Clojure - An Introduction for Lisp Programmers

- **Speaker: Rich Hickey**
- **Event: Boston Lisp meeting 2008 - Sep 29, 2008**
- **Video: Part 1: https://www.youtube.com/watch?v=cPNkH-7PRTk Part 2: https://www.youtube.com/watch?v=7mbcYxHO0nM**

```
slide title: Clojure

A Dynamic Programming Language for the JVM

   An Introduction for Lisp Programmers

                  Rich Hickey
```

Hi. I am here to talk about Clojure, which I have done several times, and never – oh, yeah, once, at the European Common Lisp workshop – for an audience of Lispers, but usually not. So it is always nice to talk to a crowd of people who already know Lisp, because then I can talk about more advanced things and more interesting things. I do not have to explain S-expressions and things like that.

But I want to make sure that that presumption is correct. Does everybody here know some dialect of Lisp? Yes? Anyone not? OK. Somewhat. I am not going to cater to you, so you are just going to have to hang on.

And amongst the people who know Lisp, how many are of the Common Lisp persuasion? And Scheme? OK, I will not make any Scheme jokes then.

[Audience laughter]

Does this work. Let us see. Oh yeah, look at that. [tries out slide control and slides advance]

[Time 0:00:54]

```
slide title: Clojure Objectives

+ A Lisp
+ Functional
  + emphasis on immutability
+ Supporting Concurrency
  + language-level coordination of state
+ Designed for the JVM
  + exposes and embraces platform
+ Open Source
```

So I have a lot to talk about. So it is nice that it is a Lisp group, because I can also presume that you are all very smart and I can talk *fast*.

So first I just want to set this up. What was I trying to do in building Clojure? Well, I wanted to have a Lisp. I fell in Lisp a long time ago and decided this is what I want to be able to do. In particular, I want Clojure to be a functional programming language. And there is a lot of reasons for that, one of which has to do with the robustness of programs, and the other of which has to do with concurrency, which we will talk a lot about.

I want it to be a language that supports concurrency. In what I do, I have not written a non-concurrent program in almost 20 years. I do broadcast automation software and things like that, and there is always more than one thread or process.

Clojure also is designed to run on the JVM. Being on the JVM is not an implementation detail. It is an

objective. Interoperating with Java is an objective, and an important part of Clojure. It is not just something under the hood. Which brings me to my next question. How many people know Java? Oh, good! OK, I will make fun of Java.

[Audience laughter]

But I hope, after this talk is done, that you have an appreciation of the difference between the JVM and Java. It is OK to hate Java and like the JVM. That is where I am at. Also, Clojure is open source because I think that is mandatory for adoption for a language.

[Time 0:02:24]

```
slide title: Why Lisp?

+ Dynamic
+ Small core
  + Clojure is a solo effort
+ Elegant syntax
+ Core advantage still code-as-data and syntactic abstraction
+ Saw opportunities to reduce parens-overload
```

So why Lisp? This is not an audience in which I have to justify why Lisp. We all like Lisp. It is dynamic. In particular, for me it was great because it has got a small core. You know the saying. We implement these seven primitives, and just macros and everything else takes you from there. Which means: I have a shot at getting it correct, unlike languages that have all that syntax, and then therefore very complicated compilers and parsers and everything else.

We all like Lisp syntax.

This is still an advantage of Lisp: code as data, and the resulting macro system and syntactic abstraction. Also I saw in Clojure an opportunity to help along the never ending battle of having people not balk at the parentheses before they see the rest of the value.

[Time 0:03:09]

```
slide title: What about Common Lisp and Scheme?

+ Why yet another Lisp?
+ Limits to change post standardization
+ Core data structures mutable, not extensible
+ No concurrency in specs
+ Good implementations already exist for JVM (ABCL, Kawa, SISC et al)
+ Standard Lisps are their own platforms
```

So one of the things I want to do, and one of the questions I expect to face in this room is: we have these two great Lisps. They are standardized to some degree. Why yet another Lisp? And I think this is an important and valid question, which I am going to answer tonight. I think any new dialect of Lisp *should* be asked this question, and should have a good answer.

So some of the reasons why I could not use Common Lisp or Scheme for what I wanted to do was: they are hard to change. I do not know how you change Common Lisp. Scheme just recently changed a little bit.

Because I want a language that is functional, I want the core data structures to be immutable. That is not something you can retrofit on either of those two Lisps.

There are no concurrency specs for those languages as they exist. That is not to say that implementations do not provide concurrency semantics for the stuff that is already in the language, or facilities for dealing with

threads, but it is not in the language. It is not something you can presume. It is not something that ports.

And there are already good Lisps on the JVM. Kawa, Armed Bear, SISC, they are all good. So if you have as your objective, "I have all this code and I want to move it on the JVM for some reason", that is a solved problem, and there is no reason for me to take it on.

The other thing has to do with languages and platforms, and I will talk more about that, but basically Lisps, like any language of their day, are kind of their own platforms, and that is also true of languages like Python and Ruby.

[Time 0:04:46]

```
slide title: Why the JVM?

+ VMs, not OSes, are the target platforms of future languages, providing:
  + Type system
    + _Dynamic_ enforcement and safety
  + Libraries
    + Huge set of facilities
  + Memory and other resource management
    + GC is platform, not language, facility
  + Bytecode + JIT compilation
```

So why the JVM? Well, it is coming around that, just like you would say, "I want this language and I want it to talk to this operating system. I will port it to these operating systems as targets", virtual machines are targets. In fact, they are the new targets. They are the new platforms. Just like you would say, "Oh, Lisp is a platform" "Unix is a platform".

For businesses, these VMs are platforms, because they abstract away platforms, or what we used to call platforms: the hardware, the instruction set, the CPUs, no one wants to deal with that. No one wants to deal with the OS level stuff any more.

So we have now got another layer of abstraction, which are these virtual machines. The two big ones are .NET and Java. And they provide a lot of facilities, more than just, "how do you access the file system?", including type systems. What is interesting, and surprising a little bit, is that the type systems in these VMs are dynamic. They have dynamic enforcement, dynamic creation of types. They are very dynamic, even though the languages that typically target them, Java and C#, are not particularly dynamic.

The big story, of course, is this huge set of libraries you get to tap into. All this code that other people wrote.

They also provide a bunch of high level services like garbage collection and memory management, resource management, meaning that garbage collection is no longer a language problem. Garbage collection is a platform problem, which is good.

And also, I will talk more in detail about this later, but they offer byte code. And in particular in the JVM, just in time compilation that is *extremely* sophisticated: run time analysis of your running program to find best paths, optimize short call paths. These things will take apart your code, make a fast path, compile a special fast path. All kinds of things that you absolutely *cannot* do with a static compiler. The compilation technology on the JVM is state of the art.

[Time 0:06:47]

```
slide title: Language as platform vs. Language + platform

+ Old way - each language defines its own runtime
  + GC, bytecode, type system, libraries etc
+ New way (JVM, .Net)
```

```
  + Common runtime independent of language
+ _Platforms are dictated by clients_
  + Huge investments in performance, scalability, security, libraries etc.
```

So old school was, well, we did not have very great tools. We were all writing our languages, or bootstrapping with C, and every language sort of solved the same set of problems, defining its own platform: its own garbage collector, its own byte code, its own type systems, and facilities for libraries.

Now we can look to these VMs to provide that stuff for us. And in particular, they can provide it in a way that is somewhat independent of language. I would not pretend that the JVM or the .NET run time, the CLR, are really independent of language, especially the CLR. They talk so much about it. But really any language can target this platform as long as it looks like C#.

But it ends up that if you said, well, if I want to have an abstract machine, I could have like a little stack machine that did almost nothing, or I could have a stack machine that had this sort of a primitive object system. Well, that is what these virtual machines have. They have the primitive object systems that are supported by C# and Java.

And a big thing for me is: Clojure is designed to let me continue to work for a living. And I know a lot of people in this room, or some people in this room, are extremely lucky and can write Common Lisp for money. Great. But most people cannot. And platforms are dictated by clients, just like they say, "I have this Sun box and your program has to run on it", they also now say, "I have this JVM infrastructure, and your software has to run on it, because my guys know how to install it, secure it, scale it, and you have to target it". If you do not, your language – it is just like your language did not support OS X. Your programs just could not run there. And they have real investments, and they are not stupid. This is a real requirement in a lot of commercial software.

[Time 0:08:38]

```
slide title: Java/JVM _is_ language + platform

+ Not the original story, but other languages for JVM always existed,
  now embraced by Sun
+ JVM has established track record and trust level
  + Now open source
+ Interop with other code always required
  + C linkage insufficient these days
  + Ability to call/consume Java is critical
+ Clojure is the language, JVM the platform
```

Again, as I said before, I would like to hopefully get you to distinguish between Java and the JVM. Unfortunately, Sun called them both Java for the longest period of time. It was like Java the ecosystem, and Java the language, which made it difficult to split them apart.

But it is really always been the case. There are hundreds of languages that target the JVM. And now that the JVM is open source, you are seeing even more adoption and more targeting, more libraries, more infrastructure.

Any language always had to have an interop layer with something. Typically that was C. It ends up that that is pretty insufficient, because lots of the great new software, lots of the great new infrastructure, is not being written in C. The old stuff was, but a lot of the new stuff is not. So some ability to call or consume Java is critical.

It is very difficult to do with a bridge. I have written two bridges for Common Lisp. You may know of Jfli or Foil. I wrote them. I tried very hard to make that work, but it is not a satisfying experience.

[Jfli: http://jfli.sourceforge.net Foil: http://foil.sourceforge.net]

So we look at Clojure as a language, and the JVM as a platform.

[Time 0:09:44]

```
slide title: Why Functional?

+ Easier to reason about
+ Easier to test
+ Essential for concurrency
+ Few dynamic functional languages
  + Most focus on static type systems
+ Functional by convention is not good enough
```

Why functional programming? I think there are a couple of benefits before you even get to concurrency. They are certainly easier to reason about, and easier to test.

Now everybody will say Lisp invented functional programming. We have been doing functional programming from the beginning. It is certainly possible to adopt a functional programming style in Lisps that already exist. And that is both true and not true. It is true in that yes, if you want to use lists only, and you promise each other never to modify them, you can take a functional approach. But otherwise, you cannot. And I will talk more in detail about why that is the case.

When you get to concurrency, however, you need to take a much more serious view of what it means to be functional, and what it means to be immutable, because it is going to be the case that we simply cannot go into this multicore environment and keep scribbling on the same memory from different threads.

There are not too many dynamic functional languages. Erlang is sort of a dynamic functional language, and Clojure is one, but most of the functional language like Haskell and ML couple this functional programming, and by that I mean, programming without side effects, with these ornate and very complex type systems, because that is where the hot research is.

But they do not need to go together, although people will dispute that. In particular, it is my opinion, having tried to get people to do it for a pretty long time, that functional by convention is not good enough, both because people mess it up, and because it is not subject to the optimization that is available when things really really are immutable. Something that is genuinely immutable, and provably immutable by the optimizer, is a different thing from something where two people have said in a conversation at lunch, "I promise not to change that."

[Time 0:11:36]

```
slide title: Why Focus on Concurrency?

+ Multi-core is here to stay
+ Multithreading a real challenge in Java et al
  + Locking is too hard to get right
+ FP / Immutability helps
  + Share freely between threads
+ But 'changing' state a reality for simulations and working models
+ Automatic / enforced language support needed
```

Why focus on concurrency? It has been a while now. We are not going to have our single-threaded programs have their performance improve by time passing and Moore's law kicking in. That is over. We have hit this plateau. It is not going up. If you have a program right now today, and it is only capable of running on one CPU, it will never ever be faster.

That was not always the case. It used to be you wrote a program and said, "Man, by the time we finish this,

it will be fast enough." And that is not the case. So you really have to get a grip on this.

But trying to do this with threads, with the conventional mechanisms, and Java has locks, and everybody has locks, and a lot of the languages, and a lot of the enhancements to Lisp that support multi-threading really have the same kind of crusty lock-based mechanisms for dealing with concurrency. They do not work. They are very very hard to use. They do not scale. And even if you happen to get it right in one moment in time, in maintenance it breaks.

Programming in a functional way helps. Of course anything that is really immutable, you do not have to worry about. Multiple threads can access it. There is no sweat. Nothing bad can happen.

But it ends up that most real programs that do real work, that people will pay for, are not functions. The things that a lot of the functional programming guys write are really functions. A compiler is sort of a big function. It takes the source code as an input, it calculates output, spits it on the disk. A theorem prover is a sort of a function.

A broadcast automation system is not a function. I work on the election system that runs on election night. I can tell you that is not a function. It is all we can do to make it function, but it is certainly not a function.

So these kinds of real programs are more like models of processes. You go into a business. They have this process. Most things have to run indefinitely. So you need something that is going to appear to change over time, that multiple parts of the program can access. And we typically would call that "state", and it is a reality.

So Clojure couples this immutable approach to programming with some mechanisms for doing safe state. And in particular, I think it is critical that the language provides both support and enforcement for that. That it is not a convention, where locks are a convention.

[Time 0:14:10]

```
slide title: Why not OO?

+ Encourages mutable State
  + Mutable stateful objects are the new spaghetti code
  + Encapsulation != concurrency semantics
+ Common Lisp's generic functions proved utility of methods outside of classes
+ Polymorphism shouldn't be based (only) on types
+ Many more...
```

Clojure is not object oriented. And there are many reasons, most of which have to do with the fact that I think we have to dig ourselves out of this rut we have gotten into with object oriented programming. We just have not done anything novel with it in decades.

But in particular, for this problem space, it is a disaster. Object oriented programming, as normally implemented, is inherently imperative. It is about creating this object, and banging on it from multiple threads. There is encapsulation, but it does not ... that is orthogonal to the problem of concurrency. So this has got to change.

Also, I think that the traditional style of object oriented programming, where you have methods inside a class, has been proven not as flexible as Common Lisp's generic functions, which show we should bring this polymorphism out of this box and apply it to a ... multiple arguments, for instance.

And I would go even further and say, "Why do we base polymorphism on types alone?" In the real world, we do not categorize everything by a fundamental at-birth type. Nobody is born a taxi driver, or a painter, or tall. But these type systems work that way. "You shall be tall, and we will make every decision about you based upon you being tall."

Whereas most systems and most reality supports some sort of polymorphism based upon value, or current state, or relationships to other things. So we need to sort of pull this stuff apart, and traditional object orientation has just put us in this rut where we cannot even see the difference between these things.

[Time 0:15:53]

```
slide title: What makes Clojure a Different Lisp, and Why?

+ More first-class data structures
+ Defined in terms of abstractions
+ Functional
+ Host-embracing
+ Thread aware
+ Not constrained by backwards-compatibility
```

So this is the question I want to answer, in addition to others tonight: Why is Clojure different?

The first thing that is different about Clojure is: it has more first class data structures. So lists, if you interpreted Lisp as LISt Processing, there is definitely a primary data structure, which is the list. And everybody who uses Lisp, especially Common Lisp, knows that that ain't enough. It is not a data structure that scales well. It is not suitable for a lot of application domains.

And of course Common Lisp has other data structures. It has vectors, and it has hash tables, but they are not first class. There is no reader support. They do not print well. They are just not as nice as lists. So Clojure enhances Lisp by making print-read representations for more data structures, and I will show you the details of that.

Another big difference with Clojure is that the common and core algorithms are defined in terms of abstractions. As we will see, a lot of the library in Lisps is based around a concrete data representation. You know, the two cell cons. That is really bad. Of course, no one is blaming McCarthy for doing that 50 years ago. It was brilliant. But today, we can do better, and we have to do better.

Clojure is different again because it embraces a host. It is symbiotic with the platform, which means it is going to get a lot of leverage of that platform, and work other people have done.

We are thread aware, and as you will see, I am not constrained by backwards compatibility. I am not going to compile your old code. That is not the objective of Clojure. And that frees me up to, for instance, reuse words which, if they are to mean what they mean in Common Lisp and Scheme forever, you will never have a Lisp with different semantics. You just cannot own the words. Sorry.

[Time 0:17:53]

```
slide title: Agenda

+ Clojure is a Lisp
+ First-class data structures
+ Abstraction
+ Immutability
+ Clojure's approach to various Lisp features
+ Concurrency
+ JVM
```

So I am going to break this down. We are sort of branching like a tree. We will talk about Clojure as a Lisp. We will look at the data structures. I will talk about abstraction and building algorithms on top of abstractions instead of concrete data structures. We will dig into how the immutability works in Clojure, and then I will look at some grab bag of Lisp features. You will see how Clojure differs from Common Lisp or

Scheme. And then we will dig down into the concurrency mechanisms in Clojure. And finally look at the JVM, both how you interoperate with it, and how is it as a host platform.

[Time 0:18:34]

```
slide title: Clojure is a Lisp

+ Dynamic
+ Code as data
+ Reader
+ Small core
+ REPL
+ Sequences
+ Syntactic abstraction
```

So Clojure meets all of the objectives of being a Lisp. It is dynamic, code as data, which is a primary characteristic of Lisps, and it is the same in Clojure. It is the same compilation model as Common Lisp. I am sort of a Common Lisp guy. I am not a Scheme guy. So I like the reader, and the properties of the reader being well defined.

It has a small core. You have a REPL. It has lifted the notion of lists to this abstraction called sequences, and I will talk a lot more about that. And it has macros, and the other goodies you expect from a Lisp.

If you use Clojure, you would not be at all confused about it being a Lisp.

[Time 0:19:17]

```
slide title: Lisp Details

+ Lexically-scoped, Lisp-1
+ CL-style macros and dynamic vars
+ Case-sensitive, Namespaces
+ Dynamically compiled to JVM bytecode
+ No tail call optimization
+ Many names are different
  + fn if def let loop recur do new .
    throw try set! quote var
```

As a Lisp, these are the details. It is a Lisp-1. It is lexically scoped. It is a Lisp-1. However, it has Common Lisp style full function macros. You can do whatever you want in there. And it has dynamic variables, which behave a lot like dynamic variables in Common Lisp, except they have threading semantics.

It is a case-sensitive language. That is really got to go if you want to interoperate with XML, and sort of anything. You have to be case sensitive.

It has namespaces instead of packages, and I might have some time to talk about that in detail. But in particular, it is important to note that Clojure's reader is side effect free, which is a big benefit.

Clojure is 100% compiled. There is no interpreter. It loads code, it compiles it right away to JVM byte code. From there, there is a lot more compilation that goes on, which I did not have to write. As I said before, there is some very sophisticated optimizing compilers inside the JITs, inside Java VMs, in particular, HotSpot.

Clojure does not have tail call optimization. Not because I am against it. I am all for it. It is something I had to trade off in being on the JVM. However, I was just at the JVM languages summit, and tail call optimization, every language designer who cam up there said "tail call optimization, tail call optimization". And they have heard it, and hopefully it will come in a future VM, hopefully soon.

