fast. Again, it is like tricky. Like electrons. It makes us think that we are looking at memory *right now*. But it is really not. It is always the past. We are always perceiving the past.

The other thing to pick up from looking at our sensory system is the fact that they are incredibly oriented around discrete events. We have neurons that carry chemical signals, which could be continuous. And we could have built brains that were continuous, that somehow took the world and consider it like this moving thing. And the moving thing comes into our brain and it is all moving around.

Guess what? We did not do that. Evolution did not do that. Why? Because that is a mess! You cannot do logic if everything you are trying to consider is moving around. So what do our neurons do? They build stuff up, and then they go, "Boing!" They discretize the input.

What is the next thing that we do? We say, "Whoa! 10 things happened at the same time." So we discretize things. And then, we love simultaneity. We have simultaneity detectors. That is what our brains are at a lower level.

So coarsely, we like snapshots. Snapshots are good. They help us think. They are like values.

[Time 0:32:28]

```
slide:

[ Photo of a long line of people waiting to use a portable toilet. ]

Action, in a place, must be sequential

Action and perception are different!
```

Another thing we have done in Object Orientation is methods. Methods are a way to read things and perceive things, and a way to make things happen. Well, making things happen and perceiving things are completely different! They are completely different. They should not be in the same construct. They are two different things.

Because action has this other property. No two things can affect the same thing at the same time. We have to sort of take turns. That succession of values that we are going to use to understand the world is atomic. It is an atomic succession. And while we have grouped them into threads that helps make it easy to understand, that is not actually that way. We certainly understand the fact that there can be only a certain amount of stuff in one place at one time. And when you are trying to act on that stuff, you are going to have to be there. So action has to be sequential, and action and perception are two different things!

[Time 0:33:34]

```
slide title: Epochal Time Model
```

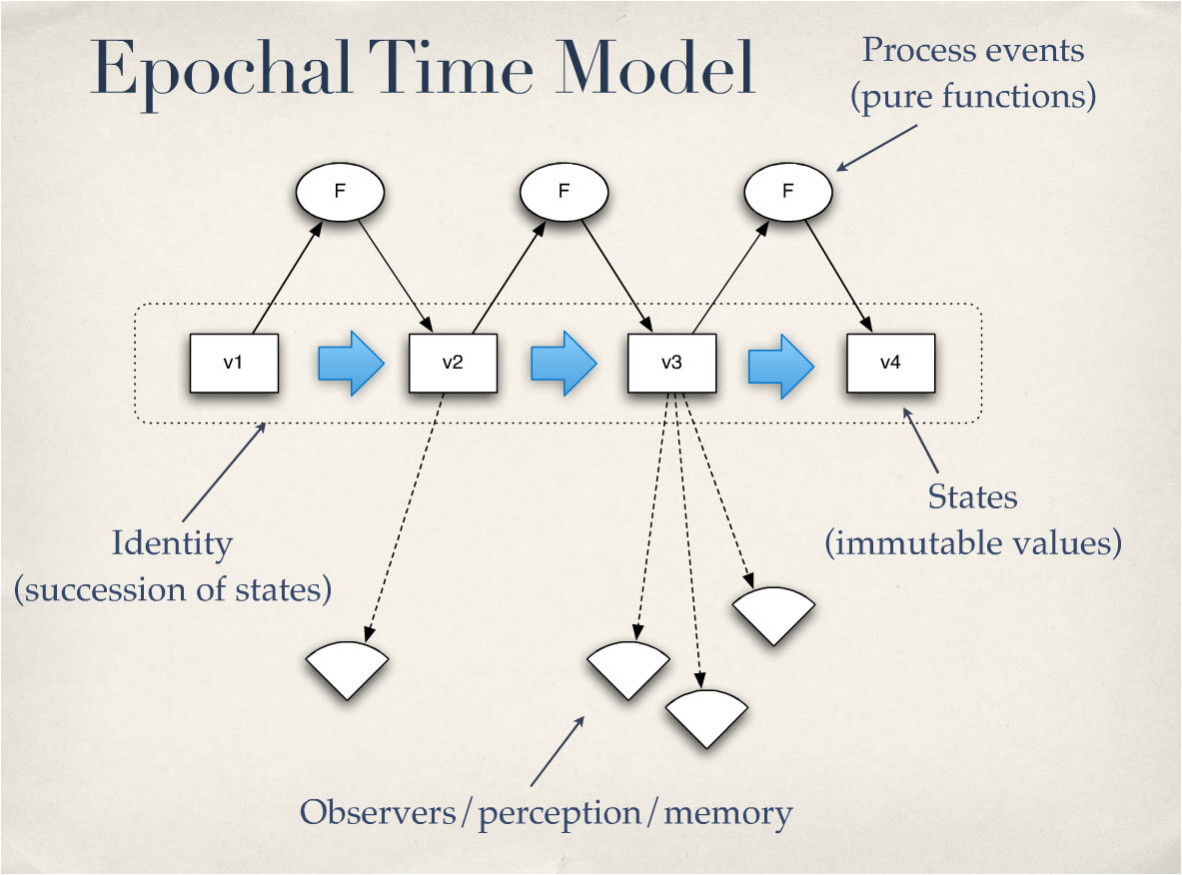

Figure 2: 00.33.34 Epochal Time Model

So now I am going to put this up. I will put it up again later. This is a model. This is not a picture of some software. This is a model for how to think about time. The first thing we need is we need a value. What is a point in time? We said a point in time is a value. It cannot be changed. So we will use values to represent points in time.

We will still probably organize our programs by identities. As long as you remember the slide from before; that the identity is a *derived* notion. It is not a thing that is doing stuff. It is a derived concept we get from this process. We can still use identities to organize things because that is going to be useful to us. Object Orientation has shown us that is useful for us to understand processes.

But, how do we get through these epochal, atomic, successive events? We use functions. We take a function of the past. We produce the future. So the Fs on the top are pure functions. They take the state of the universe, or let us just say the state of an identity, at one point in time, and produce the next one. What is inside them is indivisible, imperceptible. It is atomic. The functions are atomic.

And that is the process of the world. We say behavior in object-oriented systems. There really is a behavior that says, "Oh, I am driving." I am doing this. But when you get hit by lightning, who is behaving? There is no behavior. But there are processes in the world and they affect things. So those are those functions.

We are going to call any one of those, relative to an identity, its "state". Again, it is just a label of a

value, of an identity, at a point in time. We will call it state. And the identity itself again is a derived thing. The succession of states is Fred, or the river.

The important thing also here is that people can be looking at this. There can be observers. Light can bounce off the river. It can bounce off of Fred. Fred does not need to do that. Fred does not need to drive that. We can look at that. So we can observe things. And it is very important that observers are not in the timeline.

And then, the blue stuff in there, it is not actually reified anywhere. But that is time. Again, it is another derived thing. So the box around all the states, that identity is derived. The notion of time, it is only because at one point we looked at this, another point we looked at that, that we know that there is time. Things do not come with labels: "This was September 22nd".

