Inside Transducers

- **Speaker: Rich Hickey**
- **Conference: Clojure/Conj 2014 - Nov 2014**
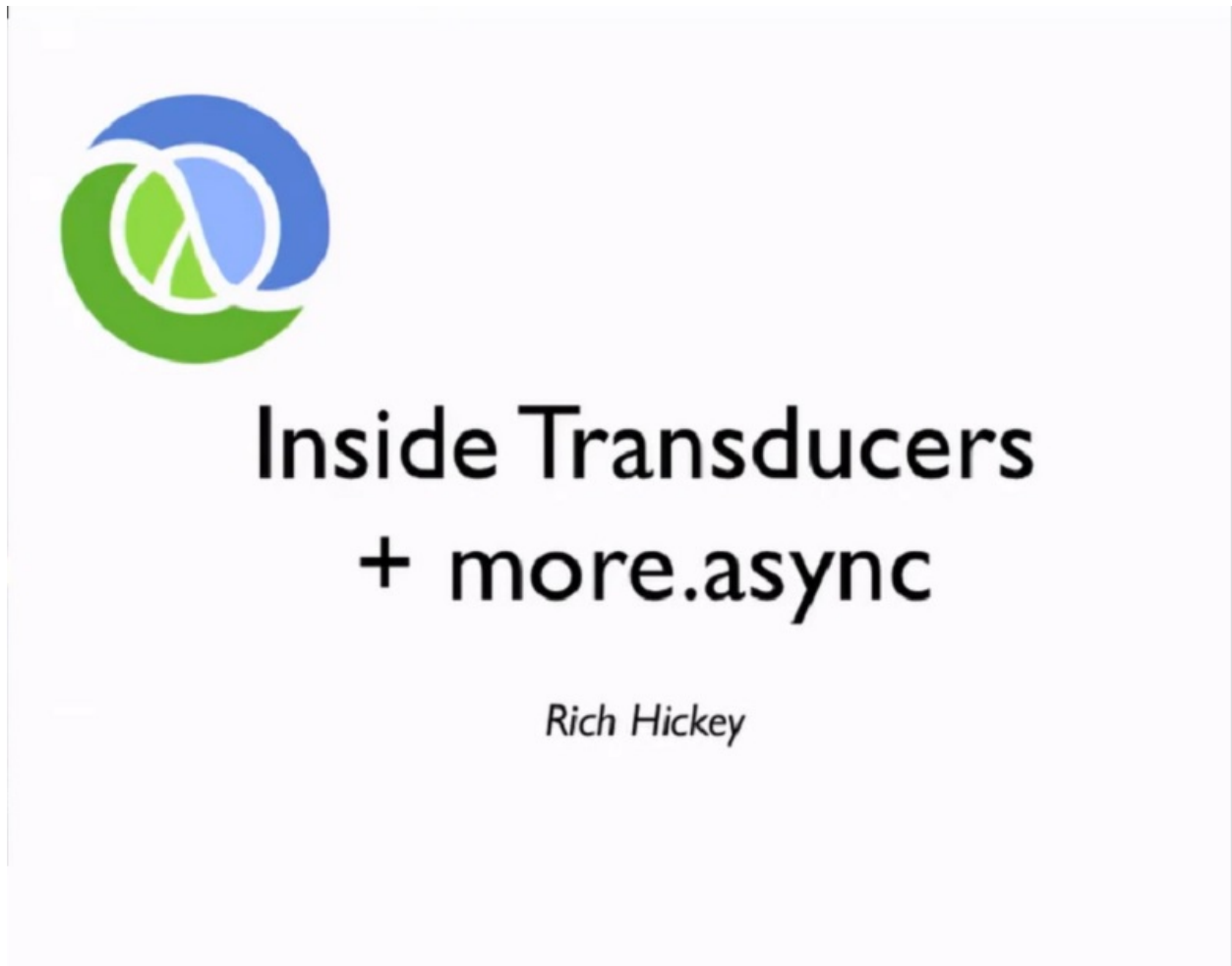- **Video: https://www.youtube.com/watch?v=4KqUvG8HPYo**

Figure 1: 00.00.00 Inside Transducers

Hi. Thanks. Thanks for coming. Thanks for using Clojure. Every year this gets bigger, this event, and this community, and this thing that's become Clojure, and it couldn't be more exciting to see this and see all the old friends here. And a lot of new people have been introducing themselves to me and telling me what they're doing, and that's always very exciting. So it's really fantastic to see what people are accomplishing, people who are learning, people who are really killing it with Clojure, succeeding in businesses, helping other businesses or starting their own businesses. It's just totally fantastic.