A lot of the names are different. As I said before, the names cannot mean the same thing forever and ever and ever. There are only a few good names.

So these are the primitives in Clojure. fn is like lambda. if. def is define. let, loop, and recur I will talk about a little bit later. do is a block operation. new, dot are Java things. Java interop things. try and throw have to do with exceptions. There is set!, quote, and var, which I might have time to talk about, but this should be familiar in terms of their operation, if not the names.

[Time 0:21:24]

```
slide title: Atomic Data Types

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

So the atomic data types are the things you would expect. Clojure has good math with integers that do not truncate or wrap. There are doubles. There are BigDecimal literals, which are important for people who deal with money. There are ratios. Strings are in double quotes. Characters are preceded by a slash, a backslash.

There are symbols. The thing about symbols is: they are not places with values stuck on them. Symbols and Vars are different in Clojure. They are two separate things. So you can consider symbols in Clojure to be more simple names. There is no interning of the symbol with associated values.

Keywords start with a colon. The colon does not participate in the package system like it would in Common Lisp, so there is no implication there other than the typical thing you would expect with keywords, which is that they evaluate to themselves.

There are Boolean literals, true and false. They unify with Java Boolean. And there is nil, which unifies with Java null. So Java null and nil are the same thing. I have a whole slide on nil, so let us save the arguments and whatever for that. And there are regex literals, but they really are not any more atomic literals than these.

[Time 0:22:51]

```
slide title: First-class data structures

+ Lists inadequate for most programs
+ Lisps have others, but second class
  + No read/print support
  + Not structurally recursive
    + cons is non-destructive but vector-push is not
+ Lack of first-class associative data structures a disadvantage vs.
  all new dynamic languages
```

So I said earlier Clojure has more first class data structures. Everybody knows you could not write a performant program with lists. It is just going to be a pig. Yes, you can use a-lists and pretend you have an associative thing, but it is not good enough.

The other ones are second class. They do not have read/print support. In particular, though, they are not very Lispy. A Lisp hash table is not Lispy at all. You cannot recur on it. You cannot build it up incrementally. You cannot unroll the stack and just pop off values and use it like you do a list. You do not use hash tables like you do lists in Common Lisp.

And the same thing with vectors. And one of the reasons why is because they are destructive. You add something to a hash table, and you trashed it, and you cannot undo that trashing.

Which means that you do nice things with lists. You make lists of a-lists as you go down, if you are writing a little evaluator or compiler, you will have a list of a-lists, and you know you can just unroll that, and pop it right back off, because it is all nondestructive. You cannot do that same job with hash tables.

In Clojure, you can. Because Clojure's data structures are structurally recursive. You could say, "Pfff! Lists and a-lists for an environment? That is really not so great. But I would much rather have a stack of modifications to a nondestructive true performant associative data structure." And that is what Clojure gives you.

In particular, compared to the languages du jour, Python, Ruby, whatever, the lack of a first class associative data structure is really a bad blight on Lisp. People look at that, and they are like, I cannot work with this. Because it is something you really need. So Clojure has it.

[Time 0:24:49]

```
slide title: Clojure Data Structures

+ Lists
+ Vectors
+ Maps - hashed and sorted
+ Sets - hashed and sorted
+ All with read / print support
+ All structurally recursive
  + Uniform 'add' - conj
```

So what have we got? We have lists. We have lists. They are singly linked. They are exactly like what you know. I have heard rumors that Clojure does not have cons cells. Clojure does have cons cells. What Clojure does not have is a giant beautiful library that is married to cons cells. That is what it does not have. It does have cons cells.

Clojure has vectors. It has maps, both hashed and sorted. By maps, I mean associative data structures. Hash tables. A hash map is a hash table.

It has sets. Again, first class sets, both hashed and sorted. They all have read/print support. They are all structurally recursive.

In addition, they all support a uniform add operation, which is conj. Another S-J pun. But it ends up being an extremely powerful thing. You can write algorithms that manipulate lists, vectors, maps, or sets uniformly. Add things to them, look for things, map functions across them, and you could just substitute a map for a set, for a list, for a vector, and not change the rest of your code at all.

So if you have this presumption that, ooh, we have these data types. More data types means more complexity for my macros, and I have all this gook, because now I have this heterogeneity to my data structure set, that is not true. Because you have homogeneity in your algorithm set, and I will show you more about that later. And conj is one of the keys to that.

[Time 0:26:16]

```
slide title: Data Structures

+ Lists - singly linked, grow at front
  + (1 2 3 4 5), (fred ethel lucy), (list 1 2 3)
+ Vectors - indexed access, grow at end
  + [1 2 3 4 5], [fred ethel lucy]
```

```
+ Maps - key/value associations
  + {:a 1, :b 2, :c 3}, {1 "ethel" 2 "fred"}
+ Sets #{fred ethel lucy}
+ Everything Nests
```

This is what they look like. Lists look like what you are familiar with. Lists are parenthesized. They are heterogeneous. You can stick anything you want in there. They grow at the front. Every time I say grow, change, add, remove, there is quotes around it, OK, because none of these things actually change. So just like a list in any of the Lisps, adding something to the list does not change the list. It is a new list, right? That logic applies to all the data structures in Clojure.

Vectors support fast indexed access, like you would expect. They also grow at the end automatically, so you can conj onto a vector and it is going to grow at the tail.

Maps are in curly braces. This is just key, value, key, value. Commas are white space. They make people coming from other languages feel more comfortable, so they can use them and we do not care if you want to use them. You do not have to. So commas are just ignored. And again, it is key, value, key, value, and the keys and the values can be anything.

Sets are in curly braces preceded by a hash sign, and that is just going to be a set of unique things, which support fast look up and detection of the things in there. As I said before, both of these, sets and maps, have hashed versions and sorted versions with different performance characteristics and sorting characteristics.

And then, all of this stuff nests.

[Time 0:27:52]

```
slide title: Data Structure Literals

(let [avec [1 2 3 4]
      amap {:fred "ethel"}
      alist (list 2 3 4 5)]
  [(conj avec 5)
   (conj amap [:ricky "lucy"])
   (conj alist 1)

   ; the originals are intact
   avec amap alist])

[[1 2 3 4 5] {:ricky "lucy", :fred "ethel"} (1 2 3 4 5)
 [1 2 3 4] {:fred "ethel"} (2 3 4 5)]
```

This is just some code. It uses data structure literals, and you can see what is going on.

You can just put the literals in your code, and you get versions of that. Of course, as in all Lisps, lists by default get evaluated, so if you want a list you have to say "list" or quote it.

But the nice thing is we have this uniform interface. We can add to any collection some thing, and it will do the right thing for whatever kind of collection it is. So we can add key/value pairs to maps. You can add to the beginning of lists and the end of vectors. Making these changes does not really change anything. You will see we print, we get the changed versions and then the unchanged versions.

If at any point somebody has questions, just

[Audience: What does that program do?]

This program just shows you ... It does nothing. It creates a vector, a map, and a list. It adds something to each of those, and it is returning a vector of the changed versions and the original versions.

[Audience: Oh, the brackets are ...]

It is a literal. A literal vector. Right. So it is a vector of the answers and the originals. And we get a vector of the answers, changed vector, changed map, changed list, and the originals: unchanged original vector, map, and list.

[Audience: Are commas white space everywhere, or just within maps?]

Everywhere! Have at it! Stick them anywhere you want!

[Audience laughter]

Just whatever makes you feel good.

[Time 0:29:22]

[Audience: Is the square bracket the only way to make a vector?]

No, there are named functions for all of them things. There are functions for all of them. So you can map "vector" and get vectors.

[Audience: Is the bracket part of the "let" syntax?]

Yes, the bracket is part of the let syntax. You have seen some Schemes start to advocate using square brackets for lists that are not calls. And then, like PLT or whatever, they treat them as lists. That same convention of using square brackets when things are not calls or operators happens in Clojure. So these things, these are not calls, so people do not get confused when they see parens everywhere, they are like, "Is this a call? Is this data? I cannot get my head around it."

So that convention was a good one, except in Clojure, I do not want to waste square brackets on another kind of list, so I have real vectors.

[Audience: So that code is TBD]

That is a vector. When this is read, this is a list with a symbol, a vector with a symbol, a vector, symbol, map, symbol, list, vector, etc. etc. etc. This is real code as data. You read this in, it is fine.

[Audience: But it could have been ... you could not have had a hash table, could you? TBD]

This is a hash table.

[Audience: No, as the first argument of the "let".]

Oh, that is the destructuring stuff. I will show you that later.

[Audience: some question about comments]

Ignored. Ignored. It is the old ignored. Ignored to the end of the line comment. So nothing is produced by the reader for that.

It is a cool reader, though. For instance, it captures all the line numbers, and associates them with the data structures as metadata. I will talk about metadata later. OK?

[Time 0:31:20]

```
slide title: Abstractions

+ Standard Lisps define too many fundamental things in terms of concrete types
+ Limits generality
```

```
+ Limits extensibility
+ Marries implementation details
+ No excuse today
  + Many efficient abstraction mechanisms
```

So, this is a big thing I want to talk about, and I want to kind of beat up on the existing Lisps. Not because they made the choice, but because it is a choice that is getting moldy.

Which is: you have so many great algorithms defined in terms of concrete data types. That is really a mistake today. It limits how general your code can be. It limits the extensibility of your language.

How many people said "Oh, I wish this was a generic function" about some library function? So yeah, you do. Because the generic functions are the abstraction mechanism in Common Lisp, and they were not used a lot in the standard library, because of the way the history was.

Again, I am not blaming the people who wrote Lisp, Common Lisp and Scheme, for doing what they did. It is just today, starting from scratch, that would be not a good idea.

Because there are so many efficient mechanisms for implementing abstractions that are fast. That are super-optimized. It ends up on the JVM. That mechanism is interfaces and virtual function calls. So under the hood, implementing this stuff, implementing these abstractions in terms of that mechanism, means it is super fast. Because some people say, well, we cannot make that thing in Common Lisp a generic function, because we just cannot take the hit. No hit. These optimizers can make them all go away and inline virtual functions pretty much everywhere.

[Time 0:32:48]

```
slide title: Abstracting away the Cons cell - Seqs

+ Cons cell details - 2 slot data structure, car/cdr, arbitrary contents,
  mutable
+ Seq abstraction interface: first and rest
+ Vast majority of functions defined for cons cells can instead be defined
  in terms of first/rest
+ An object implementing first/rest can be created for vast majority of
  data structures using (seq x)
```

So this happened a lot in Clojure, but I want to take as an example case the cons cell. Getting rid of the cons cell. Getting it out of the library code.

So the cons cell is just brimming with details. It is so sad to see the Lisp books with pair of boxes, with arrows between the boxes. What a horrible way to start. Because there is a great abstraction that is in there, that is inside conses. We do not care that it has two slots. We do not care about CAR and CDR at all. We do not care about the contents. And in particular, we do not want to know that at all. I do not want to promise that you will give me back something that can change. That is terrible.

So if we were to lift the interaction pattern you have with these things, there is only two things, right? Get me the first thing, and get me the rest. Two functions. Two functions that have *nothing* to do with pairs, with cons cells, with boxes, or arrows.

There is an abstraction in there, which is: there is this sequence, potentially. Now what if there is no sequence? What do we have? Well, the Common Lispers will say: you have nil. And I do, too.

But nil means nothing in Clojure. It means you have nothing. It does not mean some magic name that also means the empty list. Because there is empty vectors, and there is empty all kinds of things. "nil" means you do not have something. So either you have it, a sequence, or you have nothing, nil. If you have a sequence, it is going to support these two functions.

"first" is going to do what? Return the first thing in the sequence. That is good. Not surprising. "rest" is going to do what? It is going to return another sequence. Again, no change. This is not an iterator. This is the abstraction inside cons cells. "rest" says "get me the rest", which is going to be either what? A sequence or nil. That is it. That is the abstraction that is inside there.

[Time 0:34:54]

[Audience: No improper lists?]

No.

[Audience: No dotted lists? OK]

Well at this point, I have stopped talking about lists, right?

[Audience: Right. Fair enough.]

I have nothing to do with lists. In fact, the next thing I am going to say is: there is no reason you cannot build an exemplar of this abstraction on top of vectors, right? I can make something that implements first and rest on vectors. I can implement something that implements first and rest on maps. I can implement something that provides first and rest on anything, and that is what happens.

All the data structures in Clojure support a function called "seq", which gets you something that implements this interface on that data structure. Now you have got this separation of concerns. Sequencing across a data structure, and the data structure, are separate. So we have lifted that away. That is a *big* deal.

[Audience: What does seq do if you give it a vector?]

It returns something that implements this interface on top of the vector.

[Audience: A cursor, right?]

You could consider it a cursor, if you want, as long as you do not associate that with anything like iterators in these broken languages.

[Audience laughter]

It is not a stateful thing. It is literally this. What is the difference between rest and move-next?

[Audience: That just gives you TBD]

"rest" gives you *another thing*. Right? move-next is a cursor. It is destructive on the iterator. Iterators are *bad*. Sequences and this abstraction is good.

Because it means, for instance, with an iterator you cannot look at three things, and then hand the head again to somebody else. That head, well that is toast. You moved it. You cannot even get at it any more. That is not true with these.

So when you say to a vector, "give me a seq on you", you get a new thing. It is not the vector. Of course, some things can implement first and rest themselves. What can? Actual lists! What a beautiful thing. So when you implement this thing on cons cells, they are non-allocating. Exactly beautiful. The same exact stuff you used to have. It is just that we have separated this from that. But that is still the most efficient implementation of this.

[Audience: Does seq on a list return the list?]

Yes it does.

[Time 0:37:23]

`slide title: Benefits of Abstraction`

```
+ "It is better to have 100 functions operate on one data structure than
  to have 10 functions operate on 10 data structures." - Alan J. Perlis
+ Better still - 100 functions per abstraction
+ seq implemented for all Clojure collections, all Java collections, Strings,
  regex matches, files etc.
+ first/rest can be lazily computed
```

So this is a famous quote, and it is a good idea. We have to go further now. We should be implementing our functions or algorithms on top of abstractions, not data structures. Pull out the abstraction.

Seq is implemented for everything in Clojure. In addition, I implemented seq on almost everything in Java. Java collections. Any Java Iterable. Java Strings. Java regex matches. Lines from a file. etc. etc. etc. There is a seq implementation for all of those, which means that all of the algorithms in Clojure work on everything.

[Audience: Do you have any trouble with the object overhead?]

No!

[Audience laughter]

No. Certainly. Let us just take an example. Vector.

[Audience: It is pretty huge in Java TBD]

No, it is not.

[Audience: TBD]

No. I disagree. I disagree strongly.

[Audience: TBD]

No, no, no. No. We have to stop doing this.

[Audience laughter]

We have to stop doing this. I will tell you. I have an anecdote, which I will use a couple of times, because it was extremely cool. I was just at the JVM languages summit, where there was a guy there who works for Azul systems. This is a company that makes these mega-boxes that run Java. They are dedicated to running Java. They have hundreds of cores.

I gave him this little Clojure program I wrote in an afternoon. It is less than a hundred lines. It implements an ant colony optimization of the Traveling Salesman Problem. I only had a 4 core machine. I did what I had to, and I used some of the stuff I am going to show you later, the STM and the agent system.

But he said, "Oh, I want to try it on my thing." So he tried it on his thing. And boom! He pulled up my program and it ran on 600 cores. Boom! Just like that. He had these cool tools for watching the cores with LED graphs popping up and down. It was so exciting. And that program was churning. You know how much it was churning? It was churning 20 Gigs of garbage a second.

[Audience laughter]

[Time 0:39:25]

20 gigs of garbage a second. Do you know how much of the overall CPU energy that used? 7%. Guess what? Ephemeral garbage on the JVM is cheap, and that is a design principle underlying Clojure.

Because you cannot implement seq on a vector without allocating memory. What is a seq on a vector going to be? It is going to be a thing that has an index in it. And when you ask for the "rest", what is it going to do? Make a new one of those with an index that is 1 greater than that. That is going to be an allocation.

So there are many data structures you cannot sequence over without an allocation per step. That is a beautiful thing. You have to see the way these JVMs optimize that stuff away. You cannot be concerned about ephemeral consing any more. It is a real design misfeature to prioritize that.

The other key thing which we will see in Clojure is: first and rest. They are so lightweight concepts that we do not actually have to have a sequence, do we? How much of a sequence do we need to implement first and rest?

[Audience: First TBD]

One thing, and some way to get at the other things. Almost nothing. We do not have to have a realized sequence, or a realized data structure at all, do we? So it means that we can implement first and rest lazily. That ends up being an extremely powerful part of Clojure, leveraging that.

[Audience: So you implement seq over Java collections.]

Yes.

[Audience: TBD You lose some of the immutability here, it seems, when you start doing iterating over collections.]

No. You do not lose the immutability guarantees of this sequence. The sequence as it walks across the collection caches everything it finds. So if you have made a pass, you will never get different answers.

If you have made that sequence on top of some crappy mutable Java thing, *and* you have mutated that thing from multiple threads, you are going to get the normal Java error from that, which is concurrent modification exception.

But no, these seqs, they are immutable. They are golden. You walk through. You get the same stuff. You can share it. You can share it between threads. You can point at different parts of it. No. Really.

[Time 0:41:52]