All right. So how do we do this? If we wanted to take things apart like this, and then put them back together, how are we going to do it? Well we are going to need two things. We looked on the diagram before. We saw functions; pure functions. I think we know how to do that. I think we are all agreed we have the technology to write pure functions.

[Time 0:36:55]

```
slide title: 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'
```

So that leaves us with two other things on the diagram. One was values. The other was: somehow we manage that succession. Some sort of time constructs. So we need a way to efficiently create values. Save them. Maybe we will use them as percepts later. And we need something that is going to coordinate the succession of values. So we will call them "time coordination constructs". So we need those.

It ends up that we can – and maybe some theoretician will prove we *have to* – consume memory in order to model time. We certainly can. I do not know that we need to, but I have not figured out a way to do without.

So what do we do? We say we pass the old value to a pure function, we produce a new value, nondestructively. That is going to consume some memory, and we know that. But that is going to let us make this correct.

Those values have other value, because they can serve as our perceptions. The whole visual system is about making these snapshots. Admittedly, the snapshot in my mind is not the audience here. They are two different things. But in a program, they are not really two different things. If you had a value in the program, and another part of the program wanted to perceive it, they would love a copy of it. That will be great. That is a good enough record for them.

So we could use these values as our percepts. We can also use them as our memories. If we have a portion of our program that needs to remember something, this value would also serve that purpose.

So if we have a good system for doing values, we can do that. And then, the beautiful thing is: if we are consuming memory to model time, GC will erase the past and the memories that nobody cares about anymore.

[Time 0:38:59]

```
slide title: Persistent data structures

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

So the construct I think we need to do values are persistent data structures. I have talked about them before. And if anybody does not know, really quickly, we are not talking about being able to put stuff on disk here. A persistent data structure is immutable. When you make a new version of it, when you try to change it, you get a new thing. Both the old and the new thing are available after you have made it. And both have the same performance characteristics. And they make the characteristics of the data structure. And the production of the new version also has the same performance characteristics.

So that is quickie persistent data structures. So what good are they? In particular, they are immutable. So they are great for the purposes that we need: memories and perceptions. Snapshots, essentially. They are stable. Another beautiful, just practical, aspect of them is: they never need synchronization. That is just like the baseball game. That is good! There are 19,000 memories there, or 19,000 perceivers. No synchronization.

The other nice thing about persistent data structures is in their implementation. Generally, the next version of the value shares a lot of structure with the prior version. So that makes them more efficient.

The other thing that is important is when we make the new value, we do not disrupt anybody who is looking at the old value. We do not need to say, "Wait! Stop! Hang on!" Even if we are not going to destroy it, we do not need to do anything. And that goes back to the synchronization.

And if you have not ever used a functional language, or ever used persistent data structures in a nonfunctional language, just take my word for it, this is so much better. If you write a program that uses data structures like this, you will just be able to sleep at night. You are going to be happier. Your life is going to be better, because there is a *huge* quantity of things you will no longer have to worry about.

[Time 0:41:01]

```
slide:

[ Photo of a person hugging a tree that they can only get their arms a
small fraction of the way around, because its diameter is so large. ]
```

All right. So these persistent data structures, they are involved. This is old. This is really old. This stuff is so old, it is almost embarrassing to put it up here.

[Time 0:41:14]

```
slide title: Trees!

[ Figure of a tree data structure with wide branching factor. ]

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

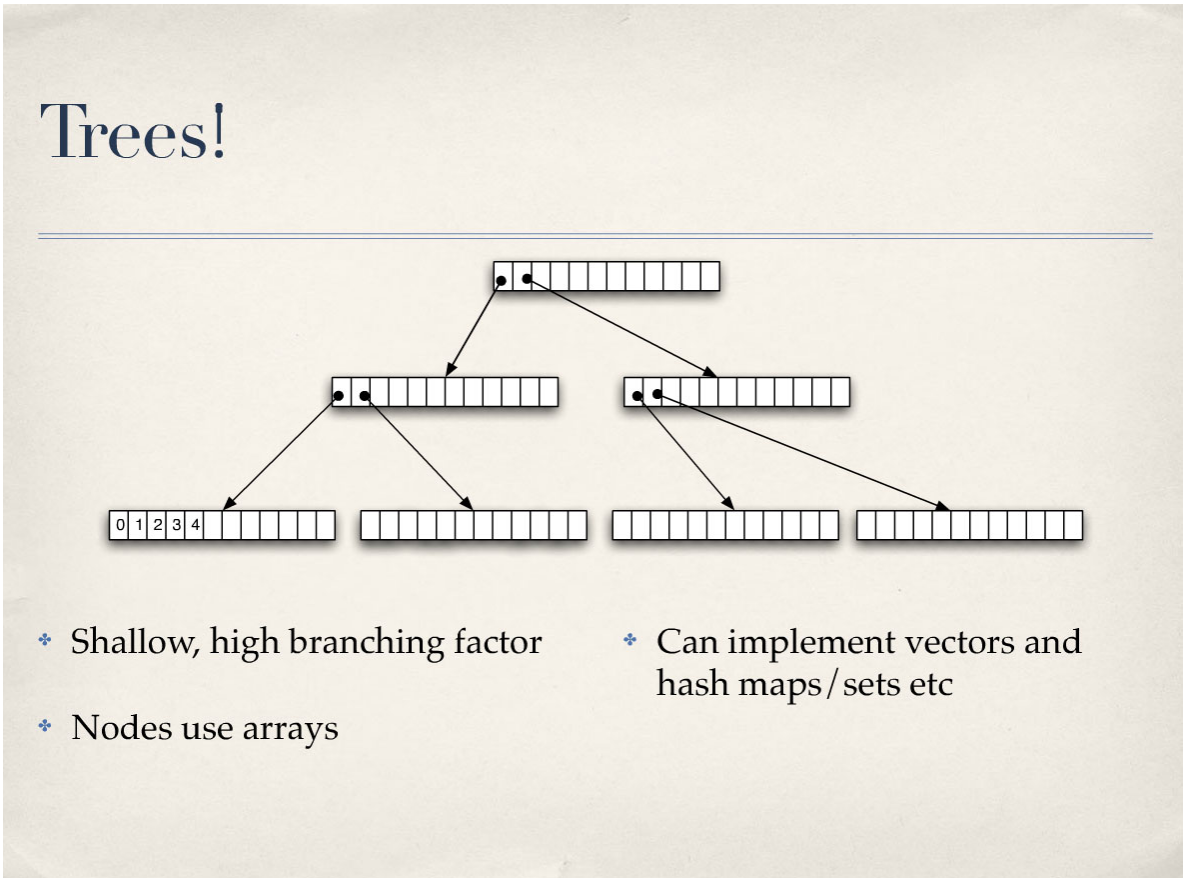

Figure 3: 00.41.14 Trees!