This year is especially exciting for me because, finally, someone has been able to come to the Conj who has been critical in Clojure's development. I think it's very difficult to understand what it takes to make Clojure, and I'm not talking about my own activity, but just what's involved in sort of dropping what you're doing and spending your retirement money and having your spouse watch all that and be like, "Go for it!" and be supportive of that and to listen endlessly to minutia about Clojure and transducers and all this stuff that nobody should ever have to listen to.

[Audience laughter]

So I'm really happy that my wife Stephanie was able to come today, and I think she deserves a big round of applause - Clojure's biggest supporter.

[Audience applause]

Without whom there would be no Clojure. I can promise you that.

All right, so today we're going to talk a little bit about inside transducers – this is not the same talk I gave at Strange Loop – and also a little bit about some new stuff we're doing in core.async.

Figure 2: 00.02.18 Transducers

Transducers are great because you could probably put a new picture up every time you talk about them and it would still be a valid transducer. Actually, these are much better examples of transducers than burritos are.

[Audience laughter]

So I'm actually not going to spend a lot of time describing what transducers are, but just as a show of hands, how many people saw the Strange Loop talk? All right, everybody else, you're going to have to go back and fill in some more of the details because I don't have enough time to both explain the ideas behind transducers and the implementation stuff that I really want to focus on today.

There will be three slides now that are in common with that talk. What is the idea behind a transducer? It is that we were writing map and filter once again for core.async and realizing that probably was not the

Figure 3: 00.02.32 What are They?

best thing to be doing. Is there some way to extract this stuff out? Finding an example of how to do that in the reducers work by basically saying map and filter are just functions that transform another function, some reducing function, and they could be actually reducing, or it could be something that has the same shape as reducing.

And the way to think about that overall is that map and filter can be thought of as functions that change some process in a particular way, so you can define them as process transformations. And, as soon as you do that, you can use them for push. You can use them for poll. You can use them for data. You can use them for events. I think it's a super important abstraction away from context, which allows you to now express things and use them in different contexts, which is sort of what we keep trying to do.

What Kinds of Processes?

- ones that can be defined in terms of a succession of steps
- where each step ingests an input
- building a collection is just one instance
- seeded left reduce is the generalization

Figure 4: 00.04.00 What Kinds of Processes?

So when we talk about being able to modify a process, what kind of process? I mean, there are a lot of processes with a lot of different shapes, and it ends with the transducers can transduce one particular kind of process, which is pretty generic. Right?

There's any process that can be defined as a succession of steps where each step takes some new input and sort of incorporates it in the process. Right? Building a collection is an example of that kind of a process, right? You have a collection so far, and you have one new thing, and you put it, and you add it to the end. But it's just one example.

And I think too much thinking about map and filter has been connected to collection processing or list processing or sequences for too long, and that's not the bottom. I think the bottom is the step, not the stuff.

So seeded left reduce is the generalization of this.

Figure 5: 00.04.55 What "transducer'?

You know, we like our words in Clojure, so reduce means to lead back. I'm not going to talk much about the ingestion part in the rest of this talk, but transduce means to lead across, and that's definitely what transducers do. They're leading the input across the series of transformations on the way to, well, maybe reduction or maybe something else, some other process. But they lead them across a transformation process, so the name makes sense.

In Clojure, moving forward, we're going to have transducers be returned by all the sequence functions that you're used to, so map of f is no longer – I mean, it can still take a collection and do what it always did, but when it's not passed a collection, it returns a transducer, so it returns a function that takes a step-like function, a reducing function, and returns one. Map f returns a transducer. Filter with a predicate returns a transducer. That transducer modifies the process. The filtering transducer takes the inputs and maybe doesn't use them sometimes. That's what filtering is.