```
slide title: Clojure's Abstractions

+ Sequences, Associative Maps, Indexed Vectors, Sets
+ Functions (call-ability)
  + Maps / vectors / sets are functions
+ Many implementations
  + Extensible from Java and Clojure
+ Large library
+ Functions + implementations -> F*I
```

OK, so I have just shown you one. The sequence. We sort of broke that down.

But there is a similar abstraction behind all the associative things, the associative maps, and the indexed vectors, and the sets. When I showed you those pictures of those literals, well there is a bunch of implementations of these abstractions.

One of the neat things about having a data structure that returns a changed version of itself, is you can have one type for when it is a tiny hash, and then as you add stuff to it, one of those adds could return a whole different type that is optimized for being slightly larger. And when it gets bigger than that, one of those adds can give you yet another different type for being huge.

All that stuff happens under the hood in Clojure. So the key thing is that we are working with abstractions and not any promises of the concrete type.

[Audience: TBD]

Yes.

[Audience: TBD]

Yes, numbers are always boxed all the time, but I am going to ... except when they are not.

[Audience laughter]

Which I have the slides for, so I will talk about that.

All the data structures have corresponding abstractions. Callability itself is an abstraction. There is no reason to say "only lambdas can be called, and the Lisp-2 thing of the thing in the first position is going to be hard wired to this type of thing". Callability is an abstraction, and it is implemented by several of the data structures.

Maps are functions of their keys. Well guess what, it is because a map really is a function of a key, isn't it? That is what it means to be associative. It is a function. So maps are functions.

Vectors are functions, and sets are functions of their keys, because that is really what it should be.

There are many implementations, as I said, and the cool thing is, this is an infrastructure. I have given you the interfaces for these things. You can extend this yourself. You do not have to call me up and say, "Oh, I wish you had this kind of cool data structure. Could you please add it?" You can do that. You can do that from Java, or from Clojure. Anybody can do it.

[Time 0:43:59]

And all the algorithms you have, and all the algorithms that I have already written, and all of the algorithms all of the other people have already written, are going to work with your new data structure without change. That is the power of abstraction. You get a large library, and by building F functions and I implementations you get F times I power.

[Audience: TBD]

Yes.

[Audience: How does that affect inlining?]

How does that affect inlining?

[Audience: TBD do we really have to worry about this stuff? Is it all TBD]

Right. If you were concerned about it affecting inlining, it would be because you were working with the biggest ones, and so that optimization is going to kick in after a number of times. For instance, the JIT in Java will typically watch your function and say, "You know, you are calling this a lot. I am going to go set off a thread on the side and try to figure out how to do that fast." By the time you are doing that, you are already on the big ones, typically.

You are either already on the big ones, you are already on the big size, or you will always be on the small size. In other words, you are not doing something that is really growing it. But yes, that kind of polymorphism does impact inlining, and that is an ongoing challenge for the JIT guys. All I can say is, they are doing it better than anyone ever has so far. These compilers are amazing.

[Time 0:45:26]

```
slide title: Lazy Seqs

+ first and rest not produced until requested
+ Define your own lazy seq-producing functions using the lazy-cons macro
+ Seqs can be used like iterators or generators of other languages
+ Lazy and concrete seqs interoperate - no separate streams library
```

```
; the library function take
(defn take [n coll]
  (when (and (pos? n) (seq coll))
    (lazy-cons (first coll)
               (take (dec n) (rest coll)))))
```

So let us talk a little bit about laziness, because it is another key component of the way Clojure works. The basis of it is very simple. "first" and "rest" are not produced until requested. How much of a list do we need to have? It ends up: none at all. We just need a recipe for finding the first thing, and a recipe for finding the rest of the things. That is really all we need to say we have a sequence.

So there is an easy way to define your own lazy sequence producing functions. It is called lazy-cons. It takes two expressions. It is a macro. It takes two expressions. One will produce the first value when requested. The other will produce the rest.

Everybody is now going to raise their hands and say, "Oh, we have had lazy streams. We know about lazy streams. We have it. Whatever." But all the lazy streams libraries, do they interoperate with the cons cells? No! There is lazy this, and lazy that, and lazy that, and lazy whatever, and lazy whatever, and you cannot cons. That is not the same.

If you have first and rest, you are a seq. You can cons concrete things onto lazy things, and then more concrete things, and then you can concatenate them all. They are all the same from an interface perspective. It is not the same as a lazy streams library. Although you can understand lazy conses by understanding the way lazy streams work, and libraries like the ones for Scheme.

You can use them like iterators or generators of other languages, but they are much better because they are not mutable. And they interoperate, I already said.

So this is what take looks like. "take" is kind of more Haskell-ish lingo: take n things from some collection. If it is still a positive number and there is stuff in the collection... This is classic Lisp punning, there is a cons or there is not.

Yes.

[Audience: You are going to tell me later how to write a take that actually returns a value of an arbitrary type? Because here you are writing a take, and it returns a sequence.]

[Time 0:47:28]

"take" returns a sequence.

[Audience: It returns a seq.]

It returns anything that implements seq. seq is an abstraction, so it can return any arbitrary type that implements seq, not one particular type. seq is an interface, so that does not dictate the concrete type. That only says that whatever the concrete type is, it has to support this interface.

[Audience: Yeah, but what I get back is going to be a seq.]

An implementation of ISeq, yes.

So that is pretty cool. So we just lazy-cons, calling first on that, and then take on the rest. OK? Yes.

[Audience: So if I wanted to take the first 100 things out of a vector and have that be something I could have random access, I would want to do some TBD]

Well, no, you can take it, and then you could dump it into a vector if you wanted to. Or you can use sub-vector, which is a constant time way to get a window on another vector. If you really wanted another vector with constant time look up.

[Audience: So there is still a difference in that sense.]

Yes, the sequence library is still a library of sequences. It has the properties of sequences, which is linear access. But there are so many things you do with that: mapping, filtering, and all those great Lisp functions, now apply to everything. That is the point. It does not make sequences now the best data structure for all usages. Sometimes you are going to take that thing, and because you know you are going to need to hammer on it from a look up perspective, dump it into something where that look up is going to be fast, because it is not going to be fast on a seq. But that is very easy to do.

Any other question on this?

[Time 0:49:15]

```
slide title: Clojure is (Primarily) Functional

+ Core data structures immutable
+ Core library functions have no side effects
+ let-bound locals are immutable
+ loop / recur functional looping construct
```

So as I said before, Clojure is primarily a functional language. There was a great talk by Eric Meyer where he said we were all wrong about functional languages, and the only possible functional language is Haskell. Because it has to be lazy, and it has to have this type system, and the type system has to let you declare effects, etc. etc. etc. The rest of us sort of banded together and said, well, we still want to call our things functional, but we will put this qualifier on it all of the time.

So it is impure. It certainly is. I do not believe in disallowing dangerous things, because sometimes dangerous things are useful. The idea behind Clojure is: it gives you tools to do the right thing, and if you can do that most of the time, your program will be a better program. If you have to do something ugly somewhere, I am not going to stop you, because I know you have to do that, either for performance or whatever.

So what does that mean for Clojure? For Clojure it means the core data structures are immutable. The core library functions do not produce side effects.

let-bound locals are immutable, so no evil twiddling inside your function. Because if you could, you would have all kinds of other issues, because you could close over those twiddlable locals, etc. etc. Also, it creates this awful style where in the whole world you are doing this nice thing with functions, and applying functions, and then inside your functions you have this mess where you have loops that trash local variables and you just end up making spaghetti – small bowls of spaghetti.

[Audience laughter]

So we do not do that.

Because of the lack of tail calls, and the desire to have a sort of functional way of programming, I needed a special construct for looping, which I will show you.

[Time 0:50:59]

```
slide title: Sequences

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

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

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

```
-> (:a 1 :b 2 :c 3 :d 4 :e 5)

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

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

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

So, this is sort of what it looks like to use.

Drop 2 things from the vector, you get a sequence of that.

Infinite sequences. Why not? cycle returns an infinite sequence looping through whatever was passed, over and over and over again. As long as you do not try to print it at the REPL, because I have not implemented *print-length*, you will be fine.

[Note that `*print-length*` has been added since this talk: https://clojure.github.io/clojure/clojure.core-api.html#clojure.core/*print-length*]

That is really a beautiful thing. It will change the way you write software, to have essentially all of these functions are lazy. They all return a lazy sequence. All of the algorithms in Clojure, that can. I mean, some things cannot be lazy. But anything that can be lazy is. So if you pass some huge humungo thing, partition 3 is going to return just the window on getting the first one. It ends up that makes it really easy in Clojure to manipulate and work on data sets that do not fit in memory.

Also, you have been paying a huge price by building your algorithms in terms of functions that return fully realized lists, and then of course n-reversing them or reversing them. It is awful. From a memory contention standpoint, it is a big deal. Remember I talked about that ephemeral memory being cheap? You know what else is really good? Ephemeral memory always lives in the generation that can get tossed for almost nothing. Garbage acquisition, or memory acquisition in that generation is pointer bumping, and the collects are extremely fast.

But when you start allocating real lists, or real data structures, to accommodate interim values, interim results, pieces of your algorithm are creating these huge things. Guess what? They are going to move into the next generation, and the garbage collection cost will be much higher than if you used a lazy approach.

So this is really great. interleave is lazy, partition, these are all lazy. And there are just tons of functions in Clojure.

[Time 0:52:55]

[Audience: Is the first line a typo, because you drop a vector and get a list back?]

No, you always get a sequence back from these. These are the sequence functions. They will take anything, but they will return a sequence, because they are returning a lazy sequence. If I had to return a vector, I would have to realize it, and all the problems I just talked about would happen.

If you want a vector, it is really easy to say . . . you can just say "into" an empty vector, "drop" whatever, and now you will have it poured in. In fact, "into" works with anything. You could say "into" a set, empty set, or a set that has some stuff in it already. So it is really beautiful. You can say "into" something, "take" blah, and that will get dumped into the thing, whatever its concrete type is.

So this is really neat, and it works on strings, and it works on whatever. "range" is also a lazy producer of infinite lists, infinite sequences.

[Audience: TBD]

Excuse me?

[Audience: "into" is just a representation . . .]

"into" takes any type of collection, which is going to support conj, and then some sequence, and it will take the things from the sequence and conj them into the collection. So it will do the right thing for the type.

[Audience: And then it returns the collection?]

It returns the collection. That is right. So for instance, you could write a completely generic "map". Because people say, "Oh, I wish you had map vector". Well if you said "into" an empty vector, "map" this thing, you would get that.

So then what is different between different calls to that. Just the thing, right? Just the, I am dumping into a vector, and I started with a vector, or I am dumping into a set, and I had a set, or I am dumping into a map, and I had a map. Well, there is a function called "empty". So if you had some collection called C, you could say "into" (empty C), "map" this function on C. And whatever C is, you will get a copy of it, and that code is 100% generic. You do not care. It will work on anything. That is cool.

[Time 0:54:54]

```
slide title: Maps and Sets

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

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

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

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

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

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

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

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

So this is some of what maps and sets look like. This is def. It just creates a global thing called m, and we give it a map.

Maps are functions of their keys, so we can look something up by just calling the map itself. Also, keywords are functions. Keywords are functions of associative things. They look themselves up in the thing, because keywords are so often used as names, as keys. So it ends up supporting a really nice looking style. And if you ever wished you had a better way to do attributes, this makes that nice looking.

You can get to keys. You can get to values. You can do all kinds of stuff. The basic operation for adding to an associative thing is "assoc", which is short for associate.

Again, you cannot own this word. So it takes some associative thing, and then an arbitrary number of keys, values, keys, values. And it gives you a *new* map with those entries made to it. If we look at m again, it will

be the same thing that it was there. In fact, this function, this call here, presumes m is the same way as it was, because it did not change here.

So "merge-with" is a very cool thing. It will take one or more maps, and it will say, "Pour all the stuff from these successive maps into the baseline map". OK, well what if there is already a key there? Well, use this function to arbitrate the value combination. So if there is already a key, take the value you have at this key, and add it. You can merge with conj and build data structures this way. it is fun.

There is a whole library for set logic that works on real sets. Union, intersection, and all that stuff, and go a higher level still. what is a relation? A relation is a set of maps. And guess what? If you build relations, there is a whole library of stuff for that, like relational algebra stuff, like joins.

[Time 0:56:52]

[Audience: When you get that union there, what determined the order of the new union?]

Because these are hash sets, the order is random. Because it is hashed. If they were sorted, you would get a sorted thing. All of the literals are the hashed versions. The literal map is a hash map. The literal set is a hash set. But you can get the sorted versions with function calls.

[Audience: In that line, could not you have done the association with conj, right?]

I could have written it with conj, but then you would have to treat each guy as a pair. The values in a collection that is associative are pairs, logically. So like if you seq on m, you are going to get a set of key value little vectors.

[Audience: You said that a hash map is arbitrary in the order . . .]

In the print, right.

[Audience: If I union two things twice, do I get the same thing?]

You get whatever you started with. Are you going to get . . . you are going to get whatever the left one is.

[Audience: So if I union these guys, if I union again]

Yeah.

[Audience: Suppose I union this guy and then union them again, am I guaranteed that the results will be the same?]

You are guaranteed the results are going to be *equal*. I am going to talk about equal.

[Audience: When you say that the reader versions are hashed, you do not mean that they are implemented literally with hashing, and that they are unordered?]

I mean that the concrete thing you get is the hash map or the hash set. I have to pick one. I am reading, and I see a literal map, right? I have to produce a real data structure. The real data structure I am going to produce is going to be in the hashed family, not the sorted family.

[Audience: Right. The point is that it has unsorted semantics.]

I am not going to promise what they are, but usually hashed things do not have any sorting, right? Because maybe I will come up with a new hashed/sorted hybrid. I do not know.

[Audience: But you are not promising that it will actually really be a hash table underneath (that is small enough to just. . . ?)]

Sometimes the fastest way to implement a hash is just to do a skip list thing, right? That is proven, when it gets small enough. But it is still in the hash family in that, when that thing grows, it will become hashed.

There is ... oh, I should make a slide of that. There is this huge beautiful hierarchy of abstractions under Clojure. Somebody made a picture of it once. It is scary.

So, for instance, there is an abstraction called Sorted, and one called Associative, which are not even in the concrete tree of things. And Indexed, and Reversible, and stuff like that. The abstraction set is good.

[Time 0:59:27]

slide:

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

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

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

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

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

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

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

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

So, just to get a feeling... Again, I cannot... I do not expect you to totally be able to grok Clojure code, but just to get a feel for the weight of it, this is some Python. Peter Norvig wrote this Python code. It is a simple spell corrector. He uses a little simple Bayesian thingy.

But Python is sort of the champ in terms of syntactic concision? Because it has no grouping. It uses white space. And length

[Time 0:59:55]

slide:

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

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

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

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

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

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

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

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

And this is the Clojure version of the same thing. Roughly the same. And we all know how to ignore this.

[Audience laughter]

As do our editors. In fact, we know the superiority about being able to move these pieces around, vs. something that is white space based.

So it is kind of neat. You know, def and blah, and you see again where we are using lists for, what we would have used lists for, data, we are going to use vectors instead, and the syntax.

There is list comprehensions, or sequence comprehensions. A bunch of other cool stuff. But it lets you write code that is the same size and weight as Python.

[Time 1:00:34]

```
slide title: Persistent Data Structures

+ Immutable, + old version of the collection is still available after 'changes'
+ Collection maintains its performance guarantees for most operations
  + Therefore new versions are not full copies
+ All Clojure data structures persistent
  + Allows for Lisp-y recursive use of vectors and maps
  + Use real maps for things like environments, rather than a-lists
```

So, saying that everything is immutable, usually the first thing is, "Oh, my god! That is going to be slow! You are going to be copying those things left and right." But of course that need not be the case. We know, for instance, that when you add something to a list, we do not copy the whole list. We do not need to, because it has structural sharing. And similarly, there are structurally shared versions of these data structures.

So by persistent data structure here, we are not talking about the database persistence, right? We are talking about the concept of having something that is immutable be subject to change, having the old version and the new version both be available after the change, with the same performance characteristics as before.

So this is not a trick one. There are some persistent algorithms that produce changed versions whose performance degrades over time. As you add more new things, all the older versions start slipping out from the performance profile. To me, that is not really persistence at all. And it is not usable. In particular, the way those things are implemented is usually with some real sharing, which means that they are a disaster from a concurrency standpoint. Really, to do this correctly, you have to have immutable data structures, which Clojure does.

The new versions are not full copies. So there is structural sharing. This really is the Lisp way. This is the kind of map, you know, or hash table, or vector, you should have in Lisp, because you want to do the same kinds of programming with these other data structures that you do with lists.

[Time 1:02:04]

And so often, you have this beautiful thing you did with lists, then you try to scale it up, and ugh, lists are really not the right thing. Then what do you do? Completely change your program! Because using the other data structures is nothing like using lists. They do not have the same properties, they mutate, etc. etc. That is not true in Clojure. So you can use real maps for things like environments, which is just so beautiful, and very very fast.

Oh, I do not have the slides for that. So I will say that Clojure's data structures are the fastest persistent data structures I know about. They are different. They are based around Bagwell's hash map trees. They have a depth of log 32 N, so they are very shallow. And they are fast. They are fast enough to use this for commercial development.

Compared to the Java data structures, they are ... for insertion times, like 4 times worse. For look up times, they can be as good, or better, to twice as slow. So that is way in the ballpark considering the benefits. And when you start comparing them to having to lock those other things in concurrent scenarios, there is no contest. So this is a small cost we are going to pay, and we are just going to be able to sleep from then on.

["Ideal Hash Trees", Phil Bagwell http://lampwww.epfl.ch/papers/idealhashtrees.pdf]

[Time 1:03:20]

```
slide title: Recursive Loops

+ No mutable locals in Clojure
+ No tail recursion optimization in the JVM
+ _recur_ op does constant-space recursive looping
+ Rebinds and jumps to nearest _loop_ or function frame

  (defn zipm [keys vals]
    (loop [map {}
           [k & ks : as keys] keys
       [v & vs :as vals] vals]
      (if (and keys vals)
        (recur (assoc map k v) ks vs)
    map)))

  (zipm [:a :b :c] [1 2 3]) -> {:a 1, :b 2, :c 3}

  (apply hash-map (interleave [:a :b :c] [1 2 3]))
```

```
(into {} (map vector [:a :b :c] [1 2 3]))
```

All right. So as I said before, we have two things impacting Clojure from a looping perspective. One is, I do not want any mutable locals. The other is, the JVM will not let me do tail calls. You have seen languages... SISC Scheme does tail calls in the JVM. It does it with this whole other infrastructure. They are calling scheme is nothing like Java's. You have to pass additional things, or trampoline, or whatever.

Clojure does none of that. Clojure has pedal to the metal calling conventions that match Java's, so I do not have tail recursion, because you cannot do that unless the JVM does that.

So we have a special construct called "recur". It does constant space recursive looping. It is going to rebind and jump to the nearest "loop", which is another special op, or function frame. So you say I want to zipmap this thing, I say loop. It is exactly like let. loop is like let. You let all these things, and then if we need to keep going, we recur. recur is going to go and hit this loop, rebinding map, k, and v, to this, that, and that.

[Audience: So does the Clojure compiler analyze and make sure it is a tail call?]

"recur" can only occur in the tail position, and I will flag you on that. And it is a goto. It is fast, under the hood. But the semantics are the semantics of rebinding.

[Audience: What is going to happen to all this if the JVM actually gets tail calls?]

[Time 1:04:52]

Well, this recur happens to target this, but you can have recur's that target the function arguments themselves, if there was no enclosing loop, so those recurs could be like a call to the same named function.

But the biggest reason for true tail call is that you want to build some sort of network of recursive calls, not self calls. And basically what will happen is, you can start doing that. Right now if you do it, you are subject to stack overflow, and when they change this, you will not be.

It ends up that in Clojure, because of the lazy sequences and this loop/recur, people complain a lot before they start using Clojure, and they complain hardly at all, after they know how to do it the Clojure way.

That is not to say I do not want tail calls. I absolutely do, and I am looking forward to the JVM guys doing it. But that is what I have.