Trees. And all the persistent data structures essentially, under the hood, are trees, because trees have these properties that allow you to share structure and do updates. But in particular, I think from a practical sense, you can implement the kinds of things you are used to having, like vectors and hash maps and things like that, using trees with a couple of properties. At least this has been my experience.

One is that they have very high branching factors. And so therefore, they are very shallow. And that gives you good performance. You can implement vectors. You can implement hash maps. And I think the world of things you can do here is still open. But the bottom line is they are all trees, and they are trees for this reason: trees support structural sharing.

[Time 0:41:55]

```
slide title: Structural Sharing
```

So the tree rooted in "Past" there is immutable. It is never going to be changed. When we need to make a new version – say add a new node – we are going to use something called "path copying". We

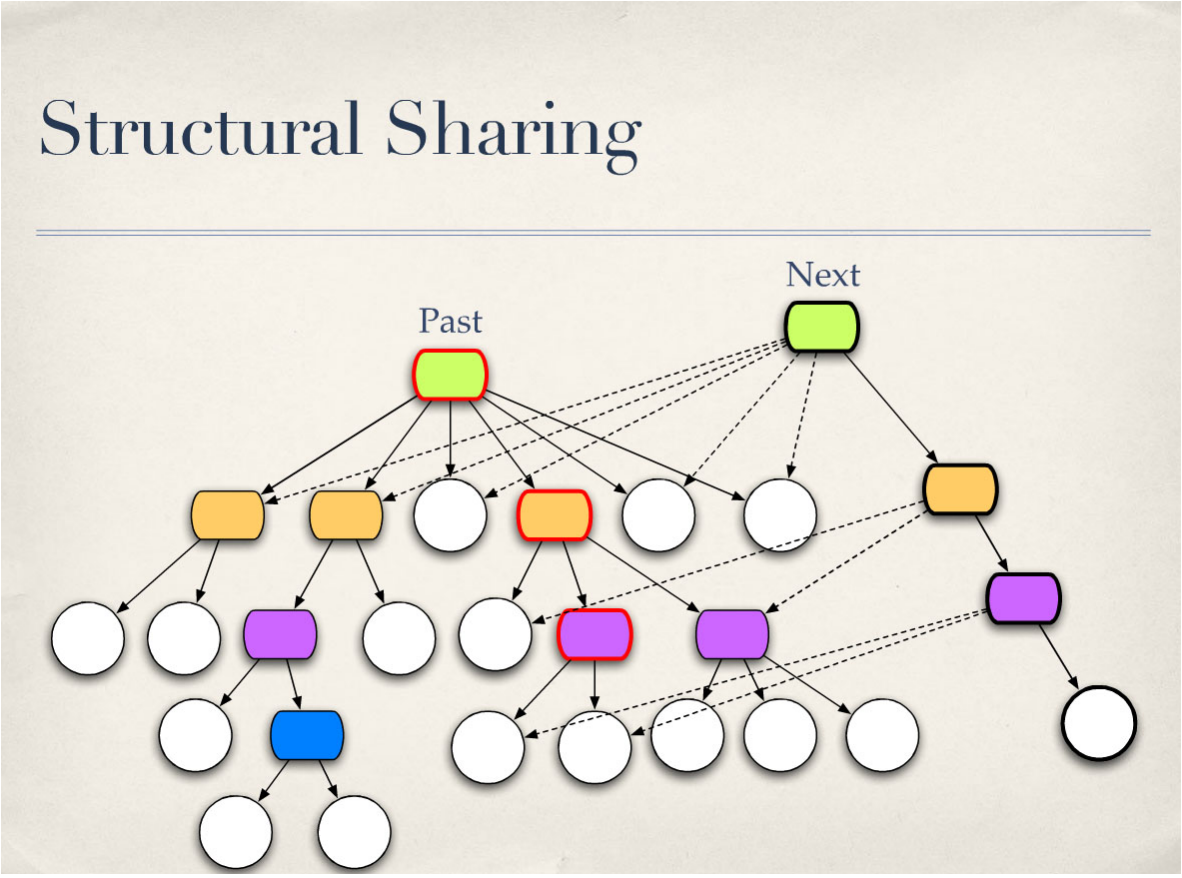

Figure 4: 00.41.55 Structural Sharing

are going to copy the path from the root to the node we need to change. So make copies of those over here on the right. This new copy will have the new node we want; the leaf node we want. And we get a new root. But that new tree rooted at "Next" shares everything with the old tree except those three red nodes. So that is good.

[Time 0:42:35]

```
slide title: 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
```

Moving forward, as we try to make programs that we can parallelize, we have to stop writing loops. I think everybody understands – it is a whole separate talk – that we are going to get our future performance gains from parallelization. Which means we are going to have to write more declarative programs. And those declarative programs are going to need to be able to take data structures and do parallel transformations on them, and produce new data structures.

And if we want to stick with this model, we want them to be persistent. So how do these persistent data structures serve that purpose? Very well, it ends up. Because they are already divide and conquer. I mean, half the work is already done. They are sitting there divided. They are pre-partitioned. In addition, if you do it right, you also have the ability to construct them in a compositional way without any collisions. So you can avoid synchronization in the building of the new versions. So they are pretty well set up for doing parallel algorithms.

I think persistent data structures should be the default data structure. I wish there was a language where persistent data structures were the default data structure.

[Time 0:43:50]

```
slide title: "It's the performance, stupid!"

the Audience

+ Persistent data structures _are_
  slower in sequential use
  (especially 'writing')
+ _But_ - no one can see what
  happens inside F

  [ Figure of function F taking
    data structure version N
    as input, producing version
    N+1 as return value. ]

+ 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
```

I mean, I am not going to lie to you. Everybody is like, performance models and everything else. They are slower! They are slower, especially for serial use. And especially for writing. For reading, you will

be very much surprised at how good the performance can be. Some of the good performance I see, I completely do not understand, but it is there. Reading is actually pretty solid. Writing though is a problem. You have that path copy and everything else.

But, I am not a fundamentalist. I am a pragmatist. So if there is this F. And if no one can ever see what happens there. In other words, if it is going to take something immutable, and it is going to produce something immutable, and those are two discrete instances of time, and everything else about this is atomic, then nobody cares what happens inside F.

You probably do care if it is a big involved thing. You probably *still* want to do it with pure functions for sanity preservation reasons. But for this time modeling reason, you can do whatever you want. Which means that when you are birthing the next version of a persistent data structure, you can do the same old good stuff you know how to do. You can allocate an array and you can bash on it, because no one has yet seen that array.

You can use Fork/Join. It works great. And these things will eventually bridge the gap. Already, on my quad-core, a parallel version of map on a persistent vector is as fast as the loop that bangs on an ArrayList. The same speed. So more cores, we start winning. Because, we have all these other great benefits. No synchronization required for this persistent data structure. Share it all you want. Rest easy. It comes with all those benefits that ArrayList does not.