Mapping is taking the inputs and applying a function to them before you use them. It's modifying a process and, similarly, take, mapcat, and whatever. They all return transducers, so map isn't a transducer. Map of f returns a transducer.

What I want to talk about today is sort of the insides. Maybe hopefully you'll leave this talk feeling comfortable being able to implement the transducer or being able to implement a transducing context.

Figure 6: 00.05.27 Transducers

Implementing Transducers

- take and return a 3-arity fn

- arity-0 flows through to wrapped fn

- arity-1 is used for completion, if none flow through

- arity-2 is reducing step, you can morph input, call nested (or not), expand etc

Figure 7: 00.06.27 Implementing Transducers

What does it take to implement the transducer? We implement them in Clojure as a function that takes a 3-arity function and returns one. That's the transducing job. The step functions themselves, the function you have to return and the one you can be expected to be passed have three arities.

The first one is optional, arity-0. When it's used at the bottom, it's used to fabricate the initial result, so you can think of plus when you call it with no arguments returns the identity for plus, which is zero. And multiplication returns one, which is the identity value for multiplication. If that's present, we want the transducers to preserve it. So far, that's the only use of arity-0 in transducers, so we just flow it through.

Arity-1 is used for completion. And if you don't have any completion to do – a lot of transducers will not. You don't have any accumulated state. You're not keeping track of something as you go along like every step is its own world – then you have nothing to do with arity-1.

In other words, somebody says, "We have this value. Do you have anything more you need to do with it because we have no more input? I'm calling you with just the return value. Do you have anything to do with it?" Usually your answer will be no, and so you just call through to the function that you are wrapping.

The big one is the arity-2 function, right, which takes the results so far, and that's a very generic idea. That could be anything. It could be nothing too. There are reducing functions and step functions that don't use that. They just return nil and you're just passing nil around, and nobody knows better from that.

But one way to think about it is that there's this something that's standing in for the process. It may be the result that's being built up. It may be something the process needs in order to do the next step. Whatever that is is passed along with a new input. Your job as a transducer is to preserve the result so far, right? You can't mess with that.

And maybe you do something with the input. You can morph it. That's what map does. You could ignore it. Right? Not use the function given this input. You say I'm only supposed to be using even numbers and, when I see an odd number, I just skip. And you skip by just doing nothing. You don't call the nested function, and you just return.

You can do expansion and various other things in transducer, so the arity-2 is the reducing step. It has the same shape as the function you pass to reduce. Results so far in a new input returns the result next.

Everybody says I don't have enough code in my talks, so now I have a big slide that says Code.

[Audience laughter]

You can imagine – no.

[Audience laughter]

That's a little bit tiny, isn't it? I had this all worked out so the page downs take me to the next thing. If I make this bigger, that's not going to work. Can people see this, or does it need to be bigger?

[Audience response]

It's okay. Great. This is an excerpt of map. This is not all of map because it's missing the meat of map, but it's the transducer returning arity of map, mostly. But it serves as a nice example.

Map takes a function. That's the function you're going to apply to every input, so you're going to map increment. Increment would be f. That's the ordinary argument to map, and this is the arity that doesn't take the collection, so just map inc. That would be f.

Then what map is supposed to return then is a transducer, so a transducer is a function that takes a reducing function and returns one. So rf is the reducing function that you're wrapping, in a sense. And you have to return one. Like I said, the reducing function or the full shape of a reducing function is an arity-3 function. You must have arity-0, 1, and 2.