So that is a little zipmap, and you can create a set of keys and values, and they will get zipped together and you get a map out. These two things will do the same thing. You have apply, you can do that. You can interleave, or you can dump into that "into" that I was talking about before.

[Audience: I assume that if you wrote the code using a self invocation, you would not intentionally take it and turn it into something like this?]

I don't, and I don't on purpose, because I get a lot of people coming to Clojure from Scheme. And I really feel like I do not want to tease you. And I do not want you to be confused about when you will get a real tail call, because I think that is an unworkable position to be in. Can I do this? Will I get a tail call?

So I just say no, you will not. The only time you will is if you do this. When we have tail calls, you can do the other. And that is really sort of clean, because otherwise people would be very frustrated, and understandably. So I do not optimize self calls, even though I easily could.

[Time 1:06:37]

```
slide title: Equality

+ = is value equality
+ identical? is reference equality
  + rarely used in Clojure
```

```
+ = as per Henry Baker's _egal_
  + Immutable parts compare by value
  + Mutable references by identity
```

OK, equality. The equals sign is equality in Clojure. There is also identical?, which is reference equality. We really do not do that. It is there. I know sometimes you need to do it, but it is not something you are going to do often in Clojure.

[Audience: TBD]

Equal! Equal! Have you read Henry Baker's great paper on equality and object identity? Anybody?

[http://home.pipeline.com/~hbaker1/ObjectIdentity.html

http://portal.acm.org/citation.cfm?id=165593.165596&coll=Portal&dl=ACM&CFID=14076373&CFTOKEN=8549

Baker, H. G. 1993. "Equal rights for functional objects or, the more things change, the more they are the same" SIGPLAN OOPS Mess. 4, 4 (Oct. 1993), 2-27. DOI= http://doi.acm.org/10.1145/165593.165596]

[Audience: A while ago.]

It is a great paper. It is still great, and it is still correct. In it he says: the only things you can really compare for equality are immutable things. Because if you compare two things for equality that are mutable, and ever say true, and they are ever not the same thing, you are wrong. Or you will become wrong at some point in the future.

So he had this new operator he called "egal". He went through all this equal, and eq, and eql, and agh! I need a new name. So he came up with egal and defined these semantics for egal. Those are the semantics of Clojure.

If you are comparing two reference types, they have to be the same thing in order for them to be equal. Again, here I mean Clojure's stuff. There is still Java stuff. Java stuff has plenty of broken equals, and I cannot fix them. So if you call equal on some Java things, it is going to map to equals, and it will be broken sometimes, because they are broken.

[Time 1:08:14]

[Audience: That maps to Java dot equals object?]

Yes, object dot equals. I cannot fix that. Use Clojure's data structures. They are much better.

[Audience: For cyclic structures, do you do the thing about considering two things that are observationally equivalence, even if they have distinct ... You know what I am referring to here?]

Well, good luck trying to make a cyclic data structure out of all these immutable parts.

[Audience: So, you cannot express ...]

[Audience: TBD]

Which is not really a data structure. It is a logical sequence. I do not have any special tests for equality determination there, because I do not think ...

[Audience: I am just recently out of Scheme, so I was curious if you followed TBD]

No, I have not yet. Don't do that.

[Audience: But if you call equals on a lazy, on a lazy generator that generates an infinite sequence, then what happens?]

It is the same as trying to print it at the REPL. You are going to consume it all. That is right. I mean, it may terminate early.

[Audience laughter]

No, not abnormally, but it may terminate early because it will find the things are not equal.

[Audience: Yeah.]

If one is finite, it is finite.

[Audience: Yeah.]

[Audience: What if you do into, a cycle?]

Same thing. I mean, a cycle is infinite. What do you expect? "take" from cycle and build what you want.

You do not get... There are no other infinite data structures. There are infinite sequences. There are not infinite vectors or infinite maps.

OK, so that is how it works. I have not even talked about references, but Clojure does have some reference types. They are the things that have the concurrency semantics.

[Time 1:09:48]

slide title: nil / false / eos / '()

	Clojure	CL	Scheme	Java
nil	nil	nil / '()	-	null
true/false	true/false	-	#t / #f	true / false
Conditional	nil or false/ everything else	nil / non-nil	#f / non-#f	true / false
singleton empty list?	No	'()	'()	No
end-of-seq	nil	nil	'()	FALSE
Host null/ true/false	nil / true / false	N/A	N/A	N/A
Library uses concrete types	No	cons / vector	pair	No

All right, the big religious debate. nil, false, end of stream, or end of sequence, or end of list, and the empty list. OK. Let us just do it.

Clojure has nil. nil means nothing. You do not have anything. It is not the name of some magic thing. It means nothing. Common Lisp has nil and the empty list, and those notions are unified. It might have made sense, because you really only had one primary aggregate data structure. I do not, right? Why should nil be the empty list? Why should not it be the empty vector, or the empty whatever? There is no good reason. Scheme punts on this, they do not like that nil punning or end of sequence stuff, and Java has null, and it means nothing in Java. It could be any type.

Clojure has real true and false. Common Lisp has nil or not. Scheme has true and false, but they are broken. And Java has true and false.

Clojure's conditional logic is like Common Lisp's. I like that. I like nil punning. I like saying nothing is conditional false. But I have two cases, because I have to deal with false. I have to deal with false. false is in Java land. It is going to come. I am going to encounter it. I tried very hard. There is no way to automatically or automagically do a nil/false translation in both directions. It is not possible. If you think it is, write it up and send it to me. But I think it is not. So I have these two things, or everything else, which is like Common Lisp's. It is also like Scheme's. Of course, this seems to be inherently wrong. If you are going to do this, then do it.

[Time 1:11:39]

[Audience: what is your objection to false?]

I do not have an objection to false. I have an objection to, if you are going to say true and false, and this is logical false, but no, it is not logical false.

[Audience: I know, I meant earlier you said you were forced to use false because Java used false.]

Well, it would be nice if I did not have to do both of these. It would be nice if I could do nil and everything else. I would be happy with true and nil. I am fine with Common Lisp's use of T and nil. I would have been done. I tried to make that work.

OK, is there a special singleton, the empty list? And the singleton is the key thing. There are empty list values in Clojure. In fact, they are not nothing. Because an empty list is not nothing. It is an empty list. And it needs to be distinguishable from an empty vector, because you want to say into, into it. If you only had nil, you have no power to have more than one special empty thing. So no. Clojure, no. We know Common Lisp and Scheme have the empty list. Here again ... yes?

[Audience: Plus then you can have metadata on an empty thing.]

You can have metadata on an empty thing. You can tell the difference between an empty thing and nothing, because they are not the same. An empty bucket is not "no bucket".

[Audience: I am not getting the distinction. Common Lisp has a specific empty list. Clojure does not have an empty list?]

[Time 1:13:00]

It does not have a singleton empty list. You can have ten different ones whose identity differs.

[Audience: Oh, OK.]

And it is not represented by nil. That is also sort of the key piece here. I am not actually trying to establish the rightness or wrongness of anything except this.

[Audience laughter]

I am just showing you how they are different, and the decisions I made, which generally, as you will see through this, correspond to Common Lisp's choices.

[Audience: If all of those empty lists are immutable ...]

Yes.

[Audience: ... then why would you not want to put them all into the same representation?]

I do.

[Audience laughter - probably while he points at the '() representation]

Why would I do otherwise?

[Audience: I thought you just said that each of those empty lists had a different unique identity?]

It could. It could, but I do not want you to start doing identity stuff with that. Use equals, OK? equals is good.

End of sequence. What happens when we are using a sequence – it used to be a list – we are walking through and there is no more. Well, I say nil as Common Lisp did, because Common Lisp meant this when it said no more [probably referring to nil/'() equivalence in Common Lisp]. I mean there is no more. There is nothing. So that is the same. And of course, combined with this [probably the nil or false conditional in Clojure], it means you can do nice elegant little loop things, because that idiom of Common Lisp is good.

I have host issues. I have to deal with the host, so nil maps to null, true and false map to big T big F True and False in the Java land. And the library does not use concrete types.

Are we OK? All right. That was easy.

[Time 1:14:52]

```
slide title: Functions and Arity Overloading

+ Multiple distinct bodies in single function object
  + Supports fast 0/1/2 ... N dispatch with no conditional
+ Variable arity with &

  (defn drop-last
    "Return a lazy seq of all but the last n (default 1) items in coll"
    ([s] (drop-last 1 s))
    ([n s] (map (fn [x _] x) (seq s) (drop n s))))
```

So this is just some Clojure-y – how does Clojure do some things that other languages do – other Lisps do.

Clojure can have arity-overloaded functions. Clojure does not have optional arguments. Instead it has real arity overloading. That means that a single function object can hold inside it multiple function bodies overloaded by arity. That is really fast in Java, and it is quite nice. It means you can report all kinds of things, and you do not really have any conditionals to do this work inside. So it is kind of a neat thing. So this is an example, drop-last, there are two overloads. You can pass it a thing, or you can pass n and a thing.

It does support variable arity with ampersand, and the argument after that is bound to the rest of the things. That maps to a sequence, which means you can theoretically pass an infinite set of things as arguments to a function, as long as you do not try to print them or do any of those things you should not do.

[Audience: So if you wanted to have the Common Lisp optional arguments, you could just have something extract . . .]

Really, you can build optional arguments or keyword parameters on this, so they are not primitive, in my opinion, and they do not belong in lambda or fn. Make macros.

[Time 1:16:22]

```
slide title: Vars

+ Similar to CL's special vars
  + dynamic scope, stack discipline
+ Shared root binding established by _def_
  + root can be unbound
+ Can be _set!_ but only if first bound using _binding_ (not _let_)
```

```
  + Thread-local semantics
+ Functions stored in vars, so they too can be dynamically rebound
  + context / aspect-like idioms
```

I am going to talk later about references, in the context of the concurrency primitives, and these are the mutable things in the language, but they are actually here because of the mapping to Vars from Common Lisp.

So you can have Vars. Unlike Common Lisp, where there is this sort of symbiosis between a symbol, and the value cell, and the Var, that is separated in Clojure. There are Vars. Vars are in namespaces. Those are the things that really have the data associated with them. Symbols are lighter weight names. Vars are named by symbols.

And they have dynamic scope. They obey a stack discipline that is exactly like Common Lisp Vars, except that you can define a Var at the root with def. It could be unbound. set! is restricted to apply *only* to Vars that have been bound thread-locally. So you cannot whack the roots with set!. Otherwise, they have the semantics of special Vars.

[Audience: I do not understand. What do you mean by shared root binding? TBD]

When you have a dynamic Var, a defvar in Common Lisp, there is a root value. You can set it in Common Lisp. You can setf it. You cannot in Clojure. You could bind it locally. And the special operator for that is called binding. It is not let. So there is a special way to bind these things. When it is bound, then you can set it, and not otherwise.

It can be unbound. There is really nothing special to them. Otherwise they are extremely similar to Vars, you know defvars, except they have real thread-local semantics. It is guaranteed a binding in a thread will be distinct and invisible to any other thread. And you can use that in concurrency programming. That is a hard promise.

[Time 1:18:16]

[Audience: When you bind a new (upper function?) that you set! and you fork off into a new thread TBD]

You get a whole new set. No binding propagation to threads.

Well, there have been long arguments about this. I decided no.

[Audience: Do you effectively copy that?]

No, you get a new binding set. If you want to propagate bindings to a fn, and you are launching another thread, do it, because I am not going to do it for you.

[Audience: It takes the global . . .]

It is a new thread. It is got root bindings. Set up bindings that you want. OK?

So it is very neat. The one neat thing, though, is because Clojure is a Lisp-1, and all Clojure functions are defined with def, they are all stored in dynamic Vars, which means you can dynamically bind functions in nested contexts. That is really cool. If you have been into ContextL or aspect oriented programming, you can do that with Clojure out of the box because of this behavior. Because functions are stored in Vars, you can rebind them in context. If you want to, in a particular context, add logging to a function, you can do that. Change its behavior, swap out the algorithm, because you know the data set is going to be unusual in this call context, totally doable. It is very powerful stuff. And it sort of fell out of it. It is easy.

[Audience: ??? How much do you have to actually do at runtime to make that happen?]

The thing about the JIT optimizations is, if you are calling something 3 times, you do not care. If you are calling it enough times, in a particular context, the JIT is gonna kick in and make a custom version.

[Audience: Is it in practice that they do that?]

I doubt it. For dynamic rebindings, I doubt it. Would I put it past them? No. What was neat about the JVM languages summit, was for some of these engineers, because they have a lot of the same guys that worked for JRockit and Azul, this is the first time they are sort of hearing the needs of these dynamic languages. The kind of stuff that would matter to languages like this. They all were highly informed by the presentations. They were, "Ooh, we do not do that simple escape analysis, and if we did, all of these boxed numbers could disappear." And various other things could happen.

[Time 1:20:41]

[Audience: TBD Do you sometimes have to locally re-bind something?]

Thread locals. It is thread locals with a fast path that tests an atomic integer, which is spun up when things are bound, so if no one has rebound it, there is a fast test that does not involve thread locals, to say, pfff, cannot be thread locally bound. Fast. It is more complicated than that, but that is the short answer.

[Audience: So if I have a let form and I (??? bind the first) it will not shadow the first function because you do not use binding . . .]

No it will shadow. It is still lexical. It is still a lexical system. In other words, I have to get all the way down and say, this thing really means that Var at the root. If you have lexically shadowed it, then that shadowing happens. Otherwise, it is a dynamic system. It is not really lexical then.

[Audience: Then why are you even doing it?]

Because I do not want to muck up let. "let" is so confusing when let does dynamic binding and regular binding, in my opinion. I want people to know when they are doing this.

I am going to talk about it. There is only one let. It is let star. it is sequential let.

[Audience: In talking about dynamic variables, if the JVM ever does get a true tail call TBD]

It is not going to interact through dynamic bindings.

[Audience: TBD]

It is still in tail position, right? It has just got to test to see which one to call. It is still tail. So that still can be optimized through. There is nothing about it that is inherently non-tail. If you make a decision and use this, and that is the tail call, then that is still a tail call. It is not like you are taking a result back, and then doing something with it, and now that is not a tail call any more. That is the only place where you run into trouble. So I consider it optimizable, but it probably is not yet.

[Time 1:22:50]

```
slide title: Lisp-1 and defmacro

+ Symbols are simple values
  + Not identities / vars - no value cell
  + But still have constant-time =
+ Reader has no side effects
  + reads plain symbols
+ Symbols can have namespace part
  + my-namespace/foo
  + ` _resolves_ and _qualifies_ symbols
```

So, I say Clojure is a Lisp-1, and it has Common Lisp style defmacro. Typically, you would say, "Errrr! That is not going to work." That is why Scheme has this hygienic system, because you are going to have these clashes.

But it ends up, you have a lot of these clashes because of the way the reader and packages work in Common Lisp, not because of any inherent problem.

So if you separate out Vars from symbols, then symbols can be really simple. In fact, they can be read by the reader, and not impact the system. In other words, there is no interning happening during reading. In fact, there is interning, only of the names, so that symbols are still fast-comparable, but it is not interning of values, or mucking with your Var space. There is no value cell or anything like that. They have constant time equality. So they are usable as fast keys and the things you typically use symbols for when you are doing symbolic programming.

But the reader has no side effects. It returns this.

Symbols can have a name space part, and even a namespace-qualified symbol is still just a name. It does not say: there is storage associated with this. It does not say or imply there is a Var there, or anything else. It is just a name with a namespace. So namespaces really are a way to just qualify names. They are not about packages or anything else. You just say my namespace slash the thing, and you get a namespace-qualified symbol. It is just a name.

Then, the trick in Clojure is, there is a process called resolution, which resolves names. So now you are in a namespace. You have read in a bunch of these unadorned, or possibly namespace-qualified symbols. Now you need to say, "What does foo mean?" Now we are going to go look up the Var namespace and say, "In this namespace, foo means this Var." But we did not mess with those namespaces just by reading some source code, or anything else.

It also means that macros are manipulating this name world, not this Var world, so they can easily manipulate names, and emit stuff, into a space in which they will be qualified where the macro is defined, and the qualified version will be what the macro emits. I will show you that.

[Time 1:25:10]

```
slide title: Syntax-quote

'foo -> foo
`foo -> user/foo
(= 'foo `foo) -> false

foo -> Exception: Unable to resolve symbol: foo

(def foo 5) -> #'user/foo
foo -> 5

(= foo user/foo) -> true

(resolve 'foo) -> #'user/foo
(resolve 'list) -> #'clojure/list

(defmacro dummy [x] `(list foo ~x))

(macroexpand '(dummy foo))
-> (clojure/list user/foo foo)
```

So I call that syntax quote. So we have regular quote. quote foo is foo.

'foo -> foo

It is just a symbol. So we are imagining a fresh REPL. Nothing is defined. Backquote foo says, well, we are in the namespace user. That name resolves to user/foo.

`foo -> user/foo

There is no user/foo yet. It is just saying this is the way the name is qualified, which means when you use backquote in macros, it is going to look up those names in your namespace and emit, again, names, that resolve to what their name means in your namespace. But still just plain names.

So we can see if these two things are equal, and they are not, OK?

(= 'foo `foo) -> false

If they do not look equal, they ain't. We are not in that namespace world. If we try to use it, we cannot.

foo -> Exception: Unable to resolve symbol: foo

We did not make any Vars called foo. So let us define something called foo.

(def foo 5) -> #'user/foo

No problem. It gives us this. This [the #'] means something different in Clojure. And now we can evaluate foo.

foo -> 5

We get 5. We can say ... we can refer to foo, qualified or unqualified. We are talking about the same Var. So it is the same value.

(= foo user/foo) -> true

[Audience: Why doesn't the first foo evaluate to 5?]

This [foo and user/foo] is 5 and 5. 5 and 5 are equal, still. Even in Clojure.

[Audience laughter]

user/foo is just a fully qualified name. We are in the user namespace, so this [foo] will resolve to user/foo, because we did this [I think that refers to the (def foo 5) above].

And we did this, and we did not get an error. In fact, if there was already a foo in this namespace, this [the (def foo 5), I think] would have failed. Clojure is stricter about names existing before you use them.

So this works. We can resolve foo, right?

(resolve 'foo) -> #'user/foo

In this namespace, what does foo name? This [the #'user/foo] is a Var. I said sharp-quote is different. This is the Var, the actual Var that is in the namespace, that has the value in it. So we name them like this. And the Var is called user/foo. A Var *always* has a namespace-qualified name.

[Time 1:27:06]

[Audience: Vars are not objects?]

Everything is an object, but Vars are the places where stuff gets stored, not symbols.

[Audience: What is the current namespace mean?]

You are in a namespace. I am presuming we just started the REPL, so we are in the user namespace.