The other thing that is possible is you can make what I call transient versions of these persistent data structures, and that is something I have been working on recently, which have nearly the same speed of the good old data structures you are using. And, in particular, support constant time creation from a persistent data structure, and constant time restoration as a persistent data structure, and can be made safe. They are like 90% as fast as a mutable thing.

So, obviously, this is something you should care about. On the other hand, I would not deny the power of this model because you are afraid of this.

So that is about values. Now, let us look at the time model again to remember what we are talking about when we talk about time.

[Time 0:46:40]

```
slide title: Epochal Time Model
```

So we just said we now know what the Vs are. They are going to be these persistent data structures. So how do we make sure that there is only one blue arrow train for any particular identity? How do we coordinate time? Again, remember, identity is a side effect. We see that later. The same thing with time; we see that later. But it is convenient to us, when we are trying to model in our program, to pretend we are driving it forward.

[Time 0:47:11]

```
slide title: 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
```

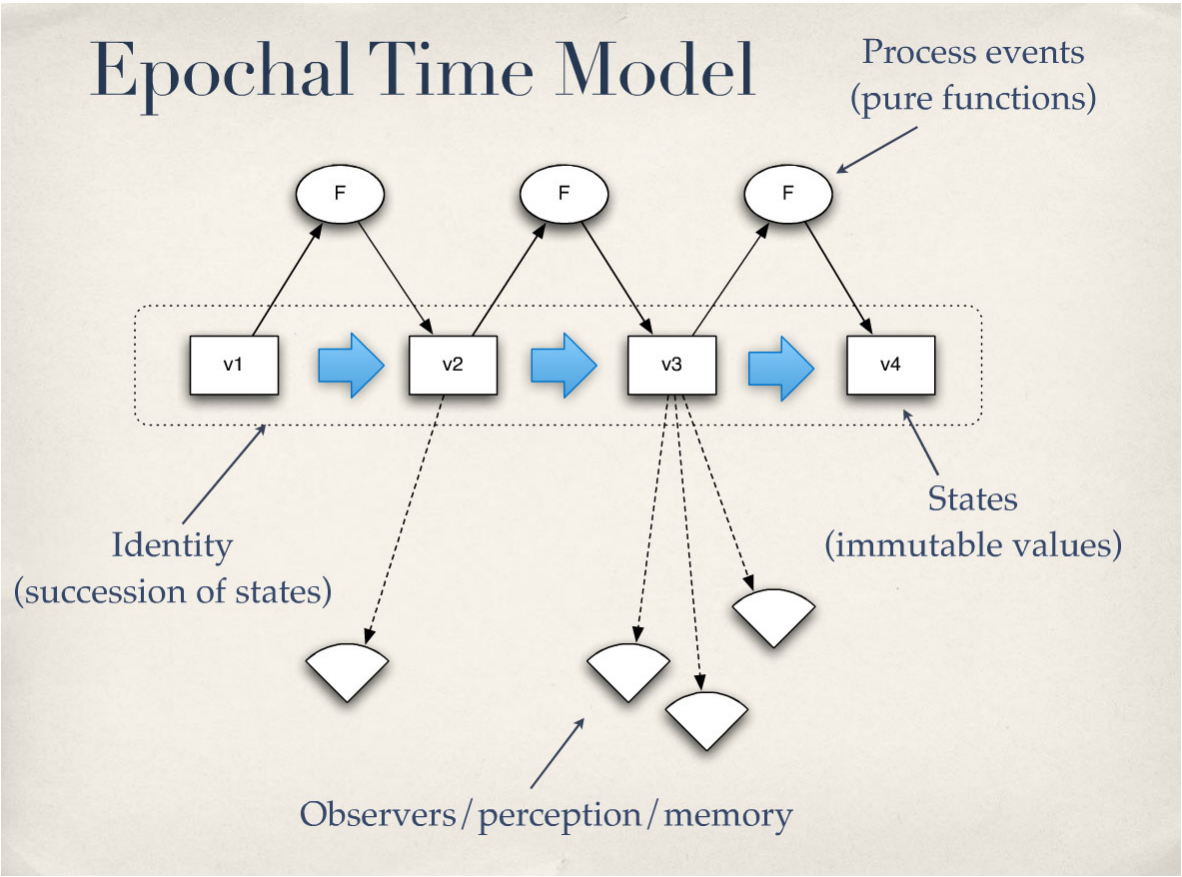

Figure 5: 00.46.40 Epochal Time Model

So what does a time construct do? Its main job is to make sure that you have atomic succession of values. That is its main purpose; that we go from one value to another, incorruptibly. And that there is no in-between. That is what epochal means.

The other thing a time construct has got to do is: it has got to provide some way for us to see the identity; to see the thing that it is managing. It has to provide visibility. And again, it has to do that atomically. Because what really happens is the baseball game. And there is photons. And then, there is a point-in-time and the photons are the same place the baseball game is. Then they go their own separate ways again. So there was a moment there where those things could connect to each other, but what was represented was that snapshot. Again, a value at a point-in-time. So they need to provide that.

We want to have multiple timelines. Again, this whole "I am the program. I control the universe. I am stopping everything or I am the only thing". That is not working any more. We need to have lots of threads of control, which means we want multiple timelines.

The nice thing about this whole thing is that there is no inherent semantics to this. Other than complying with these couple of points, there is a variety of different semantics you can apply.

[Time 0:48:31]

You can use CAS, which is essentially saying there is one timeline per identity. And it is uncoordinated. It is impossible to coordinate two things that are using CAS timelines, but CAS timelines are still useful. They have semantics you can understand.

There are agents or actor systems, which are also one-to-one. There is one timeline per entity, so they cannot be coordinated. But they are asynchronous, so that they are not connected to the timeline of the person enacting the event.

There are things like STM, which allow you to coordinate timelines. And maybe even there can be new constructs based around locks. Because you can look at locks as saying, "Well, that is the way to enforce timelines". It definitely is the way to enforce timelines. If you have a way to automate that, and package it up into one of these time constructs, that is great, and you should. The difference between them and STM would likely be that they have fixed regions, as opposed to STM, which has arbitrary regions. But if you said, "Well, all these timelines are really timeline X", X could be represented by a lock. And if you have some sort of time construct that ensures lock acquisition order, you can play this game.