Map does not accumulate anything from step-to-step. Right? Every step of map is its own world. I take the input. I call the function on it, and I use that input later. I have no state in between calls so, A, I have nothing to construct. Right? This is really for the bottom so that this arity is always just called a nested function.
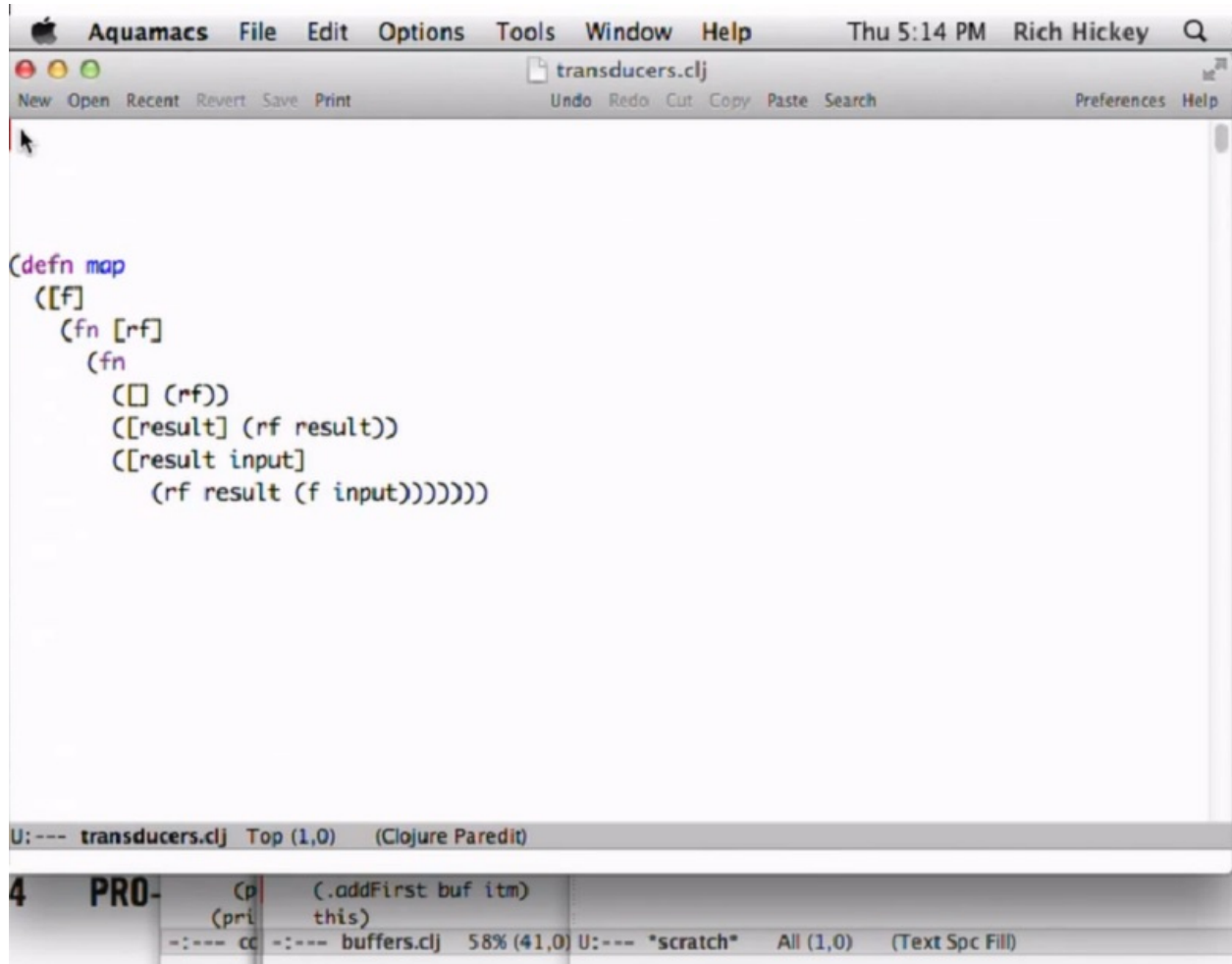
Figure 8: 00.09.21 Code

Figure 9: 00.09.34 screenshot

When I pass the result to complete, I have nothing to do. Right? Map has no more work to do, so you just call the reducing function of the result. Maybe they have something to do. It's not up to you.

Then, finally, there's the actual step, and this is the essence of mapping is in this step. Given a result and input so far, and some function you want to transform, mapping is taking that function, the results so far, and passing a transformed input to that function, right? Calling f, right? We imagined we were calling map with inc. Inc would flow down here, your increment to input, before you use the wrapped reducing function.