[Audience: How do we know what namespace you are in? TBD]

There is all kinds of functions for that: in-namespace, namespace, by-namespace, you can change namespaces. It is a lot like Common Lisp that way.

[Audience: It is global state? You can setq it?]

It is thread local state, because it is bound. So it is thread local, and it will unwind when you do a compilation unit. In a compilation unit, you can say I am in this namespace, and the current namespace gets pushed, it gets bound, a new binding gets pushed and popped. So it is not global. There is no trashing from other threads.

[Audience: So you do not have to define a namespace before a symbol can be qualified in that namespace?]

No. You can say anything you want. These are plain names. Until you get to resolution, it is fine, which is really the way it should be, right? Who cares? It is just this name. In fact, you may want to use it because it names something in some XML file.

They are names. Symbols as names are more useful than symbols that are bound to be Vars. The only cost, really, is that the reader cannot read pointers. The Common Lisp reader can effectively read pointers. That is a cool thing. It has a huge cost.

[Audience: Can you elaborate on that? ??? What can read pointers?]

If I read, right, I read a symbol. It gets interned. It is storage. If I read that in another file, I get the same storage. Just by reading, I have two pointers to the same storage. Which is cool, and there is some Common Lisp code that leverages that. You cannot do that in Clojure. It is a cost.

[Audience: The videographer begs to pause for about 30 seconds so he can switch tapes.]

OK

[Time 1:28:58]

[All time values below are relative to beginning of Part 2 of video.]

[Time 0:00:00]

So in practice, this combination of things. So look, we resolved foo, this was a local guy. We got the local one. Now "list", we never defined in the user namespace. In fact, all the Clojure functions are auto-imported into the user namespace, which means if we resolve list, we see it says clojure/list. It is the system one, which is neat.

Which means, if we write a macro that says backquote list foo x [referring to expression '(list foo ~x) on slide], and expand it, look what we get. Data structures with symbols, but the symbols have qualifiers. No matter what namespace I resolve this in, clojure/list resolves to clojure/list.

So this system actually solves a lot of the hygiene issues from being a Lisp-1. Separating symbols out. I do not want to spend any more time on it, but I will say I think this goes a long way towards that ... oh, I also have auto-gensyms. That is not to say you will never call gensym. You will, but you will have to be writing a more sophisticated macro, in which case you know what you are doing already.

[Time 0:01:12]

```
slide title: Pervasive Destructuring

+ Abstract structural binding
+ In _let/loop_ binding lists, _fn_ parameter lists, and any macro
  that expands into a _let_ or _fn_
+ Vector binding forms destructure _sequential_ things
  + vectors, lists, seqs, strings, arrays, and anything that supports _nth_
+ Map binding forms destructure _associative_ things
  + maps, vectors, strings and arrays (the latter three have integer keys)
```

Clojure has pervasive destructuring. Destructuring is pretty cool, I think. Pattern matching is all the rage and I am not exactly a fan. I like the destructuring part of pattern matching, but I do not like the switch statement part of pattern matching. So Clojure has destructuring, but it has it everywhere.

But like the other parts of Clojure, destructuring is based around abstractions. It is what I call "abstract structural binding". So in let and loop and fn and binding lists really anywhere where you are doing bindings, or anything that is built on these things, which is basically everywhere you are doing bindings, you can use data structures where symbols would go, and you are going to get destructuring.

So you would say let a1 or let a42. Now, instead of saying a, you can put a data structure there, and there is interpretation, two interpretations. If you have a vector as your binding form, it will destructure anything that is sequential, so that is anything that implements the Sequential interface. You can destructure lists, and strings, and arrays, and anything that supports nth, and it will pull that apart. Of course it will be a vector with names in it, and those names will be bound to the parts of the thing, and I will show you this in a second.

You can also have a map as a binding form, and that will destructure anything that is associative, which includes maps and vectors. Vectors are associative. You could do a map style binding form and get the 17th thing out of a vector, and call it fred.

[Time 0:02:48]

```
slide title: Example: Destructuring

(let [[a b c & d :as e] [1 2 3 4 5 6 7]]
  [a b c d e])
-> [1 2 3 (4 5 6 7) [1 2 3 4 5 6 7]]

(let [[[x1 y1] [x2 y2]] [[1 2] [3 4]]]
  [x1 y1 x2 y2])
-> [1 2 3 4]

(let [{a :a, b :b, c :c, :as m :or {a 2 b 3}} {:a 5 :c 6}]
  [a b c m])
-> [5 3 6 {:c 6, :a 5}]

(let [{i :i, j :j, [r s & t :as v] :ivec}
      {:j 15 :k 16 :ivec [22 23 24 25]}]
  [i j r s t v])
-> [nil 15 22 23 (24 25) [22 23 24 25]]
```

So unfortunately, I think these examples are a little bit too verbose, but this first one, we might be able to understand. So let, there is not a simple name here. The first thing, it is a vector, so we are going to bind a to the first thing b to the second thing, c to the third thing, d to the rest, and we are going to call the whole thing e. And then we are binding that to a literal vector of numbers. So a, b, c, d, e. That is really cool.

This nests, so this is a vector with two vectors, where we are naming the parts inside the vector, and then we are passing a vector of two vectors, and we are naming the parts of it, so that all does the right thing.

This one is particularly wordy, and too large, but the idea here is name, key, name, key, name, key, same :as thing, defaults, etc. etc. We bind it. We pick the pieces out. We can get the defaults in. We can name the whole structure. a, b, c, I did not have b, right? So we got the default, and m is the argument.

[Audience: What does the :or do?]

:or says if you do not get the key, use this as the value.

[Audience: You said that destructuring works for function arguments?]

Yes, you can put these things in function arguments.

[Audience: That means you can have optional arguments? You can just :or them?]

A certain flavor of them. If you are using :or, someone has to be passing you something that is Associative, which the argument list is not by default, but if they pass you a map, they could get defaults that way.

You can build all kinds of cool things with this. They nest arbitrarily, so if you do a vector destructure, or a map destructure, and then for one of the arguments, you destructure that key as a vector, you can pull all the pieces out, and if you stare at this long enough, you can figure out what it says. But the idea is that they nest. The nest arbitrarily without limit, and it is very powerful.

[Audience: there is also the keys . . .]

Yeah, it is not even on here, but there is other sugar that lets you avoid saying {a :a b :b c :c}. If you want the same names, you can just say ":keys [a b c]", and it will pull them out. Similarly you can pull out strings that way or symbols that way, so there is lots of power, user features, inside this. But the nice thing is, it is every place.

[For more, see: https://clojure.org/reference/special_forms#binding-forms https://clojure.org/guides/destructuring]

[Time 0:05:17]

```
slide title: Polymorphism via Multimethods

+ Full generalization of indirect dispatch
  + Not tied to OO or types
+ Fixed dispatch function which is an arbitrary function of the arguments
+ Open set of methods associated with different values of the dispatch function
+ Call sequence:
  + Call dispatch function on args to get dispatch value
  + Find method associated with dispatch value
    + else call default method if present, else error
```

So polymorphism. I said Clojure is not object oriented, but Clojure is not anti polymorphism. I am very pro polymorphism. I have two fundamental ways to do polymorphism.

One is the Java way. You have interfaces, all the abstractions in Clojure are defined in terms of interfaces. The nice thing about doing it that way is: it is a bridge to Java. If you want to allow extension via Java, Java can do that. Clojure can also do it.

Multimethods are the other way. That is kind of Clojure-specific, because Java does not have this kind of power. But what I wanted to do was avoid marrying dispatch to types. I am sort of tired of types. I am tired of having these rigid type systems that really incur a lot of accidental complexity for you. Every time you make a class, you have all this complexity. You do not necessarily realize any more. And fixing valuable features like polymorphism to a type system, which usually you only get one taxonomy for your arguments.

[Time 0:06:19]

So what Clojure's multimethods are is a complete generalization of dispatch. What is generic dispatch? It is . . . there are these arguments. I do not want to hard wire what happens when you call this function with these arguments. I want to do something specific to the arguments. In Java and C#, we are going to do something specific based around the type of the first argument. In Common Lisp we have more flexibility. You can use the types or the values of any of the arguments, but it is still a hard wired heuristic. Unless you are getting into CLOS underpinnings, you are still saying: it is either types or values, and there is an

order dependency, and all this other stuff. But the reality of it is, it should be an arbitrary function of the arguments that determines what your method is. And that is what it is in Clojure.

The dispatch function is an arbitrary function of the arguments. You want to have a multimethod, give me a dispatch function. I do not care what it is. That dispatch function will return different values for different argument sets. I do not care what they are. You can associate actual methods with those values. I do not care what they are either. Have at it. You want to build single dispatch, fine. You want to build CLOS style dispatch, fine. You want to do something that compares the values of arguments, go for it. You want to do something with the first and the third and ignore the middle guys, I do not really care.

So how does it work? You are going to give me a dispatch method, a dispatch function when you define the method. When I am called, I am going to call it on the arguments to get a value. Then I am going to look up and see: have you associated a method with this value? If you have, I will call it. If you have not, and you have provided a default, I will call that. If you have not defined a default, you will get an error.

[Time 0:08:05]

```
slide title: Example: Multimethods

(defmulti encounter (fn [x y] [(:Species x) (:Species y)]))

(defmethod encounter [:Bunny :Lion] [b l] :run-away)
(defmethod encounter [:Lion :Bunny] [l b] :eat)
(defmethod encounter [:Lion :Lion] [l1 l2] :fight)
(defmethod encounter [:Bunny :Bunny] [b1 b2] :mate)

(def b1 {:Species :Bunny :other :stuff})
(def b2 {:Species :Bunny :other :stuff})
(def l1 {:Species :Lion :other :stuff})
(def l2 {:Species :Lion :other :stuff})

(encounter b1 b2) -> :mate
(encounter b1 l2) -> :run-away
(encounter l1 b1) -> :eat
(encounter l1 l2) -> :fight
```

So this is what it looks like in my silly bunny example. The key thing to remember here is, so we are going to define a multimethod called "encounter", and the dispatch function is: take two arguments, and pull out their species, and return a vector of the two species. The trick you have to remember here is that keywords are functions of Associative things. So those are two function calls [referring to (:Species x) and (:Species y) expressions]. The species of the first thing and the species of the second thing.

So we are faking types. Although it is not the type of the first thing and the type of the second thing, because you only get one type. It is an attribute of the first thing and an attribute of the second thing, which means you can have other multimethods with different taxonomies. You can look at metadata, which I have not talked about. You could do whatever you want.

So in this case, we are going to pull out the species. So we encounter. If the dispatch value is bunny and lion, then we run away. if it is lion and bunny, then we eat it. Lion lion, they fight, bunny bunny, they mate.

[Audience laughter]

Then we define a bunch of literals. b1 has a species of bunny, and I do not care what else. I do not care what type it is. I do not care how many other attributes it has, or anything. I do not care. I should not care, and I should not dictate to you that it be this particular type in order to interact with my API. That has to stop.

Let people just glom stuff on so they can talk to you, and not have to *be* something so they can interact with you.

[Audience: TBD about map interface .. the species, but stored in a different way. ... When you call :Species on it.]

Absolutely. I do not care. The way keywords act as functions is: they look themselves up in the thing, using the map interface. Does it have to be a map? No. You could have species 1 and species 2, and you could pass 2 vectors. That would work. You know, whatever you need to do.

[Time 0:10:01]

So then we define a bunch of data, and then the encounter things do what you want. I have not had time to update this slide, but I have enhanced this system substantially by making it such that the values that are produced by the dispatch methods are compared via an isa? relationship, which will work for equality, and two things that are equal will be isa?.

But it also allows you to define relationships between really any names as hierarchies. It is called ad hoc hierarchies. So you do not need to have hierarchies only in types. You can just have hierarchies. You can say colon square isa? colon rectangle. You just tell me that. And then you can do this dispatch and say: if it is rectangles, do this. And if you pass squares, it will work. Not types. Why does hierarchy have to be in types? Why stick it there? It is a dungeon. So hierarchy is about ... Excuse me?

[Audience: So it is an ad hoc hierarchy per method?]

Ad hoc hierarchies. No, it has nothing to do with methods. It is just ... you can leverage them here because the multimethod system uses isa? to determine what happens.

[See: https://clojure.org/reference/multimethods]

[Audience: (Some question relating to performance of this mechanism.)]

It is a hash table look up. So all that hierarchy stuff is looked up once and cached, and then it will track any changes to the hierarchy system.

[Audience: TBD]

Well, yeah, the overhead is a hash table look up.

[Audience: There is an extra function call.]

An extra function call, and a hash table look up. Yes. It is not bad.

[Audience: So if you are using the isa? test to ??? do not you have ambiguity based on the ...]

Yes. Yes.

[Time 0:11:49]

You could. Let us just talk a little bit about that. I am sorry I do not have it on the slide. So isa? works between any two names. You define what the relationships are. Then you say: what if I have a vector of these and those? Vectors do a per-value isa? Which still begs the question: there could be ambiguity.

Yes, there could be ambiguity. For instance, you could use Java interfaces to do dispatch, and something could implement two different Java interfaces, so which one do you use? If I get an ambiguity, I will tell you when you first call it, and it will fail, but you can then tell me, for this method, prefer this to that. And that will resolve the ambiguity.

[Audience: So you actually test for applicability of every method that you defined, and then ...]

No, that only happens once. Once I get it resolved once, I cache that, so it is just a hash table look up by value. I see the same dispatch value, which is a result of the dispatch method, and it is a hash table look up. If I fail to find it, now I have to do the work. If I do the work and find an ambiguity, I will tell you. If you want to resolve that ambiguity, you can by saying: for this method, prefer this to that. Prefer this interface to that one, for instance. Or some argument set. It ends up being a pretty good compromise.

[Audience: So is this similar to the predicate dispatch thing?]

What it is missing from predicate dispatch, which I really have not been able to figure out how to do fast, is logical implication. A *real* predicate dispatch should know that being less than 10 implies being less than 100. But this is not a predicate dispatch system. But it is pretty powerful, and very general.

[Time 0:13:30]

```
slide title: Metadata

+ Orthogonal to the logical value of the data
+ Symbols and collections support a metadata map
+ Does not impact equality semantics, nor seen in operations on the value
+ Support for literal metadata in reader


  (def v [1 2 3])
  (def trusted-v (with-meta v {:source :trusted}))

  (:source ^trusted-v) -> :trusted
  (:source ^v) -> nil

  (= v trusted-v) -> true
```

Clojure supports metadata. This is the ability to attach data to any value that does not constitute part of the value's value. It is *about* the value. It is not a component of the value.

So it is orthogonal to the logical value of the data. You could have a vector, and you can say, "I got this vector from Fred", or "I found it on the Internet", or "I last read it 3 weeks ago," or anything else you want to say about it. That is not part of the value. If it is a vector of 1, 2, 3, it should still be equal to 1, 2, 3. It is just, you found it on the Internet instead of pulled it out of the database.

So it does not impact the equality semantics. You do not see it when you do any operation on the value, but it allows you to adorn values, and it is a very important thing that is extremely hard to do in most systems, which is track trustworthiness, or age, or validity, or your source, or anything else. So you have the ability to attach metadata to all the Clojure collections, and to symbols.

[Audience: But how is it that the namespace . . .]

It has nothing to do with it. It is attached to values.

[Audience: Yeah, so for the things that are in the namespace (something about not attached to the Var) . . .]

Vars have a separate notion of metadata. This is really the real deal.

[Audience: But if I wanted to have metadata attached to one of these TBD]

You can attach metadata to Vars, but this example is about real value operations. Vars are not values. Vars are references. So they have separate semantics for metadata, but they do have metadata. But I do not want to talk about that right now, because it is too confusing.

[Audience: But cannot I just do metadata in Common Lisp by saying we have an object, and then there is this hash table that maps the objects to, I do not know]

[Time 0:15:22]

Everybody has to know about that hash table. It does not flow through your call chain. Nobody can go an inspect it. They have to play. Metadata you do not have to play. And a real hash table where everybody had their metadata is a concurrency disaster.

It belongs on the object, so it should flow. It should be copiable. And if you do that, it has to be associated with identity. These things are associated with values. I can have 1, 2, 3, and then say I found it on the Internet. I could have 1, 2, 3, and say I got it out of the database. What am I going to put in my global hash table? Errrr. I am in trouble. I would have to put the identity of these things, but I do not want an identity system. I want a value system. I cannot have a hash table by value, because they are the same value. And I can talk more about it later, but it would not work.

[Audience: Is this implemented using Java annotations?]

No. No, annotations are properties of

Not really.

[Audience: TBD]

Not everything. It is attached to the class stuff, usually. Not to every instance.

[Audience: It is attached to code.]

Yes.

[Audience: TBD It avoids a lot of added overhead, TBD]

In real systems, I need this all the time. As soon as you have a big enough system, you find you need this kind of thing all the time. I have this huge calculation stack. I source it with some data. I mean, I work on this election system. So I source it with some data from counties. But the data looks exactly the same as data that comes from precincts. Now I am in the middle of a calculation. Do I have to say precinct map of this, county map of this? Putting it on types is wrong. It is just where I got it from. So I can flow that through.

[Time 0:17:14]

So essentially any of the Clojure data structures support the attachment of a map of metadata, and you can use it for whatever you want.

Clojure actually uses metadata in the reader. You can adorn your code with metadata, and you can talk to the compiler that way. In fact, that is how you give hints to the compiler. Because there is literal support for metadata.

So let us look at this. We can define v. It is the vector 1, 2, 3. Then I can say trusted-v is v, same value, with this metadata associated to it. So one thing to note about with-meta is what? It is returning a new value, because this cannot actually be the same v.

Then we can look. This caret is the same as saying meta of the thing. The metadata of trusted-v is trusted. I mean the source of the trusted-v is trusted. The source of this thing [v], we do not have. it is great.

[Audience: The caret gets you the metadata of TBD]

Caret is reader syntax for meta of blah.

[Audience: TBD]

Most of the things, like assoc or whatever, will build only on the first guys. They will not automatically incorporate. But all the operations are metadata preserving. I cannot really properly combine metadata. I do not know what it means. So I cannot do that for you. But it is really useful.

[Audience: TBD of a sequence will take the first thing TBD]

Probably. I mean, when you are adorning with metadata, you do not want to have little pieces of metadata on sub-pieces of all kinds of things. It is not that meaningful. You are usually attaching it to things whose contents are going to be consistent, or you are going to be responsible for adding or removing, and those additions and removals will produce new values with the same metadata, so they are metadata preserving. That is sort of as far as I will go, and anything beyond that you can do.