[Time 0:49:48]

```
slide title: CAS as Time Construct

[ Figure ]

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

Let us look at CAS as a time construct. It is the easiest possible thing. So you have some CAS-like thingy like AtomicReference. That is going to store your timeline. There is no history in it, which means essentially that each successive value will replace the other. But what we care about is that there is a timeline. In this case, there is. It represents one identity. The thing that is in it, that is always going to be an immutable value.

CAS ensures atomic state succession. If two things, if two processes decide, "I am going to move value 2 forward. I am the process that is going to do that", only one of them can succeed. So this red line,

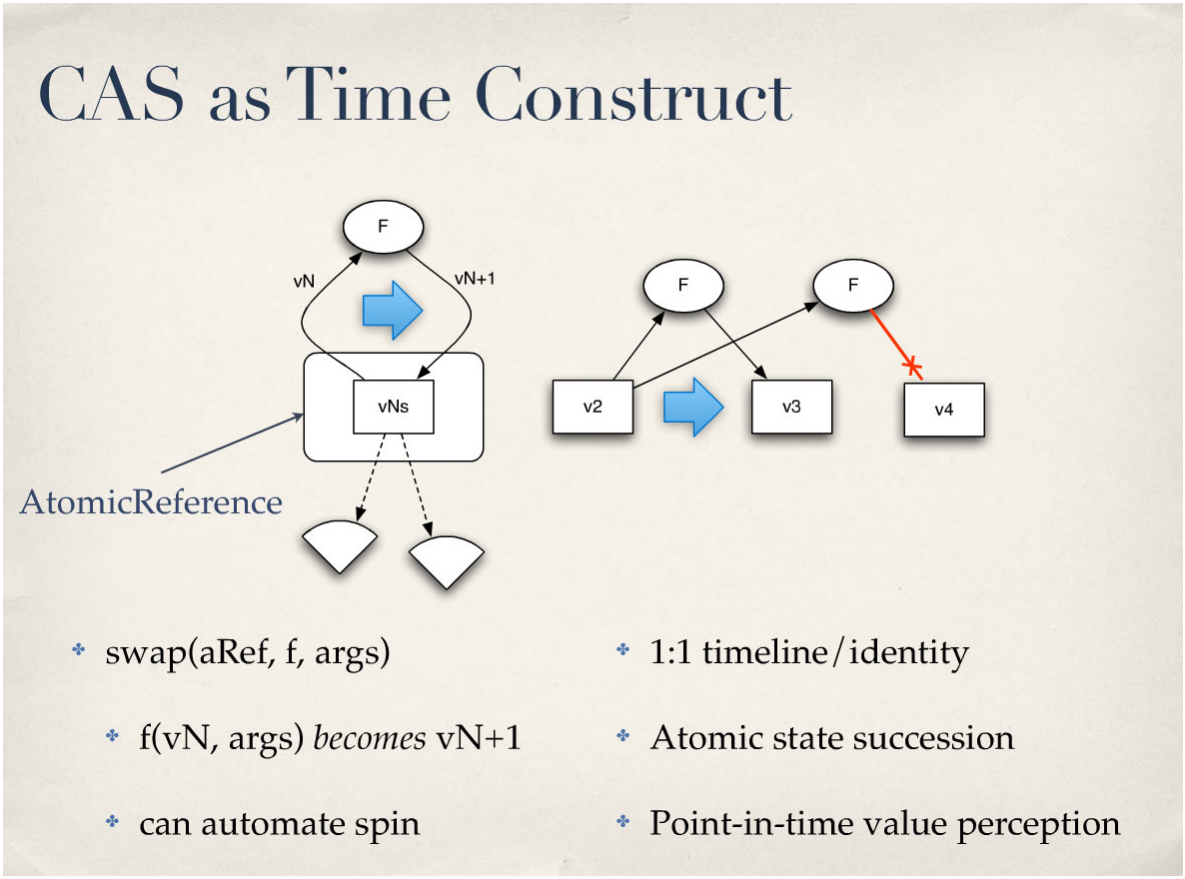

Figure 6: 00.49.48 CAS as Time Construct

that will be prevented by CAS. And you can wrap up the logic associated with doing that correctly with CAS, which is that spinning thing. And just package it in the construct.

So it could look something like this. Swap some CAS-based reference using this function, which will be the function of the past. Maybe plus some extra information; the args. And what happens in any time construct is this latter point here. You are going to call the function on the current state. Also pass the args, if you want. And that will become. That is what the construct does. It allows that to become the next value. Time is derived from that. Identity is derived from that. But that is what is really happening. So, it looks like that. And again, under the hood, we can automate the spin.

The other thing AtomicReference allows is the ability to atomically look at what is inside of it. And as long as what is inside of it is a value, we have good point-in-time perception.

[Time 0:51:30]

```
slide title: Agents as Time Construct

[ Figure ]

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

I do not want to spend too much time on agents. They are a lot like CAS, except that there is no longer a coordination. In CAS, when somebody is calling this function, when someone is saying, "swap!" there are actually two timelines. There is the timeline of the identity they are trying to manipulate, and there is the caller. They have their own timeline. Those two timelines meet at swap.

With an actor or an agent system, they do not meet. You initiate some energy force, and it flows out towards that thing, and you walk away. And eventually, that energy force hits that thing, and whatever the results is the result, and the thing changes.

So there is now an asynchrony between the caller's timeline and the timeline of the identity. But otherwise, it is still doing all the same work. It is one to one relationship between the timeline and the identity. Atomic state succession falls out of two things. The succession falls out of the fact that everything is being put in a queue. And the atomic falls out of the fact that there is only one reader.

And they can also provide point-in-time value perception. The reason why I call these things agents and not actors, is actors typically do not. In fact, they definitely do not. But in an in-process model, I think perception should always be supported.

[Time 0:52:53]

```
slide:

[ Photo of The Three Stooges, with Moe grabbing Larry and Curly. ]
```

All right. So what happens when you need to coordinate two things? Or more than two things? These things, these CASs and other things are not going to work. Because you cannot coordinate them.

[Time 0:53:07]