We've modified a process. Rf was a process. We don't understand what it is, but we just changed it a little bit. We said if you were getting a number, you're now getting a little bit bigger number or whatever f does. This is the simplest example I can give of a transducer.
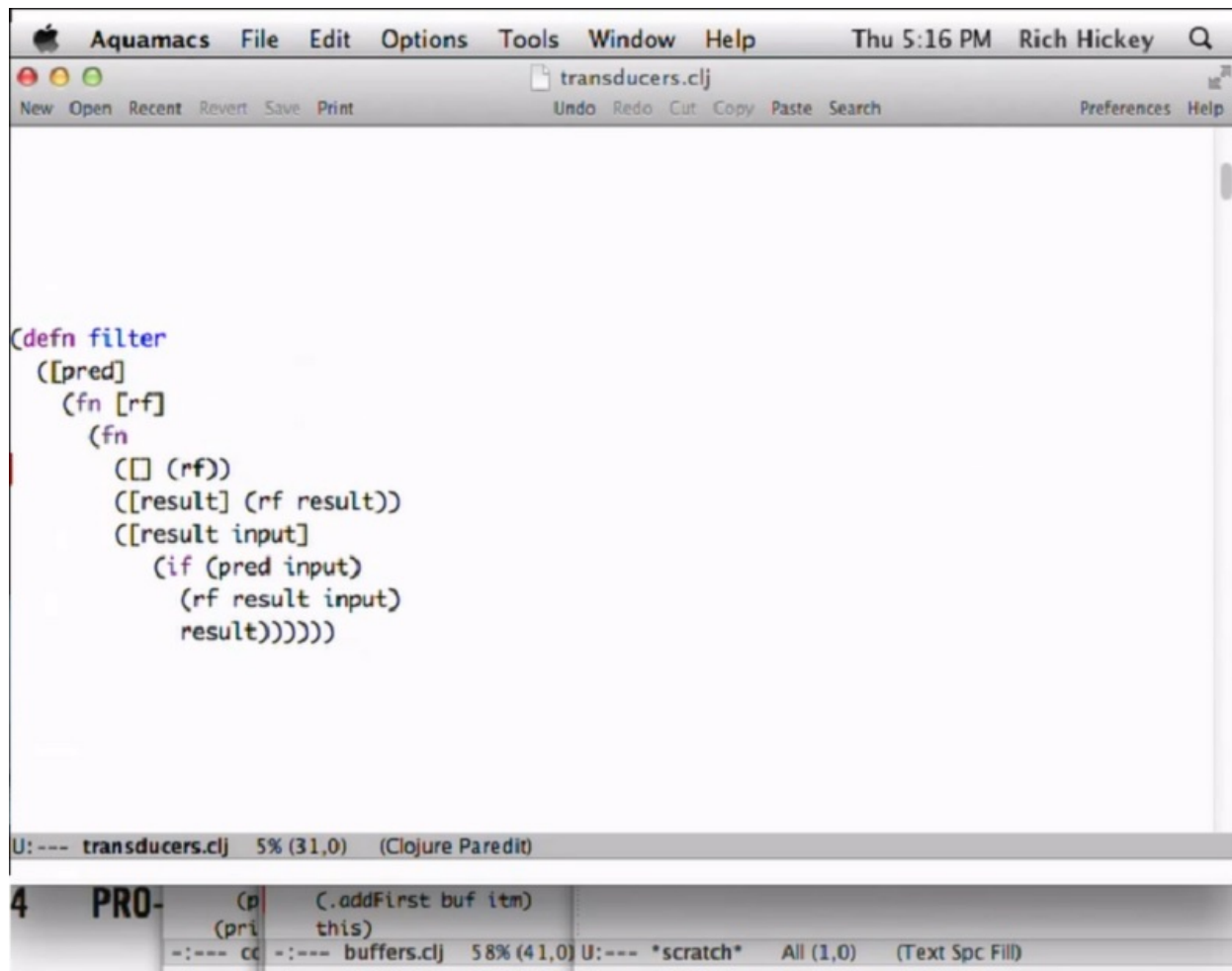

Figure 10: 00.12.09 screenshot - build slide

If we move on – oh, look at that. It's still working. So the first example is map one-to-one. Filter is one of the sort of three cases. Map is one-to-one. That's sort of the easiest. Filter needs to use the result or not, right? Again, we see arity-0, so filter takes a predicate. It says, if this predicate is true of the input, do the work; otherwise, don't.