[Time 0:19:14]

```
slide title: Concurrency

+ Interleaved / simultaneous execution
+ Must avoid seeing / yielding inconsistent data
+ The more components there are to the data,
  the more difficult to keep consistent
+ The more steps in a logical change, the more difficult to keep consistent
+ Opportunities for automatic parallelism
  + Emphasis here on coordination
```

Let us get into some of the meat. This is an important part of the talk, because *everybody* in every language is facing this. Your programs, are they ready to run on 600 cores at one time? Are they going to use them profitably? I do not think so, but that is coming. What they are basically saying is: these cores are not getting any faster. We are just going to give you more of them. Making use of them? Your problem, software developers. Hardware guys are *punting*! Punting! They have left us out here.

So I want to be clear about what I am talking about in this part of the talk when I say concurrency, because there is a bunch of different things. We are talking about interleaved execution, and now of course, there was always threads, and there were ways to do green threads and pretend, but the reality is we are getting *real* interleaved execution now. Multiple cores means really two things are happening at the same time.

We want to avoid seeing inconsistent data, or creating inconsistent data. And as the complexity goes up, as your universe of work involves more things, it is get more and more difficult to keep consistency, especially if you are using locks.

There are also whole other sets of concurrency things that have to do with parallelism. I want to do this one logical operation and in the background this vector matrix multiply, just put it on all of these cores. I am not talking about that right now. Java has a really nice library coming up in Java 7 for doing that kind of parallel math, and Clojure has a very beautiful wrapper for it.

So here I am talking about task level parallelism. Most typical applications are going to try to leverage these cores by setting up a few threads to watch the web interactions, and another couple to deal with the database, and maybe you have some streams coming in from this data source, and maybe you have a couple of calculation engines that you are going to use. That kind of task level parallelism is what we are talking about here.

[Time 0:21:04]

```
slide title: State - You're doing it wrong

+ Mutable objects are the new spaghetti code
  + Hard to understand, test, reason about
  + Concurrency disaster
```

```
  + Terrible as a default architecture
    + (Java / C# / Python / Ruby / Groovy / CLOS ...)
  + Doing the right thing is very difficult
    + Languages matter!
```

So, this is my message. I do not care what language you are using, except if you are using something like Haskell, or something already very purely functional. If you are using a system where you are using objects or shared data structures that are mutable, it is not going to work. It is not going to continue to work. You absolutely have to find some different way to do it.

I think object oriented programs, when they get to a certain size, are just spaghetti. I have never seen one that grew indefinitely and kept being maintainable. So it is just a new spaghetti code. It is all object oriented, but it is still a disaster. It is very hard to understand a program, even an object oriented program, once you have all these relationships. Once you create a graph of interconnected changing things.

But one thing is for sure, whether or not you agree with that: from a concurrency perspective, it is a disaster. And unfortunately, it is the default architecture of all of these languages, including Common Lisp. You create objects, you get these mutable things.

And it ends up that, even if you know this, and you say, well, I will take a different approach in my applications, doing the right thing is really hard, because it is not idiomatic. It is almost anti-idiomatic to, for instance, try to do immutable data structures in Java, for instance. Although I did. That is what I am giving you in the library that underlies Clojure. But you look at the Java, you are like "Wow! That is nasty!"

[Time 0:22:33]

```
slide title: Concurrency Mechanism


+ Conventional way:
  + Direct references to mutable objects
  + Lock and worry (manual / convention)
+ Clojure way:
  + Indirect references to immutable persistent data structures
    (inspired by SML's ref)
  + Concurrency semantics for references
    + Automatic / enforced
    + _No locks in user code!_
```

So what happens now? Even if you are using a Lisp, one of the more cutting edge Common Lisps that has threads and everything else, chances are good they have these same old concurrency constructs that people in Java and C# are already saying do not work. So copying those is not really moving forward.

What do you have? Applications have direct references to mutable things. And what could you possibly do in that context? It is essentially, each program is written to be imperative, like, "I own the world." Each part of the program. I own the world.

When you do not own the world, you either have to stop the whole world, that does not work, or somehow somebody has to preserve your illusion that you see the whole world. And doing that with locks is incredibly difficult, in particular because accomplishing that is completely convention. You have to have meetings. You say: this data structure is going to be looked up from multiple threads. If you are going to touch it, make sure you lock first, and then you have all this complexity. Well, what if I am touching data structure A, B, and C? Well, we have to grab the locks in the right order, or you are going to deadlock. What if I am only using C and B? Well, you still have to grab all three. Or maybe we will have this global lock and our concurrency is going to go down. It does not work. If you are just starting to do this, just skip to the end. It does not work.

[Audience laughter]

It does not work. I can save you years of grief, headache, and nightmares. Because it is a nightmare. I have been doing this in C++ and in C# and Java for 20 years, and it just is awful. You cannot ever believe that it is working.

So, what do we do? Instead of that, we are going to say we are going to have indirect references to immutable persistent data structures. So this ref notion, it is kind of like Standard ML's ref notion. If you are going to have something mutable, you are going to have to call it out. This could change! All right?

And then, Clojure has concurrency semantics for those references. So those references are like cells. All they are going to do is point to some bigger immutable aggregate. And if you are going to want to change those cells, because they can change, you are going to have enforced concurrency semantics. And your programs in Clojure do not do any locking.

[Time 0:24:49]

```
slide title: Typical OO - Direct references to Mutable Objects

[ Figure showing 3 references to a shared mutable associative table
named "foo". ]

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

So this is what it looks like in the old school. A bunch of parts of the program are pointing to the same block of memory. The problem, fundamentally, is this: object oriented programming done the traditional way, unifies identity and value.

You want identities in your program. Like "tomorrow" is going to mean different things as your program runs and runs and runs. So you want that identity, but the value of "tomorrow", or today's date, or the time, is something that is going to change.

Unfortunately, when you say the identity of this thing is the address where we are keeping its value, game over. You cannot do the right thing. Anything can change at any time, and the consistency problem goes to the users.

[Time 0:25:34]

```
slide title: Clojure - Indirect references to Immutable Objects

[ Figure showing 3 references to a shared "ref" cell named "foo",
which has the only reference to an immutable associative table. ]

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

So what do we do instead? We have the programs refer to the identity, and the identity is that cell. That reference. And that in turn points to some immutable aggregate thing. It could be a Clojure vector or map or any of the Clojure stuff. So now we have separated identity and value.

If you want to obtain the value, you need an explicit dereference. At that point, you can have a pointer to this. A hundred people could have a pointer to this. Is that OK? Sure. This cannot change. It is immutable. Have at it! You want to look at it from a hundred threads, 600 cores, that is going to be great. That is going to perform well, because there is no locking required to look at the value.

So we are going to explicitly dereference. The values can never change. You could never see an inconsistent value, because they do not change. Somebody is going to make a consistent value, stick it in there. You pull it out. It is consistent. Objects do not change in your hands.

[Time 0:26:26]

```
slide title: Persistent 'Edit'

[ Same as previous figure, except there is now a second associative
table identical to the first one, with the value associated with key
:a changed from "fred" to "lucy". ]

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

So what does it mean to do editing? Well we have already seen with all of these data structures, we make new values by calling functions on the old values, and getting a new value. So while everybody is referring to foo, and while foo is referring to this immutable thing, anybody who wants to could be saying, "I have got a new better, improved foo!" and make one.

It shares some structure, as we saw. That is the way these data structures work. It can share structure with the one that is already in there. That is no problem. They are all immutable. Nobody who is reading this is impeded by the creation of this new value. It goes on in parallel, and it is not impeded by the fact that anybody is reading. So nobody is in anybody else's way.

Then when we want to edit, we want to change this reference atomically ...

[Time 0:27:14]

```
slide title: Atomic Update

[ Same as previous figure, but now the reference in cell labeled "foo"
refers to the newer associative table.  The old one is still there,
with the new one sharing structure with it. ]

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

... to refer to the new thing. And that is always coordinated in Clojure. There are actually multiple semantics. I have not said how this happens, when it happens. You can have different semantics for that, and Clojure has three. Anybody who is consuming the value is unaffected. If I was in the middle of doing a calculation on this, I am still fine.

[Time 0:27:36]

```
slide title: Clojure References

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

So the only thing of Clojure's data structures that can change are these reference types. There is three of them. Refs, that is an unfortunate name in this context, which are about shared synchronized coordinated change. What is the hardest problem in a system with multiple threads? It is my unit of work touches three data structures. In other words, this composite unit of work. It is very hard to pull off.

So Refs do that hardest job. Change three things, and either have all of them happen, or none of them happen. It uses a transaction system. I will show you that in a second.

Agents are more of a lightweight autonomous change model, where you say: you know what, just eventually do this, and I do not care when. I just want to return right away, and I want you to take care of this job for me. Those things are asynchronous. They are also independent. There is no way to move something from one collection to another with agents.

And Vars you already saw. Vars have concurrency semantics when you set them, because they are all thread independent. We saw we could not set it unless it was per-thread bound.

[Time 0:28:50]

```
slide title: Refs and Transactions

+ Software transactional memory system (STM)
+ Refs can only be changed within a transaction
+ All changes are Atomic, Consistent and Isolated
  + Every change to Refs made within a transaction occurs or none do
  + No transaction sees the effects of any other transaction while it is running
+ Transactions are speculative
  + Will be retried automatically if conflict
  + User must avoid side-effects!
```

So Clojure has something that is called a software transactional memory system. If you have ever worked with a database, this should all be familiar to you. You can only change a Ref, and here I am talking about that STM Ref, you can only change them in a transaction.

Any set of changes you make inside a transaction will happen atomically. Your transaction will see a consistent view of the world while it is running, as of a point in time. In addition, Clojure as an STM is kind of unique in supporting consistency rules for the values that you are going to take on.

So you have isolation, you have atomicity, and then you have consistency. You can attach to a Ref. You can say validate any data anybody wants to put in here with this function. If that validation fails, that transaction will fail. I will not take the value. You can attach those validation functions to Vars, Refs, or Agents. So they all support consistency.

No transaction sees any other transaction. Essentially what happens is your transaction sees the world as if it was at a particular point in time. And when you commit, it is as if all the changes you made happened at a particular point in time.

The only caveat with this system is, unlike databases, which sort of do deadlock detection and keep guys waiting, and sort of have a big lock chain thing, in STM systems typically, what happens is they are semi-optimistic, or fully optimistic. Clojure's is only semi-optimistic, which means you can proceed along down your transaction and determine, "Eh! I am not gonna win. I am not going to be able to maintain a consistent view of the world and have a successful commit." So do over, and that is what happens in STMs. You get an automatic retry. Which is great, and it is absolutely what you want, but you have to avoid doing any side effects in the transaction.

I cannot enforce that, but if you have side effects, and you get retried, you will have your side effects happen more than once.

[Time 0:30:45]

```
slide title: The Clojure STM

+ Surround code with (dosync ...)
+ Uses Multiversion Concurrency Control (MVCC)
+ All reads of Refs will see a consistent snapshot of the 'Ref world' as of
  the starting point of the transaction, + any changes it has made.
+ All changes made to Refs during a transaction will appear to occur at a
  single point in the timeline.
+ Readers never impede writers / readers, writers never impede readers,
  supports commute
```

Using this is so simple. Put dosync around your block of code. Any changes you make to Refs inside that block participate in the same transaction, including inside nested functions, including if those nested functions run their own transactions. They all join the enclosing transaction.

Under the hood, Clojure's STM is pretty unique. If you have done any reading about STMs, it is a multi-version concurrency control system, which is unusual. Yes?

[Audience: Can you imagine mutations to Refs that are not wrapped in a dosync as participating in just a very small transaction?]

No, you can imagine them throwing exceptions.

[Audience laughter]

It is a very clear image, too, because that is what happens. No, that is not OK. Although you can do flying reads, because the contents of any reference is consistent. So I do not care if you read it outside of a transaction. That is perfectly fine. If you want to read a bunch of things outside of a transaction, and you only cared that each one was consistent, that is also fine.

If you want to read a bunch of things and know that they are all consistent as of a point in time, even those reads need to be in a transaction.

[Audience: Sorry, I am being stupid here. You said that users have to carefully avoid side effects.]

Yes, in transactions.

[Audience: I thought this language does not have side effects.]

Well, you can call Java, you can print, you can read a database, you can ...

[Audience: Yeah, I am trying to figure out what you mean by the language being functional, and not having side effects.]

Any language that has no side effects can only heat up your computer, right?

[Audience: Yeah, right.]

[Audience laughter and applause]

Yeah, that is not my line, but many times redone.

Yeah, of course... Clojure is a realistic language in saying: of course your programs do I/O. Of course your program may want to have identities, which are associated with values that change over time. How can I help you do that right? I cannot help you with I/O. You have to just do I/O separately. I can help you completely with this data part.

[Audience: So if you were to send a function to an agent, it will produce a ...]

[Time 0:32:47]

Ah. No, no, no. No. In fact, I have not talked about agents yet. So hang on to that question. No, that is beautiful.

[Audience: TBD]

Oh, how do you start multiple threads? I do not care. You can definitely use Java's Executor framework, or Thread.start, or whatever. You can also use the Agents, which I have not gotten to yet.

[Audience: TBD]

I do not care. I guess I do not want to say, yes it would be the Agents, because that is one way, but I am perfectly happy for you to use the Executor framework of Java to launch threads. I do not care how they start. Clojure does not require your threads to be adorned, by me, in any way, for this to work. So *any* way you can start a thread in Java, this will work, including Agents, which I am getting to.

[Audience: So on the previous slide, you put that note about do not do I/O.]

Right.

[Audience: It is kind of a footnote, but it is a pretty big one.]

It is a big thing. That is true of all STMs.

[Audience: No, but like in past STMs, the type guarantees that you are not going to do I/O inside a transaction. Well now . . .]

Absolutely. I think if you are in a strongly typed system, you can have more enforcement. I am telling you that I cannot, in Clojure, because it is a dynamically typed.

[Audience: No, I am talking about in practice, how easy is it to shoot yourself in the foot? I do not think I am doing I/O here, but in reality you are.]

It is the same as anything. I mean, this is a dynamic language, so you will get run time errors. You will start seeing things print three times and be like, hmm, Oh! I am in a transaction. I got retried. And if you are in a dynamic language, you are used to dealing with problems that way. There is no enforcement. I cannot enforce it.

[Time 0:34:27]

I am not anti-enforcement. There are languages that try to enforce it. They come with very intricate type systems that are not yet expressive enough, in my opinion. But that is another way, and if you do that, you could do more. I would love to try to add just enough effect type system to this to help with that, and that is something that I will probably research in the next couple of years, but right now it is on you. It could happen. OK?

So, this is pretty cool. This makes sense from a database standpoint. This is a particularly unique thing about Clojure's STM, which is, what happens in an STM? Well, there could be multiple transactions going on that both want to increment this counter when they are done.

In most STMs that means that one transaction is going to win. The other transaction is going to retry, which means doing all their work again in order to get in there.

But it ends up that incrementing a counter does not need to be a read, modify, write action. You could do it that way. You could say: look up the counter. It is 41! Add one. It is 42! Set it back into the Ref. It is 42. Now if I do that in a transaction, and somebody else does that in a transaction, one of us will win and the other one will redo.

But, if you have something that I call commute, you could instead say: You know what? I do not care what value is in there. All I care is that when my transaction commits, it is one more than whatever it is. If you have a way to say that, then both those transactions can get all the way to the end, and one could win, and the other could win. They both can win. Neither one needs to retry. So commute is a beautiful facility. It is actually ... It was an old database thing called field calls, or something. Great idea. And it ends up that you can now build systems with even better concurrency, because if you only need to end up adding something to a map, or end up incrementing something, you can use commute instead of read-modify-write.

[Field calls are described in the book by Jim Gray and Andreas Reuter referenced below.]

[Audience: So are you going to give an example of the syntax of commute ...]

[Time 0:36:30]

```
slide title: Refs in action

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

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

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

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

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

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

Sure!

[Audience laughter]

OK, so what does this look like to you? We define foo to be that map.

(def foo (ref {:a “fred” :b “ethel” :c 42 :d 17 :e 6}))

So we have a map, but oop! It is a reference to a map, which means that if we want to see the value, we have to dereference it. This [@foo] is just reader syntax for deref of foo.

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

So just like quote, there is deref, and there is meta. There is a couple of other ones that keep me from too many tedious parens.

So we deref foo. We see that value. Notice that it is in a different order, because it is a hash thing, so it is not necessarily what I typed in.

I can use that value in calculations. This is a flying read:

(assoc @foo :a “lucy”) -> {:d 17, :a “lucy, :b”ethel“, :c 42, :e 6}

Read the value, associate this thing with it. Have not changed foo, right? I got the value out. It was this immutable thing. I did some logic on it. I got a new value. This is just that value. Nobody put it back in foo. So that is totally fine. I look at foo. It is the same.

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

I have not really changed foo.

Now, I can try to commute foo. The way commute works is it says commute, this reference, with this function, and these values. Well what will happen is that function will be passed the current state of the ref, and those values, and the return value of that function will become the new value in the ref. It is beautiful. There is also alter, which is the read, modify, write. But commute is the one that says: whenever I am done, do this.

So it is nice. It is a functional interface. You say apply this function to the value that is in that reference, and that result of that function becomes the new value in the reference.

[Time 0:38:00]

So we do this:

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

Errr! Right? We asked before, can we just do this with a single guy? No, you cannot do this anywhere that is outside of a transaction. This is not OK, because we did not start a transaction. This is one in a transaction:

(dosync (commute foo assoc :a “lucy”))

Start with dosync. I do that. I could have done a bunch of things inside this dosync, but this is just one. Now this works, and when we dereference foo, we see the changed value:

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

It is pretty easy. It is very easy. And the nice thing about it is, you end up reusing a lot of the functions you already wrote. I could test this function right now. I just did, right? In fact, I could do it with any piece of data I want.

Because you are applying these functions inside transactions means you can test these functions on values, not burden your whole system with mock objects, and all this other nightmare stuff. Test your functions. You have some data, you are going to apply this function, it is supposed to do this transformation and return this result. Do all that before you get involved in references. There is nothing about that that involves references. Then you say, oh, I want to store these things in references. OK.

[Time 0:39:07]

Yes?

[Audience: Are there any requirements that the function actually be a function, and not write any other references inside of them?]

No, you can write other references. They will all participate in the same transaction.

[Audience: If I say commute it, then it will . . .]

Same thing. You can commute some references in a transaction and alter others. It is totally fine.

[Audience: It seems like there is a way I could shoot myself in the foot here.]

No. Completely not. Because all those things participate in the same transaction. They are either all going to succeed, or none will succeed.

[Audience: So, if I am commuting, so the function call, and the function is in a commute does some other random write to some other reference.]

Oh, the commuting function. No, the commuting function has to be pure.

[Audience: Can you check that?]

No I cannot check it. I am a dynamic language. That

[Audience: You could check it dynamically.]

No. No.