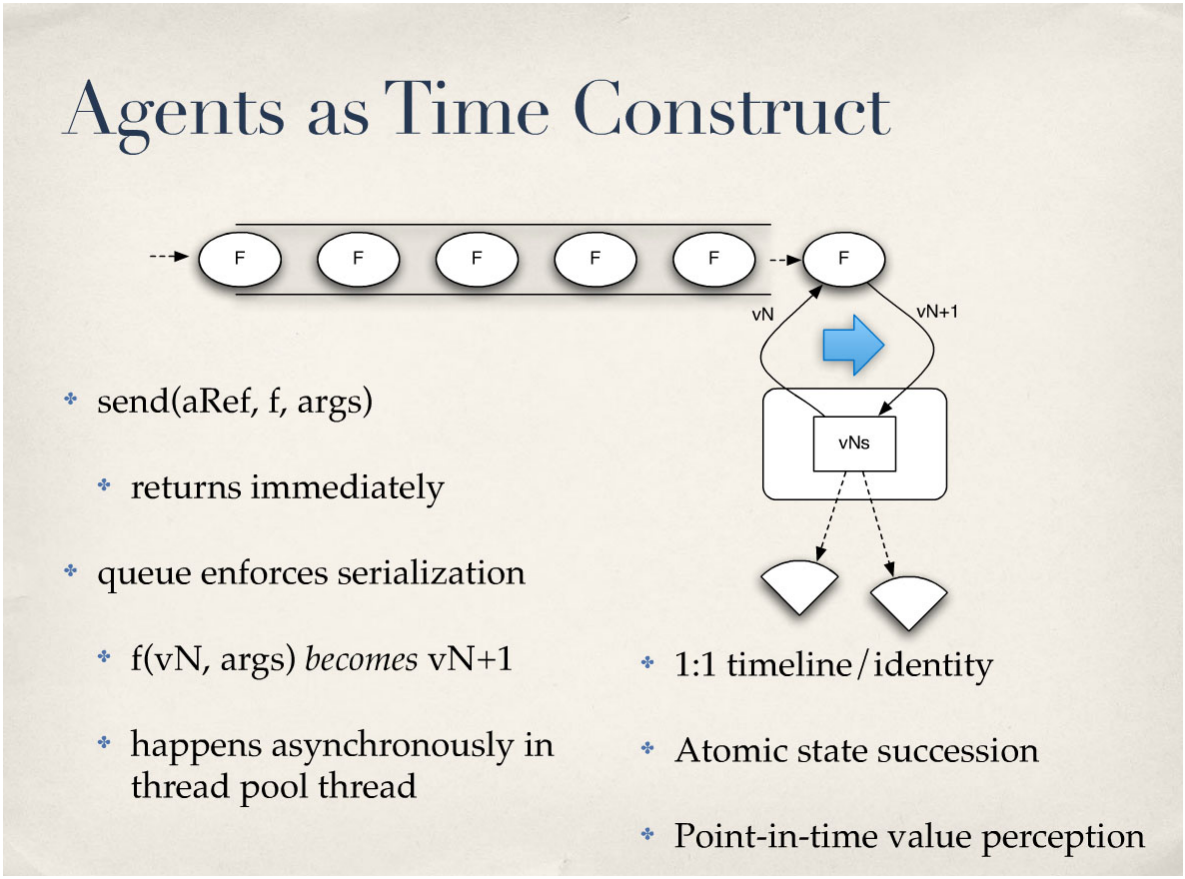

Figure 7: 00.51.30 Agents as Time Construct

```
slide title: 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
```

So you need something else. One possible other thing – it is probably not the only one – is Software Transactional Memory, or any kind of transactional thing, which allows you to coordinate the activities of multiple arbitrary regions. So multiple timelines. We are going to say, "OK, this action I am invoking is going to affect three things". Which means somehow their timelines have to meet. They have transactional capabilities, which are not really interesting for this.

But the most important thing is: we are not walking away from the epochal time model. This is *still* the epochal time model. For any value that is going to participate in an STM transaction, it is still the same thing. You are going to have some function on the past produce the future. A pure function, and values in and out.

[Time 0:53:58]

```
slide title: STM as Time Construct
```

So what does this look like now? There is multiple identities, potentially, or places, or whatever. Whatever construct is meaningful to your program, you still have that. And any particular transaction is going to take an arbitrary set of these and atomically do that function transformation. So it is a way of collecting a bunch of little micro processes and making them one process.

Internally, each one works exactly the same way as before. I just did not put all those arrows because it would be unworkable. And the set of transactions themselves feels like a timeline. In particular, if blue and yellow do not overlap, they technically happen at the same time. Really, they happen at times that – there is no time, right? Because there is no succession between those two things, there is no time. You would have to superimpose it. Because we said, time only is a derived concept from one thing happening after another. So if they are unrelated, it really gets messy about time, as physicists will tell you.

[Time 0:55:07]

```
slide title: Perception in (MVCC) STM
```

So, I left perception out of this because this was too messy. So what is the perception story for STM? Can we look at the whole stadium at one time? Can we glance and see multiple entities? And it ends up that you can build systems that do that. In particular, an STM that uses multi-version concurrency control can do it. And I will explain that more later, but I just want to show it to your first.

Essentially, what happens is there can be perceivers. What is really important about this diagram is they are still not in the timeline. There has never been somebody perceiving who got up into this box. Perception does not interfere with process. You cannot munge those two things together.

So relative to these atomic events that are in fact more than one thing, any perception is either going to occur completely after one or completely before it, if it is transactional itself. That is what STMs provide.