It's interesting because that means of describing what filtering does is a process-oriented way of talking about it. It's not like if you're given something and we're supposed to be filtering and you're not supposed to use it, then give me nothing. That's a data orientation. That's not really what filtering is. Filtering is ignoring, and that's exactly what we're going to do.

Again, the first two arities are the same as map. When we're given a result and input, we say if the input

satisfies the predicate, use it; call the nested reducing function with that input. In other words, it's okay input. It passes the predicate. We use it. Otherwise we don't call rf. Don't do anything. This is the way any kind of reductive or conditional transducer would work. If you don't need to use the input, you don't call the nested function. You just don't do it. That's the pretty much the most efficient way to do nothing is by doing nothing.
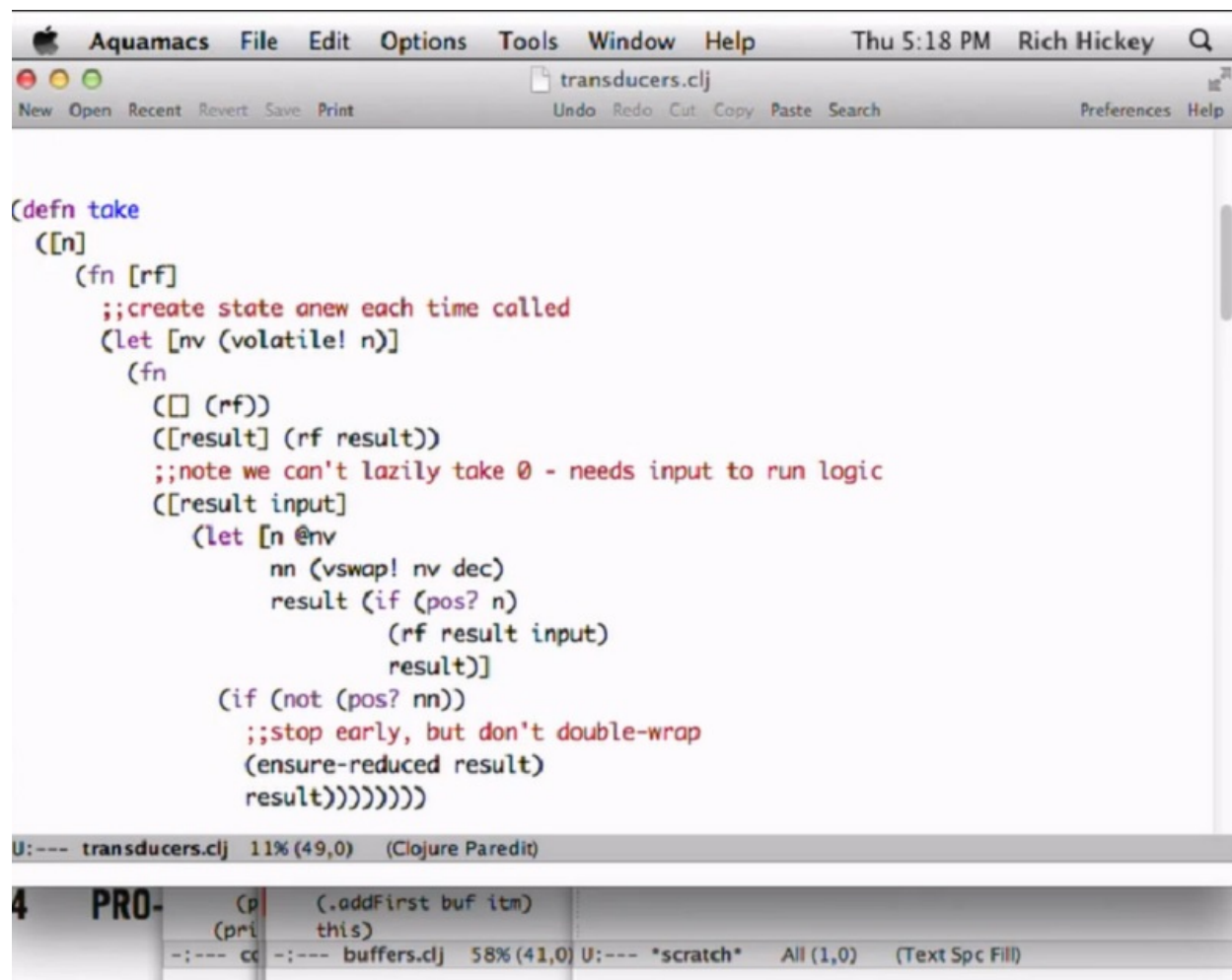
[Audience laughter]

Figure 11: 00.13.36 screenshot - build slide

That's what we do. I'm not going to dig totally into take, but I wanted to show you an example of a stateful transducer, so take has some logic that crosses across execution. As you go, you're only supposed to use so many of the inputs, and then you're not supposed to use anymore. But I wanted to show you a few things just so you have some rules of thumb about creating a stateful transducer.