That is an interesting idea, though, checking it dynamically. I guess I could do that. Yeah, I could do that one.

[Audience: TBD]

It is already in the library, sure. Right. So you say commute x inc, and that will happen when it commits.

[Audience: Question about what happens if two transactions that do "commute foo inc" commit at the same time, and they seem to be leading up to wondering whether it is possible for both to read the current value, do the inc concurrently, and both write back the same value, ending up with 1 more than the original, instead of the desired two more than the original.]

[Time 0:41:15]

They do not commit at the same time. What ends up happening is commutes get all the way through. Any transaction that involves a commute happens all the way through to commit time. Now typically, if it is read, modify, write, you will see the conflict. Two guys tried to modify foo. One will win. The other one will go back.

With commutes what ends up happening is, two guys try to commute foo, the second guy in will wait. This guy will finish, he will go. So he will do 42, he will do 43.

[Audience: How do you get 43? It read that value.]

No, no, no. Whatever happened inside your transaction in a commute, is a temporary value. It is just to make you feel good. At commit time, whatever actual value is in the Ref is going to have that function applied to it then.

[Audience: Right, so then it has to read from this . . .]

It is going to happen twice. It is going to happen twice, and it is a pure function, and that is the cost.

[Audience: OK. So there is no difference between doing that and redoing the transaction, right?]

Well, you could have done a hundred other things, though. Let us say you have a really complicated transaction, and it is completely independent from this other one, but both of you have to increment the same counter. Well, if you are really doing that with read, modify, write, you have made both of those transactions completely restart in conflict with each other. With commute they are not.

[Audience: So you are only going to do a small part of it again.]

You do whatever you like. I am just saying you have this facility and it has this property, and you can leverage that in your designs to get more throughput.

[Time 0:42:40]

[Audience: So you could also use this to do I/O conditionally, say]

No.

[Audience: I will only do I/O if I succeed.]

No. Do not. No side effects. I am going to give you a recipe for that in one second.

[Audience: If my TBD is down, and I call alter early in a transaction, right, and you commute foo, does that make the . . .]

That order will work. The other order will not work. I will track when you try to use that commuted value, which is really just a junk value to make you feel good.

It can be useful. For instance, if you commuted something that was empty, your transaction would see that it had something in it. If you did not care about the value that was in it, you can at least see that non-emptiness, which is important sometimes.

[TBD: I would like to see an example where getting this junk value back from commute is useful. It seems to me perhaps nice to have the option of a different variant of commute that only calls the function once at the end of the transaction, not twice.]

[Audience: If I commute first, and then try to alter it, that is . . .]

You are going to get an exception. Yes.

[Audience: Is there a way after you commute to find out in which order that, like after the fact.]

You can go back and read. Not otherwise.

[Audience: This has all been worked out in the database world for a long time. It works.]

It is very cool. Except software transaction systems, you read the papers about, do not work like this.

[Audience: No. No. I said in the database world.]

Yeah. This is very much database driven. MVCC and everything else is really inspired by like Oracle, and PostgreSQL and that database stuff. Yes. This is a very database-y kind of thing.

[Time 0:44:07]

```
slide title: Agents

+ Manage independent state
+ State changes through actions, which are ordinary functions
  (state => new-state)
+ Actions are dispatched using _send_ or _send-off_, which return immediately
+ Actions occur _asynchronously_ on thread-pool threads
+ Only one action per agent happens at a time
```

All right, let us talk about Agents. So Agents are for managing independent state. We saw transactions are super powerful. They let you change more than one thing atomically, which is very powerful, but it has some overhead to it. Sometimes you just do not care. You want to make a system where you have loosely coupled processes and they just need to communicate with each other.

So Agents are a way to do that. If you have read about the actor model, or whatever, Agents are sort of similar, but they are not the actor model. In particular, the actor model requires messages to read the values. Agents do not.

So you get independent state. You have actions, which are again, just functions. You say: apply this function to your state at some point in the future. It will happen in a thread pool. I do not care when. And that will transform your state. They happen immediately. You call send or send-off, you pass this function, you return right away.

And then some time later, that will happen. And it ends up happening asynchronously in a thread pool. The system makes sure that only one thing is happening to an agent at a time, and it makes all kinds of other great promises.

[TBD: Can send/send-off block if there is no memory available to store the message, e.g. if they build up a backlog because they are being sent faster than they are being consumed?]

[Time 0:45:11]

```
slide title: Agents

+ Agent state always accessible, via deref / @, but may not reflect all actions
+ Can coordinate with actions using _await_
+ Any dispatches made during an action are held until _after_ the state
  of the agent has changed
+ Agents coordinate with transactions - any dispatches made during a
  transaction are held until it commits
+ Agents are not Actors (Erlang / Scala)
```

You can dereference Agents and see their values. But remember, all kinds of stuff is happening at the same time. You have to start programming that way. If you are still programming from a perspective of, "the world has stopped any time I want to look at it", you are going to have to change that. When you have 600 cores, you have to start having looser ideas about how much of the world needs to be consistent in order for you to make decisions. Agents let you do that.

What is neat about this is: if you send actions during an action, they will be held. Also if you send actions to Agent during a transaction, they will be held. So they will happen once, and once only, when you commit. This is your way to do side effects, because Agents can do side effects. So you are doing a transaction. As part of that transaction, you send a message, "I am done! Woohoo! Tell somebody." And that will only go out when you commit. So it is kind of a cool thing that those two systems work together.

[Audience: Does Clojure itself manage the (something about thread pools)]

Yes. There are actually two

[Audience: TBD]

No, not so far, but I probably need to give you some more control to say: these set of Agents live in this pool. I do not have that now. Right now there are two pools for two different kinds of work.

And as I said, they are not actors, and the actor model is a distributed programming model. The Agent model is only a local programming model – the same process.

[Time 0:46:33]

```
slide title: Agents in Action

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

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

(send foo assoc :a "lucy")

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

(await foo)

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

So what does this look like? It looks a lot like the transactions, which is a good thing. We say def foo to be an Agent with this map value. We can dereference it. We can see what is in it. No problem.

We can send foo. Look how similar this looks to commute blah, or alter blah. So send foo, a function, and maybe some arguments. That function will be applied to the state of foo. The arguments will be passed to it as well, and that will become the new value of foo.

If I send this, and immediately go and look, it may not have happened yet, right? It happens asynchronously. "await" will make sure anything that has been sent from this thread has already been consumed. Then if I look at it, I will see the change.

So this is a powerful way to do asynchronous programming. And this is a way to launch threads. There was a question about how do you launch threads. Well, this does things in threads.

[Audience: TBD]

Effectively what happens is it puts a message in the queue, and then waits for it to be consumed. It is a way to allow you to coordinate. I want to make sure you heard everything before I tell you more. But other people could be talking to the same agent.

[Time 0:47:41]

```
slide title: Clojure is Hosted

+ JVM, not OS, is target platform
+ Java, not C, is interface language
+ Shares GC, memory model, stack, type system, exception handling
  with Java / JVM
  + JVM _is_ the runtime
  + Quite excellent JIT, GC, threading, security
+ Integrated, wrapper-free, ad-hoc Java interop
+ Libraries available for _everything_
```

OK, this is the last leg. You guys are doing great. Oh, look! We are good.

[Audience laughter]

Well, we are pretty good.

I would like to have enough time to have some good talk. So I will try to get through this quickly, so any lingering questions. Now you will see all of Clojure, maybe you can put it together.

So Clojure is specifically hosted. I have not shown you much Java stuff. But again, the integration is not a mere implementation detail. I want to be able to call Java. I want to be able to interact with it.

Java is my low level language. And the nice thing about a language like Java being your low level language is it actually shares the same GC, and memory model, and threading model, and everything else. Think about all those issues you have when you try to start doing multi-threaded programming, or even just GC. The interactions between GC and calling out to C. it is a nightmare. Yeah, I am sure people are like uuuuuhhhhhh. You know, it is *bad*.

So JVM is my runtime. I cannot say enough about how high quality the infrastructure is in these JVMs. You have a wide range of choices. Sun has their thing, but you can also get JRockit, and IBM has excellent VMs. I mean this is a really mature platform with a lot of competition. A lot of extremely bright people working on it. This is excellent technology. As goofy as Java is, the JVM is stunning technology. You cannot ignore it. These just in time compilers and these GC systems are the state of the art.

Clojure allows wrapper free interaction with Java. A lot of languages that are hosted on Java, they put their own classes around every Java class. There is none of that. You call Java, you get a Java call.

Of course the other advantage of being on Java is there are libraries for absolutely every single thing you could ever possibly want to do. They may not be the best, but they are there. Some are better than others. I am not saying Java is perfect, or anything like that. But there will be choices of libraries, in every domain.

[Time 0:49:46]

```
slide title: Java Integration

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

So what is the integration? What form does it take? There is as much unification as I can possibly make happen. For instance, Java Strings are immutable. Pfff! That is perfectly fine for Clojure strings. They are immutable. They are Unicode. We are done. Clojure strings are Java Strings.

The numbers I use are the boxed Java Numbers. The only thing I had to add to that set was Ratio. Otherwise, I used the capital I integer, and things like that.

All my collections implement the Collection interface of Java, so if you want to use them from Java, you can pass them. If you have a thing that uses Collections, or Iterable, they are all Iterable.

All my functions are Callable and Runnable. So if you want to use a Clojure fn, a Clojure closure as a callback for Swing, you do not have to do anything. You just pass it, because it is Runnable.

And I showed you before, the Clojure sequence library works with everything Java. You know, Iterables, Strings, Arrays, the works.

You can implement Java classes and interfaces in Clojure, which is nice. It keeps you from going there.

In addition, I do support primitives. Java primitives as locals. So if you have a core piece of code that is your inner loop, where you want to max out the performance, you can make a few little type declarations and say this is an int, this is a long, this is an array of floats, and you will get the Java primitives that correspond to that. In addition, when you start doing math with those things, you will get unboxed primitive math.

And that does not mean you will necessarily get unsafe math. So there is two different levels. There is do I have the boxing, which you can turn off with the declaration, and then you are going to have to tell me explicitly, I want bad plus, and I will give you bad plus. And it turns out that the boxing is much more overhead than the overflow detection.

[Time 0:51:36]

```
slide title: Java Interop

Math/PI
-> 3.141592653589793

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

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

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

(into {} (filter #(re-find #"java" (key %))
                 (System/getProperties)))
-> {"java.specification.version" "1.5", ...}
```

So this is what that looks like. You can access static members just directly. It looks a lot like that namespace syntax, doesn't it? Because that is really what static class members are. They are things in a namespace, where the namespace is the class, so I just unify that.

You can call methods. This is a macro, dot dot. It says put dots in between everything. As many as you want.

[Audience laughter]

System dot get properties dot get Java version. And it ends up, this has no more, it has fewer parentheses than the Java version. You rapidly have fewer parentheses than the equivalent Java. Anybody from Java is like, "Oh, Lisp! Parens!" You just show them *this* one, right?

Do to a new Jframe, add a new label with hello world, and then pack it, and then show it. It has a third of the number of parentheses and lines, well, it has no lines, one line. You can make Java completely easy.

And interacting with Java stuff, if you use the sequence library, you do not even know it is Java. You totally do not care. Things like this. This does not look like Java, right? This looks like Lisp. into a map, filter, this is shorthand for fn, so this is like the most concise lambda expression you can do in Clojure. Do this regular expression find across, ooh! This is actually Java! This is the only Java part of this. But it is just transparent. It does not turn your Lisp code into Java code. It turns Java code into Lisp code. That is the beauty of it. So do not be afraid Clojure is Java with parens, because there have been Lisps from the JVM that went that way.

[Time 0:53:10]

```
slide title: Swing Example

(import '(javax.swing JFrame JLabel JTextField JButton)
        '(java.awt.event ActionListener) '(java.awt GridLayout))

(defn celsius []
  (let [frame (JFrame. "Celsius Converter")
        temp-text (JTextField.)
    celsius-label (JLabel. "Celsius")
    convert-button (JButton. "Convert")
    fahrenheit-label (JLabel. "Fahrenheit")]
    (.addActionListener convert-button
      (proxy [ActionListener] []
        (actionPerformed [evt]
      (let [c (. Double parseDouble (.getText temp-text))]
        (.setText fahrenheit-label
          (str (+ 32 (* 1.8 c)) " Fahrenheit"))))))

    (doto frame
      (setLayout (GridLayout. 2 2 3 3))
      (add temp-text) (add celsius-label)
      (add convert-button) (add fahrenheit-label)
      (setSize 300 80) (setVisible true))))
```

This is what it looks like to do Swing. Some imports. You know, it is Lisp. This is a callback. We are proxying. This is a one-off implementation of a Java interface. It is an anonymous inner class, effectively, that you would use in Java. And again, this is so much shorter. The Java for this is three times the size.

[Time 0:53:32]

```
slide title: Experiences on the JVM
```

```
+ Main complaint is no tail call optimization
+ HotSpot covers the last mile of compilation
  + Runtime optimizing compilation
  + Clojure can get ~1 gFlop without even generating JVM arithmetic primitives
+ Ephemeral garbage is extremely cheap
+ Great performance, many facilities
  + Verifier, security, dynamic code loading
```

I really like the JVM. It is a great platform. It is a serious piece of technology. My main complaint is lack of tail call optimization, as I have said before. I am hopeful that it will get that.

I cannot also say enough about HotSpot, and I am using HotSpot, which is like a Sun branded thing. There are equivalent ones from IBM and from JRockit, and whatnot. But these JITs are amazing. For instance, I talked about that primitive math. Actually Clojure does not emit any of the primitive arithmetic byte codes. If I can get to a static method call on primitive types, those JITs will finish, and finish emitting (a) the byte codes for Java for arithmetic, and (b) turning that into the right instructions for the CPU. In other words, I only have to get as far as a method call, and these optimizers get me to optimized machine code, which runs as fast as C, and often faster now, because they are doing run time profiling. There are things you can optimize only by looking at a running program, and that is the way these optimizers work.

As I said before, ephemeral garbage is extremely cheap. I leverage that. You should, too. If you are not aware of it, it is great.

And there are so many other features. There is the verifier, the security, there is a lot of... the ability to load code over the wire. Java has a lot of stuff under the hood.

[Time 0:54:59]

```
slide title: Benefits of the JVM

+ Focus on language vs code generation or mundane libraries
+ Sharing GC and type system with implementation / FFI language is huge benefit
+ Tools - e.g. breakpoint / step debugging, profilers etc.
+ Libraries!  Users can do UI, database, web, XML, graphics, etc right away
+ Great MT infrastructure - java.util.concurrent
```

So the benefits are, you can just focus on your language. If you are looking to write a language and want to target this. Of course, I recommend just starting with Clojure.

[Audience laughter]

But for me, it really let me focus on the language. The infrastructure was great. Sharing with the implementation language, especially these kinds of things: GC, big deal. You get tools for almost nothing. I emit the correct stuff, the correct debugging information, you can debug Clojure, step debug, breakpoints, the whole works. I mean, I am a LispWorks user. That stuff only came a couple years ago to LispWorks. You have excellent debugging, excellent profilers, and your choice, all work out of the box with Clojure.

And then there are libraries for everything. Clojure users were instantly doing all kinds of stuff I had no ... I could not possibly imagine in advance. They were just talking to databases, and doing net kernel stuff, and big document processing, map reduce, Hadoop, lots of cool stuff. There is also excellent infrastructure on the JVM for implementing things like the STM. This java.util.concurrent library is really good.

[Time 0:56:05]

```
slide title: There's much more...

+ List (Seq) comprehensions
```

```
+ Ad hoc hierarchies
+ Type hints
+ Relational set algebra
+ Parallel computation
+ Namespaces
+ Zippers
+ XML
```

There is a lot more to Clojure. It has list comprehensions. Ad hoc hierarchies I mentioned before. With type hints you can get Java code to run exactly the same speed as Java, so Clojure is fast.

I hinted at the relational set algebra, and the parallel libraries for doing parallel computations – automatically leverage all the CPUs to crunch big arrays. And namespaces. There is zippers, like Huet's paper, XML, and a whole bunch of other stuff.

[Time 0:56:35]

```
slide title: Thanks for listening!

[ Clojure logo ]

http://clojure.org

Questions?
```

[Audience applause]

Any questions?

[Audience: So what does generic plus turn into in Java byte code, and then into, in practice, in execution?]

Generic plus, it turns into a Java type test quickly, and then branch to, in my library, individual types, which implement logic for that type. They will do automatic promotion to the bigger type.

[Audience: But how bad is the type dispatch in Java byte codes?]

The thing is that these optimizers optimize a lot of that stuff away. And in fact they really can take something where you have not hand coded a fast path, and they will chop it out, and they will stick it over here, and they will say, well, let me try running that in all cases, and they will have a little very inexpensive test they will use to see if it is still the right way to go, and then it will slow path the others.

So I could not say, because each JVM does different things, but it is pretty fast. The main optimization required moving forward is something that they call escape analysis, where if you new up a boxed number, and immediately consume it, they should be able to completely avoid that allocation, and that is something that is just coming into the next generation of optimizers for the JVM. So that will help all of these things.

And in addition, like I said, if you really care, and you need to go right to the metal, you say int x and you are done. No dispatch, no nothing.

[Audience: Could you describe your development environment when you are writing in Clojure? Do you use a debugger or something to debug . . .]

I use Emacs. There is Emacs plugins. There is SLIME. There is all kinds of cool stuff already ported for Clojure. There is an Emacs mode, there is a VI mode, which is not as sophisticated. There is an environment for NetBeans, a plugin for NetBeans, which has syntax highlighting and everything else. That has the best debugger support right now. The NetBeans plugin. But you can use any debugger. You can use JSwat, or jdb, or profile with YourKit, which is just a gorgeous piece of technology. Really, all the tools work.

But in terms of editing, I edit Java code with IntelliJ and I edit Clojure code with Emacs. And I do not debug.

[Audience laughter]

[Time 0:59:04]

[Audience: I was just about to ask, is there a Clojure language level debugger?]

No, I am not going to do that. Because these debuggers are so good. Oh, these debuggers, when debugging Clojure, debug Clojure language. When I am setting breakpoints and stepping, you are stepping through Clojure code. You are setting breakpoints in Clojure code. Clojure emits a mapping file, and line number information that is necessary for Java debuggers to say, line 42 is over here in the Clojure file.

[Audience: But if I am typing away at the REPL, and somewhere down inside somewhere I take the car of 5, and it says, no you do not]

Attach a debugger. You can attach these debuggers ... you do not have to be in debug mode in Clojure. If you start it with the debug API alive, you can attach a Java debugger to it whenever you want, including right then. But I am not going to reimplement Common Lisp style debugging, because this stuff is ... You know, it is not the same, but it is really good, and I do not have time for the other. If somebody wants to do it, they could, but.