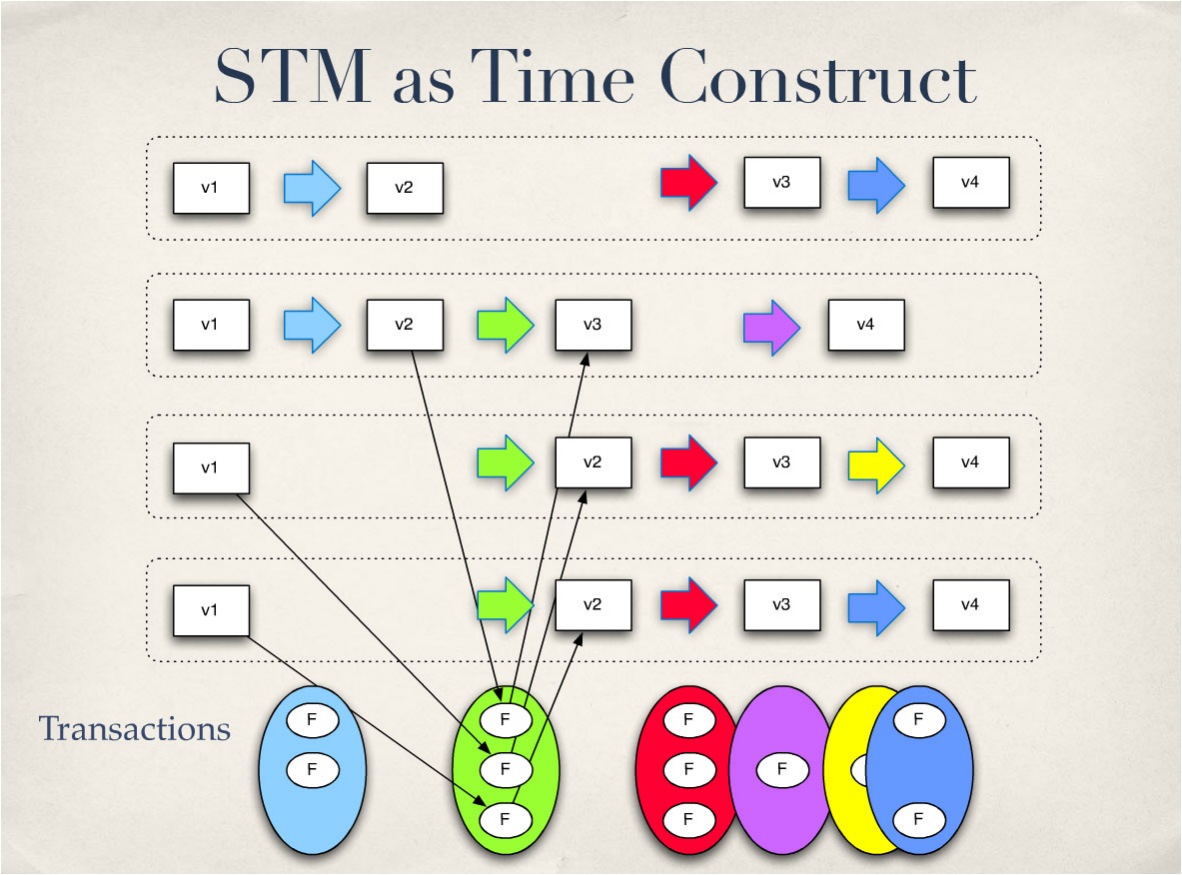

Figure 8: 00.53.58 STM as Time Construct

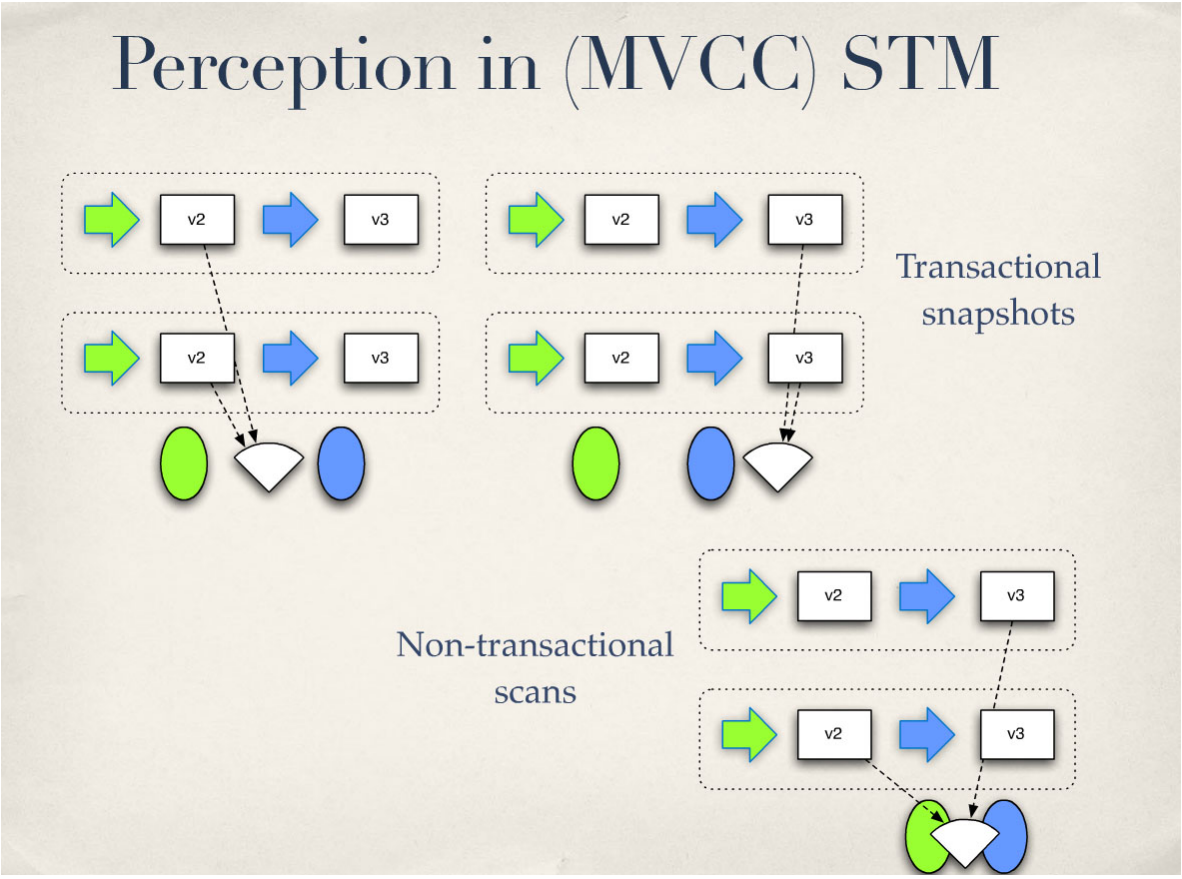

Figure 9: 00.55.07 Perception in (MVCC) STM

You can still do a non-transactional scan. You can pan. You can look at this part of the stadium, and then go over here and look at that. Or you can look at a car on the road, and you can look up at the clouds. You see the red car here, and look up at the clouds, and you look over here and you see the red car. But you know when you are doing that, what? That may be the same red car. You realize when you are panning like that, you are not seeing a point-in-time. But you have the choice.

Yes?

[Audience member: Do you think that all the procedures should agree on the order that transactions commit, or]

We will have to save that [Laughs].

[Audience member: How far do you want to take relativity, kind of? Because you would say for just normal memory operations you would probably say, "No, they do not need to agree". So yeah.]

No, they do not need to agree. They do not. But you could have multiple STMs. Because STM sort of constitutes a little universe.

So we have transactional viewing, which is like glimpsing. We have non-transactional viewing, which is like scanning.

[Time 0:57:29]

```
slide title: Multiversion concurrency control

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

So one way to do this is using multiversion concurrency control. This is the same old – all this stuff is old. This is the same old stuff from databases. So multiversion concurrency control means that you are keeping some history in order to satisfy readers. That is the database thing.

But there is a way to think about it in this model as well. That very critically, one attribute, the key attribute of multiversion concurrency control is that readers do not impede writers. That perception does not impede process. That is huge. I think you cannot do without that. Everything about everything I have shown you before, if you stick perceivers in the middle of those timelines, your life is going to get way more complicated. Stop the baseball game! So we do not want to do that.

And so, in this modeling, let us pretend we are doing the real world in our programs. You could say that multiversion concurrency control models light propagation or sensory delay. That whole chain that if somehow the light bouncing off the baseball game is capturing its value, good enough for us. That transmission delay means that that value has got to be somewhere while it is being transmitted, while the process keeps going. The game keeps going.

So we do this by keeping some history. Quite interestingly, and fortuitously, persistent data structures make keeping that history cheap. There is something profound about that that I do not understand.

The other cool thing about multiversion concurrency control is: it allows readers to have their own notion of a timeline. "I saw this then. And then later, I saw that." That is really important to some decision-making. In fact, when our brain reconstructs behavior, that is exactly what it does. We just

looked at the sensory system. It is discretizing and "snapshotizing" everything. But we definitely have percepts for the lion is running towards me. Well running, we have to re-derive that; that running. And we do that by a mental process that somehow allows us to compare a snapshot from before to a snapshot we know is later, and see the deltas of that and say, "running lion".

In addition, again, we know the difference between a visual scan. We know when we have looked at something, and we have carelessly looked at something else over here, we are not allowed to correlate those things and say they happened at the same time.

[Time 1:00:12]