I think there have been some blog posts that sort of made a lot of stateful transducers. The fact that we can do stateful transducers is great. That doesn't make stateful transducers sort of the most important thing about transducers or where the emphasis should be.

But if you are to write a stateful transducer, some rules you need to follow are: Make sure you – no, it was supposed to be "a new." I'm just looking at one word and not the sentence.

When you have a stateful transducer, if you have some local state, make sure you create it a new every time you're asking to modify the reducing function. So you'll note that this let that creates a volatile variable

here is inside, is underneath the call. So every time you're asking to transform a new reducing function, there's a new state associated with that. That's so that independent transduction stacks are independent, including the fact that their state is independent.

Like the others, there's no accumulation in take. Take does keep track of a counter, but it's not accumulating values that need to be flushed out during completion, so both init and complete are just dummy implementations. But there are some interesting things. In a particular, you can only be so lazy with a transducer, so take 0 can't be lazy because none of the logic of a transducer runs until an input has been supplied, so there has to be some input supplied.

But here it's the basic logic of just saying, "Are we done yet?" The one other thing you want to see here is that take is something that aborts the process, so we're going to take until a certain point. Then we're not going to go anymore. The way we do that is with reduced, which is something we already had in Clojure as a way of saying, "Stop reduction." Right? We're saying transduce and transduction and transducers are sort of an abstraction of reduction. But we still use reduced as a way to signal that we want early termination of this process.

And ensure reduce is a nice way to say we called our f here and we might have gotten a reduced value back. We don't want to reduce it more. We just want to make sure it's just got a single wrapper. So ensure reduced is another helper that's present I wanted to call your attention to.

All right. Okay. What just happened to me? Here we go. It's my cursor. I have to use the right keys. All right, so that's take.

Mapcat is the other kind of transducer, so we've seen one-to-one. We've seen elision, right, maybe we won't use the input. We've seen early termination in take, and now we have expansion.

What does mapcat do? It says I have a function. I'm going to call it on something and presume that that result is itself a collection, and I'm going to integrate that collection in the result. Now mapcat itself sort of streams the answer into a resulting sequence, but mapcat the transducer sort of concatenates into the transduction, which may be sequence building or something else - whatever is produced by calling f on an input. You call f on an input; you get a collection. What you want to say is all of that collection now gets incorporated as input. That's what we want it to feel like.

So we get to see an example of transducer composition here. Mapcat is just a composition of map and cat. If we mapcat f, we get map f composed with cat. Cat is a new thing. We haven't seen cat before because – I don't know if cat is really an idea with sequences alone, but it certain is one with transducers.

Cat is a little bit different in that the cat function is a transducer. It's not a function of some argument that returns a transducer. Cat is a transducer because it has nothing else to do, so it just takes a reducing function and takes its input and reduces it into the result. It's going to take a reducing function and take input, which is presumed to be a collection, and incorporates it as input, which means it uses each piece of that collection as an argument to the reducing function. We're just going to reduce, with the reducing function, all those inputs.

We got one collection of five things as an input. We took each of those fives things and used them as an input to the function we're wrapping. That's what catenation is. And so cat is a useful transducer. Although, once you have mapcat, you'll probably end up using that alone, but you can use cat directly. If you don't have a function that returns a collection and you just have collections, you can just cat them if you're composing transducers, and so cat is kind of cool as a transducer.

All right, so this gives you expansion. Cat gives you the ability to say I had one input and it yielded multiple inputs to the next stage, so it expanded. We had one-to-one. Filtering is potentially reductive. Taking is abortive, and catenation is expansive. That's sort of the whole flavor of things. Right? We can be one-to-one. We can have fewer things than we've got. We can have more things than we've got. We cannot use everything that we get.