[Audience: You said you have also programmed in C#]

Sure.

[Audience: TBD the .NET TBD]

Yeah, they do not always do it, though.

[Audience: Well, it can always do it later.]

Yeah, well later is still ... I am waiting for later, too.

[Audience: Is there anything in Clojure, other than STM stuff, which TBD]

No, I started Clojure for both platforms. Clojure ran on both platforms for a long time. I just got tired of doing everything twice. I also wanted to make a decision about what will library authors do? How will the library for Clojure grow? And as much as I am expert enough to do C# and Java, and many other people are, everybody is not. And it is a big thing to take on having to do everything twice, and do the least common denominator, and everything else. So I abandoned this C# port maybe 2 years ago now. It certainly would be possible to do. Yes. C# has everything you need to do this.

[Time 1:01:02]

[Audience: How long have you been doing this?]

Three years. It is been out since last October, so it is coming around a year. It is really taking off. I mean, I have a big group of users. I have over 500 people on the mailing list, and we get 5 or 600 messages a month, and I have thousands of SVN hits a month, so I think I have a couple of thousand users.

[Audience: What would you say is the most interesting TBD]

Well, you know, I do not hear from people unless they have a problem.

[Audience laughter]

So I wish I knew. I do know there is a couple of companies using it, and it is sort of like their secret weapon thing. You know, they do document processing and some other stuff. I have seen interesting database work. I have seen interesting graphics work, and XML things.

There is some really neat stuff. Like there is a zipper library which allows you to sort of pretend you are manipulating something in a destructive way, when you are really not, and producing a changed version of a data structure. And somebody wrote a really neat, Xpath-like way for processing XML functionally with that.

[Audience: How large is the runtime?]

Excuse me?

[Audience: The runtime. How large is it?]

400K. Fits on a floppy disk. If you could find a floppy disk. And half of that is the ASM library that I use to emit the bytecode. Clojure is about 250, 300K.

[Audience: You have listed Common Lisp, Oracle, and Standard ML as influences. Anything else?]

Sure. Haskell, and all the STM work, and basically anything I could get my hands on to read. A lot of database stuff went into the STM. That old Andreas Reuter book, or whatever, which will make you really never want to use a database that is not MVCC ever again. Really whatever I could get my hands on to read. I like to read.

["Transaction Processing: Concepts and Techniques", Jim Gray, Andreas Reuter, 1992]

[Time 1:02:56]

[Audience: TBD]

Clojure versus Scala. Well, Scala is statically typed, and has one of these sophisticated type systems. I think Clojure is more strictly functional, because it defaults to immutability, and defaults to ... the data structures are these persistent data structures, but it also has a recipe for concurrency built in. There is some library work for that, for Scala, but Scala is really still a multi-paradigm language where the objects are still those old share-the-reference-to-the-block-o-memory objects, and I think that is doomed.

[Audience: So you mentioned running on these Azul's 600 core boxes.]

Yeah. I highly recommend it. It is so much fun.

[Audience: So does the performance actually scale fairly linearly, or]

Well, he found it did on this problem. Of course. You know the whole thing with STMs – everybody is like STM is this holy grail, and when you try to say something is the holy grail, it is going to fail, because nothing can do everything.

You still have to architect your programs. You still have to have some sense of what is the contention profile in my program. You still may run into performance bottlenecks that you have to solve by changing the contention profile of your program. I mean, it is not a silver bullet. What it is, is a way to write something that is correct easily. And the tools for doing that. But there is still going to be work.

[Audience: So the moral is ... I was being sort of idiomatic here. You can probably add a 100 times more cores than you an add ...]

If you have ever written programs with locks, you can be wrong on two cores right away.

[Audience: You can go to one core!]

You can be wrong on one core. It is so easy to be wrong. It is so easy to be wrong.

[Audience: Could you explain your logo?]

My brother did it for me. It is pretty. It is a little yin yang, a little lambda. Yeah, he is good.

[Time 1:04:57]

[Audience: In Haskell, it seems like every time a Haskell programmer runs into a speed or space issue, he goes through and makes everything strict. Have you found that in Clojure? Have you gotten around it? Is it just not happening?]

It is not the same thing. Because in Haskell, everything is lazy, so that is a big difference. When everything is lazy, it is very difficult to tell when you are going to have this explosion in memory requirements. And that is the problem people run into with Haskell.

With Clojure, the laziness is limited to these sequences. And in particular, I am very careful that inside of a sequence operation, as you move through, which is usually what you are doing – usually you are just – maybe you have a concatenated set of operations. But you are walking that window through your data set. I make sure, by cleaning up the stack that you do not retain the head.

It is easy to retain the head accidentally in Haskell, in which case, as soon as you have realized the thing, you have realized the thing. As soon as you have walked it, or touched it, you have the whole thing.

In Clojure, you walk it, you do not have the whole thing. In fact, it can erase as you go. I mean, it can get GCed as you go. So I have not seen that be a problem. In fact, I think the laziness is a real attribute in the low memory utilization of Clojure programs.

[Audience: How much of Clojure is written in Clojure, and how much is written in Java?]

It is about 15,000 lines of Java and 4,000 lines of Clojure.

[Audience: TBD]

Absolutely. In fact, when you define a Java class in Clojure, you end up with a Java class that is extremely dynamic, much more dynamic than a regular Java class. In fact, all it is is a set of stubs that hook into Clojure Vars. So you can, no, it is not funny. It is very powerful.

[Audience laughter]

You can write Java classes in Clojure that you can fix while they are running. You can fix the definitions of methods. You can also write transactional Java objects in Clojure. You can write Java objects in Clojure whose data can only be changed inside transactions. It is cool.

[Time 1:07:07]

[Audience: I was a little surprised by your answer about the source code being three quarters Java. Are you planning on moving towards a system that is more bootstrap?]

Well, you know how it is when you have something that works?

[Audience laughter]

There is a reason that the data structures are written in Java, and the interfaces are written in Java, and that is so that consumption from Java is straightforward. Clojure does not yet have all of the mechanisms in Clojure's Java-emitting syntax to do the lowest level, like field type declarations you need for the highest performance classes. And I am not sure I want to go there, because it will end up being Java with parentheses. By the time you have private, protected, whatever, and all the declarations you need to support exactly what you would do in Java.

So, the biggest thing that is in Java, that probably I would like not to be in Java, is the compiler itself, which is about 4,000 lines of Java. And people have asked about bootstrapping that. That certainly would be possible to do. It is the old "it is working right now" kind of problem. And I would rather do new things.

So I do not have a big thing, and I know it is like sort of a chest beating proof of whatever, and I really just do not care.

[Audience laughter]

[Audience: Could you do a mobile subset of Java?]

Some people have tried to do that. Unfortunately, even 450K is a bit much, but one of the other challenges is just that I do need a fair amount of Java concurrency mojo. Like I only go as low as Java 1.5, because I need a lot of the java.util.concurrent stuff to support some of the things. So that is definitely something I would like to do.

Another thing that is tough on the mobile phones is dynamic code generation. The byte code loaders, and some of the things are secured that way. One of the things on my to do list is ahead-of-time compilation to byte code, which will let me target some of those static platforms, like Google Android.

[Time 1:09:02]

[Audience: There is a standard problem with promises in Scheme, where it makes streams moderately high when you filter out all of the integers, numbers that are larger than 10 billion, and then you try to get one of them. If the promises were implemented naively, you would get a string of 10 billion promises TBD Do you do the appropriate trampoline?]

I do not do it that way. No, that will walk through – you will have to walk to get to 10 billion, but it is going to throw everything away on the way.

[Audience: So . . .]

So it is not a chain. There is no realized chain. If you have not hung on to the first thing, you do not have it. If you say drop 10 billion from a cycle, whatever, and then take 3, you will have the head of that 3, and you will not have 10 billion of anything. It is not a chain of promises.

[Audience: So there is a trampoline.]

No. It cleans up as it moves. That is the trick. What happens is, there are suspended expressions, yes? In the first, in the head. When they are realized, they are discarded. Now the realized versions are cached. Now we move to the next. Now we have new expressions. Now we evaluate those. Now the expressions are let go, of the value that they are.

In addition, if this was held on the stack, and this is a nested call, or some reduce or something like that, no one else is looking at this, we erase the reference to this, boom! What is left? One guy, 2 values. No lingering anything. So no is the answer. It does not build up.

[Time 1:10:48]

[Audience: TBD]

No, I cannot talk about Haskell's implementation. All I can say is, it is a known problem with some Haskell programs, that the laziness has this interactions that can make it difficult to predict the memory usage of your program in advance.

I am saying in Clojure that is not the case because the laziness is limited to these sequences, and I have just told you their behavior that makes it so that they do not accumulate anything that you have not asked to hang onto. If you hang onto the head, and then go 4 billion nexts, or rests, well you are now holding on to a 4 billion item thing. But you did not have to do that, and I will not do that accidentally for you.

[Audience: ??? That is kind of a weak spot in ??]

Yeah, it is.

There is hope. Right now I have the same that Java does. Arrays of arrays of arrays. Multidimensional arrays are arrays of arrays. But I have seen some cool stuff in IBM's X10 language, where they take a giant array, and then they pretend it is multidimensional, even though Java does not support it, and they have some very neat syntax for using little tuples to do subranges and to do multidimensional indexing. And I was inspired by

that, and maybe if it comes to it, I may copy something like that. That is the neatest I have seen that gets around those problems.

[Audience: TBD]

Datalog. Anybody know any good Datalog implementations? Yes, I would like to add some ... Where Clojure is going is I would like to add some declarative data access stuff. Not really Prolog, because it is not that great for database work, but Datalog, which is sort of a subset of Prolog, for which a lot of research has been done in optimization, would be a good mini-rule system plus data access that is pretty. You know, it supports recursion and things like that. So I am still researching whether I am going to roll my own or use Iris Reasoner, which is a Java one that is open source.

[Time 1:12:50]

[Audience: TBD]

Maps. Yes, there is actually something called a struct in Clojure, and it is an optimized implementation of map, where you say: I know I am going to have 10 million of these things, that are going to have fields a, b, and c. So repeating a, b, and c in all of those little maps is kind of a waste of time. So I can make that go away. But it still implements the map interface, and for all intents and purposes, it is a map. You can still add extra stuff to it, just like you do. You can still assoc them. So it is nice. It is the best of both worlds.

[Audience: And for multiple values do you ...]

Multiple values, you can do whatever you want, but I do not have them on the stack. I cannot do them. Clojure is stack compatible with Java, so there is no multiple returns.

[Audience: Is there any interaction that we should know about between metadata and transactions?]

No. No, I mean, like I said, most of the transformative functions of values will be metadata preservative. And you are just going to pass a function to the transaction to say: modify the value with this function, which will be something like assoc, or something you wrote based on assoc. It will still preserve metadata.

Metadata is very neat. You can do multimethods that key off metadata, so do this if it is data I do not trust. That kind of polymorphism is something we need, and doing everything with types really has limited us.

[Audience: So I read that at the JVM summit there was some talk about these dynamic languages at the summit. Did you learn anything interesting about these TBD]

I learned that those guys need to learn a lot more about languages that are not a lot like Smalltalk.

[Audience laughter]

No, I mean their first cut is to try to support the popular languages there, which are JRuby and Jython, which have very similar object models. So they have a distinguished receiver, and stuff like that in the MOP. And I did try to tell the guy, look, just move the parenthesis over, and let us just realize that it is all functions. Can we just do that? There is really no such thing as methods.

[Time 1:15:02]

[Audience: You made a note on the discussion board about how, basically, having all these defns, TBD generation was not a big deal for Clojure. Can you elaborate on that? Was TBD]

[Audience: Yeah, so you do a defn and it gets implemented as a class, right?]

Oh, the classes, right. And there were some of these languages that need to generate tons of classes in order to, like for instance do these wrappers. I said Clojure has wrapper free access to Java. A lot of these languages do not.

In addition, a lot of these languages have, like Ruby and Python, they were written by guys who were writing something fun for themselves. They were not thinking about compiling those languages when they wrote them. Now these Java guys doing Jython and JRuby are trying to compile them, and it is very challenging. So one of the things they are trying to do is polymorphic inline caches, and all that stuff.

And when they generate any code in Java, you have to generate a class, because that is the unit of code. So they are doing a lot of ephemeral class generation, and classes have an impact on this Permgen. Clojure really does not do any of that.

[Audience: TBD]

Yeah, which is not that many ... unless you really need to fix that function a lot.

[Audience laughter]

You should be OK.

[Audience: Were you doing genetic programming?]

Well, you know, those things are all compiled, so even if you see fns, that is not like a new class at runtime. It is a class at compile time. See, that is the stuff that I am talking about.

Yeah, if you were writing genetic programs, and you needed to do a lot of evaluation of function definitions, sure, you would get into permgen issues, yes.

[Time 1:16:44]

[Audience: Any advice or hints about what sort of applications do you think would shine in Clojure? ? the Lisp side ?]

I think anything. I mean, right now, if you want to write a program ... Clojure is designed to be useful anywhere you would use Java, and Java is being applied, you know, people already asked about phones and these 600 CPU machines, so it is a wide range. It is the same question for Lisp. I mean, what is Lisp good for? Pretty much anything.

[Audience: What would be dramatic? What would work really well as a 600 core machine TBD]

Anything that you can think of a way to parallelize. I mean, by the time you get to 600 cores. I actually think most applications will never be able to leverage 600 cores. There will be certain kinds of things that can, and I talked about that task parallelism. Most applications will be able to number their tasks in the low dozens, I think, really. So the job of those mega machines will be to run multiple of those applications.

The main thing for Clojure is: you can write that multiple dozens of tasks parallelism without error. And in terms of application domain, every application I have ever written has needed to do that, and struggled to get it correct. So I think it is good for a wide variety. I would not want to pin it down.

[Audience: Can you talk about your STM implementation? For instance, TBD]

No it is a runtime thing, a transaction.

[Audience: So how does that work?]

It is a thread local thing that we are in a transaction. So a transaction is going to be in a thread, and then the transaction starts in the thread, and there is a transaction object. References, all interaction with references, are going to test for the presence of that object and make sure that you are in a transaction. That is how we got that exception before.

[Time 1:18:41]

[Audience: So you have to go through references.]

You have to go through references. That is the beauty of this STM. You have to have identified the things that can change. Either they are in an Agent, they are in a Var, or they are in a Ref.

[some people talking at once that are difficult to understand]

[Audience: That is where you track the writes and reads in order to TBD]

Correct, except I do not track reads. Clojure's STM is unique. It does not track reads.

[Audience: Oh, because you are doing MVCC.]

Right. it is MVCC.

[Audience: What about the condition system? You know, restarts . . .]

[One description I like describing Common Lisp's condition system is a chapter in this book: "Practical Common Lisp", Peter Siebel, 2005

You can read that book on line. The chapter on the condition system is here: http://gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html]

You know, I looked at that. I mean, I actually think that the condition system is a hard coded special case in Common Lisp. That if you had some more general primitives, the condition system would just be one thing you could implement with them.

In Clojure, you have two things you can utilize to help you with those kinds of problems. One is the exception system, which allows you to travel up and down the stack.

The other is that dynamic binding of functions. You want to have a function. I am going to bind this function to something that you should call if you have a problem. Well now you have got that ability to call some code from above down low to see what you should be doing. Combining dynamic functions and exceptions gives you some of the condition system.

[Audience: But it does not give you restarts.]

[Audience: You can, once you throw an exception.]

But you do not throw an exception. That is what I am saying. Let us say you are somebody who needs to talk to the file system. And you say: if I have a problem talking to the file system, I am going to call "I had a problem talking to the file system". Somebody above you can bind that to do whatever you want. You are going to call that and use the return value, and not throw.

Right, and so I think that gets you most of the way there. Dynamically rebinding of functions and exceptions, I think, is enough. I like the condition system, but it seems to me to be a hard wired special construct, rather than two . . .

[Time 1:20:34]

[Audience: it is just throw with some TBD]

Right.

[Audience: Did you do TBD for that to be rebindable, or is the important thing just the presence of dynamic variables.]

I have looked at whether or not you should, instead of having everything be potentially dynamic, you call out the things that you are going to allow to be dynamic. And the problem with that is, you do not know. Now your program is running, and you say, "Aww! We are having this problem in here. I wish I could log this function." Well, you did not declare it dynamic, so now we have to stop the running system.

And so, rather than do that, I have made it so that all Vars are potentially rebindable, and I think that is a better way to go, only because I was able to make the check extremely inexpensive, so you can incur it everywhere. So far that has worked out.

The cost of calling them out would be: you could not, on an ad hoc basis, say "Oh! I wish I could dynamic rebind this."

[Audience: I take it you do not have first class continuations?]

[http://en.wikipedia.org/wiki/Continuation]

No.

[pause]

[Audience laughter]

[Audience: Well someone had to say it.]

There was a talk about it at the JVM. I mean, if the JVM supports them, I do not know why I would not, but otherwise, I think they are too complicated for me.

[Audience: First class what?]

Continuations.

[Audience: TBD]

No.

[Audience applause]

[Audience: The way the symbols are TBD]

[Time 1:22:23]

No, no. I think the system is good. If you are coming from Common Lisp, and you get your head around how this is different, you will see. It does get rid of, I think, 80 plus percent of the hygiene issues really just go away. Because when this macro was written, list meant the list function in Clojure. It will emit something that will make sure that it gets that binding. And that is what hygiene is about. But it does not have a sophisticated renaming system.

[Audience: Well, would you want to possibly get a hygienic macro system?]

I also do not understand those too well. Especially the implementation. It is like the sourdough implementation of that. No one ever reimplements it. Everybody just passes this one thing around, which means if your language is not Scheme, you have to ... what do you do? I seem to see this one implementation of hygienic macros reused, let us say, everywhere. It is complicated!

[Audience: Do you have a white paper or something specifically on macros TBD]

No, just the material on the site. There is a lot of material on the site. But we have a good group, and any question or dialog you start will inform everyone. I encourage anyone who is using Clojure to join the Google group and I will be happy to expound on any details.

[Audience: What is zipper?]

What is zipper? Some French guy whose name I cannot quite pronounce. Huet? Huay? Or Huet? H U E T, wrote a beautiful functional pearl on zippers, which is a way to navigate through a tree-like data structure as if it was imperative. And you can go through, and you say boom, and you get here, and say replace this with something else. And then you can say up up up up up, and you go back up to the root of the tree. And what you have then is a changed thing. There is an implementation of that here in Clojure.

[http://en.wikipedia.org/wiki/Zipper_(data_structure)

In latest Clojure as of Aug 2009, the Clojure implementation is in namespace clojure.zip]

[Audience: I think we should probably stop here with the questions at this point.]

OK. Thank you very much.

[Time 1:24:34]