```
slide title: 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
```

So this is really not a talk about STM, but I do think that one takeaway I would really like you to have is that STMs are different from each other. There is no one STM. If you want to beat up on STM, pick one, pick its attributes, and find out what is wrong with it.

Because there are some that I think really get time *wrong*, still get time wrong. So if you do not have multiversion concurrency control, you either are going to be limited to scans, non-temporally related. I looked at this thing, I looked at that thing. I have no idea. I have no ability to look at two things at once.

Or, you are going to have some gook. You are going to be back to "Wait! Wait! Stop the process so I can get my perception in the middle of it." Without multiversion concurrency control, that is where you are.

The other thing about STMs I think is super, super critical is that granularity matters. If you are using an STM that forces you – or you are incorrectly using an STM and you find yourself – requiring a transaction in order to see a consistent value, you have got time wrong again. So STMs that require a transaction to read four fields of an object consistently are not doing time right. They are not really solving this problem.

[Time 1:01:38]

```
slide title: 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
```

So in conclusion, sometimes – I mean, all the time – I think if you are suffering from excessive complexity, you have got to think about changing something. And sometimes, people actually do

change. We moved from languages that were not garbage-collected substantially, to ones that are, because that reduces our implicit complexity. There is no other reason.

But, in the current state of the art object-oriented languages, this conflation of behavior and time and identity and state is just making our lives much, much harder. And it is going to get worse. We need to become explicit about time in our programs.

We really need to pay attention to the functional programming people who are saying, "Look at all these great properties of pure functions". They are there. They definitely satisfy Whitehead's, "We move forward by taking away the need to understand the insides of things."

I believe that this epochal time model is worth trying. I think it is a general model. It supports multiple implementation ideas and it will work in the local process. I am not here talking about distributed computing at all.

The other thing I can tell you is that the current infrastructures that we have are sufficient for experimenting with this; for doing the implementation.

[Time 1:03:08]

```
slide title: 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?
```

So what is still unresolved here? Well, coordinating this internal time with the external world is going to become an important thing. And it is a hard problem. Tying STM transactions to transactional I/O would be a very interesting, and I think, possible thing.

Again, overall though, you want to move away from transactionality. Transactionality is control, control, control. You want to become as happy as you can with the lack of control. That will give you more concurrency.

There always could be more parallelism. The more performance, and there definitely could be more work done on parallelism inside these data structures. There are more data structures to be done, I am sure, and better versions.

There are definitely going to be other time constructs. I think moving locking under this model is extremely interesting, because locking has some particular efficiencies that we would want to leverage, and there is a way to understand it in terms of this.

I will leave this an open question for everybody else: is there a way to reconcile this with Object Orientation? Could we separate perception of an object from its identity enough so that we would still get the benefits of objects, but we do not get a mess later?

That is it. Thanks.

[Audience applause]

[Time 1:04:33]

```
slide:
```

```
"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
```

Do we have time for questions? I know we are running late. Any questions? Yes?

[Audience member 1: So a variation on this talk has been given at every functional programming language conference for the last 20 years, and they have been wrong for 20 years. So why do you think that this thought will be wrong, or not wrong?]

Well, I do not know what those talks were, but from what I have seen, functional programming, in many respects, just tries to get time out of the way.

[Audience member 1: Right.]

Get time out of it. Well that is not what this is. This is about: time is an important part of programs that you have to contend with.

[Audience member 1: Okay.]

I am not advocating purely functional programming here *at all*. I am saying there are programs. There are programs that are one of those boxes. One transition from value to another. That kind of program is a calculator. Most programs have to deal with that progression of time. And that is a hard problem. So I am not trying to walk away from it. I am trying to walk towards it.

[Audience member 2: I think what you are doing here is you are recapitulating the early history of philosophy because it was Heraclitus that said "everything flows", so that is your object oriented thing. And then, it was Plato that followed and said no. And Parmenides also had said, "No, everything is stable." And that is the functional community, saying there is no such thing as change. And then, it was Aristotle that synthesized the two and figured out how to get the hybrid model, the multi-paradigm model, and integrated change with invariability, with the concept of a substance. And that is what I see you dealing with more in this.]

[Time 1:06:11]

Yeah. Well, I mean, read Whitehead. I mean, he really did that. He really did exactly what you are saying. I am just trying to program without going crazy.

[Audience laughter]

But I am inspired definitely by that, by those notions. I think they are really important for fixing the model we have. Yes?

[Audience member 3: So, have you read a paper called "Time, Clocks, and the Ordering of Events in a Distributed System" by Lamport in '78?]

Yes, I have. Yes.

[Audience member 3: So, I think that that is exactly the same set of ideas.]

Yes, it is. I mean, I think there is lots of cool things. I think that is interesting. I think the whole Wave, what is happening in Wave right now, those operational transforms, are a really interesting way to think about this.

[A Google search on the terms: wave operational transforms

turns up this as an early result: https://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html]

I mean, there is a lot more to this. For instance, the composability of transformations and things like that that are very interesting. But yes, Lamport is. . .

[Audience member 3: There is one fascinating difference between Lamport's treatment of it and yours, is Lamport's is very much communication-oriented. He says time advances when one entity communicates something to another. Whereas in your framework, time advances when the outputs of one function are used as the inputs to the next function. But, it is really the same thing.]

It is. I mean, the communications part gets tricky, I think. I like the fact that this is sort of communication-free. But maybe time constructs are a different form of communication. And I do not know if they are different.

[Audience member 3: Different words for the same thing.]

Yes. They may be. They may be.

[Audience member 4: Yeah, yeah. Communication is in your F box, not in the flow of time.]

[Time 1:07:49]

Yeah. Well, it is sort of in the flow of time too. That is definitely – in the coordination aspect, that is communicating to people. You next. Now you, now you. Yes?

[Audience member 5: So, have you found that the JVM provides all the right primitives to make things like STM fast, or do you wish that there was additional support that you could take advantage of?]

I am having a good time right now.

[Audience member 5: OK.]

I do not . . .

[Audience member 5: What would you ask for out of JVM?]

The garbage collection pressure of this is going to be significant. So just keep making everything you have faster.

[Audience laughter]

[Audience member 5: I can do that.]

[Audience member 6: You were a little bit harsh I thought on the FP people. What about Functional Reactive Programming is [TBD]]

That is pretend time.

[Audience member 6: What is the difference?]

I do not want to characterize Functional Reactive Programming. I will say this. I worked in broadcast automation systems for a long time. I read that book. I saw absolutely *no* correlation between that and what I actually had to do in the real world at all.

[Time 1:08:54]

But fabricating time and turning it into an argument to functions, now you are punting again. You are pretending. Right? That is not time. That is again punting. And as soon as you connect it to the outside world, you will see that is the case.

[Audience member 7: Have you looked at any of the modern event processing languages that have streams of events and everything in it can operate on an immutable stream of events? You can store an event into tables or aggregation things, but all your data is immutable as the next flow.]

As long as all your data is immutable, I think you are on the right track. I think the key takeaway here is: there is no such thing as a mutable object. If you can really believe that, you can build better

systems. Which, again, is not to disagree. I mean, obviously, I like functional programming, right? But I do not see them talking about a lot of the problems that I think real people have.

Anything else? Thanks!

[Audience applause]

[Time 1:10:01]