Oops. I just lost my whole system here. Let's just do this.

Okay. And the final and most complex one I want to show you is partition-by. Again, I don't really want to have you think through and understand all the implementation of this, but I just want to point out a couple
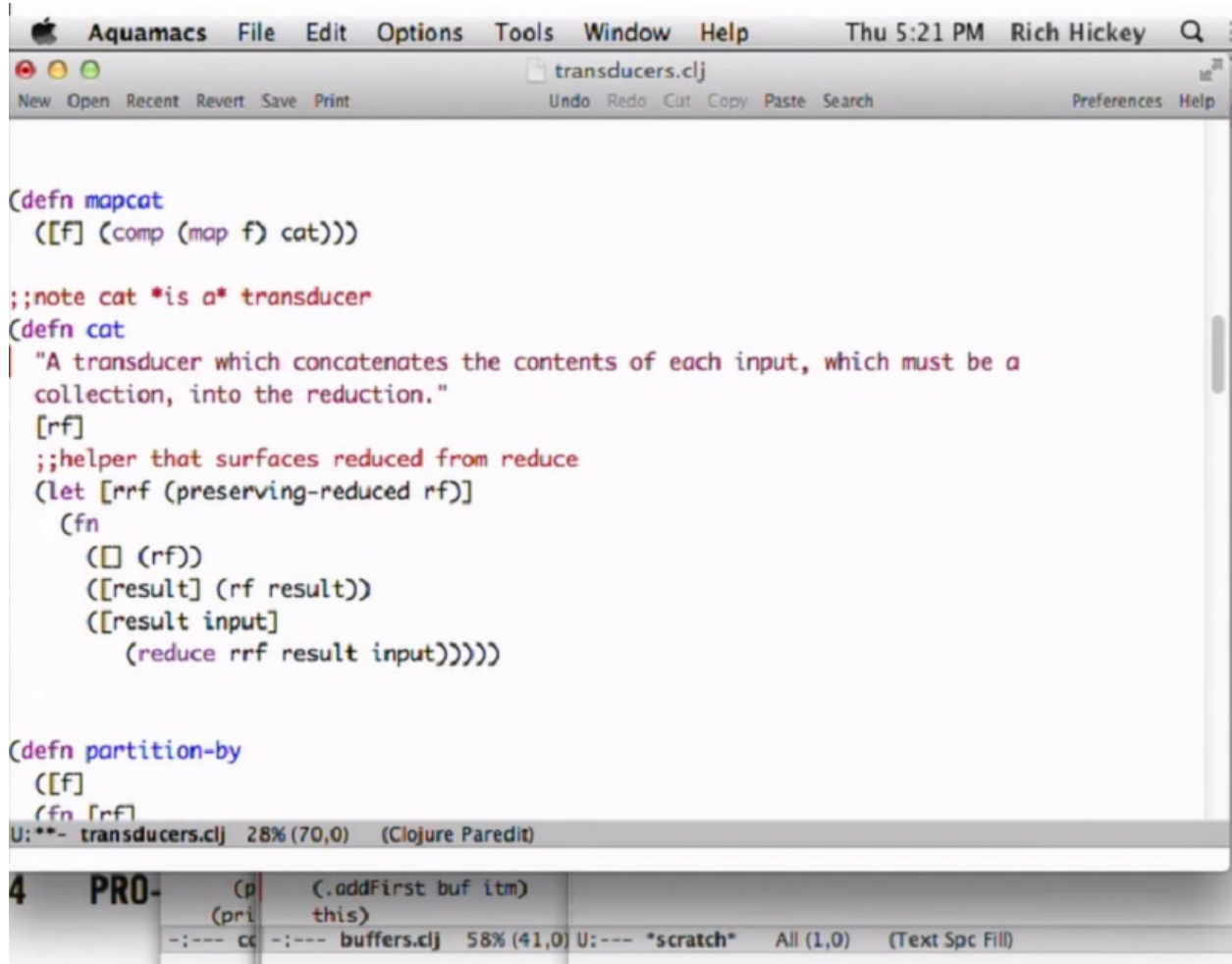
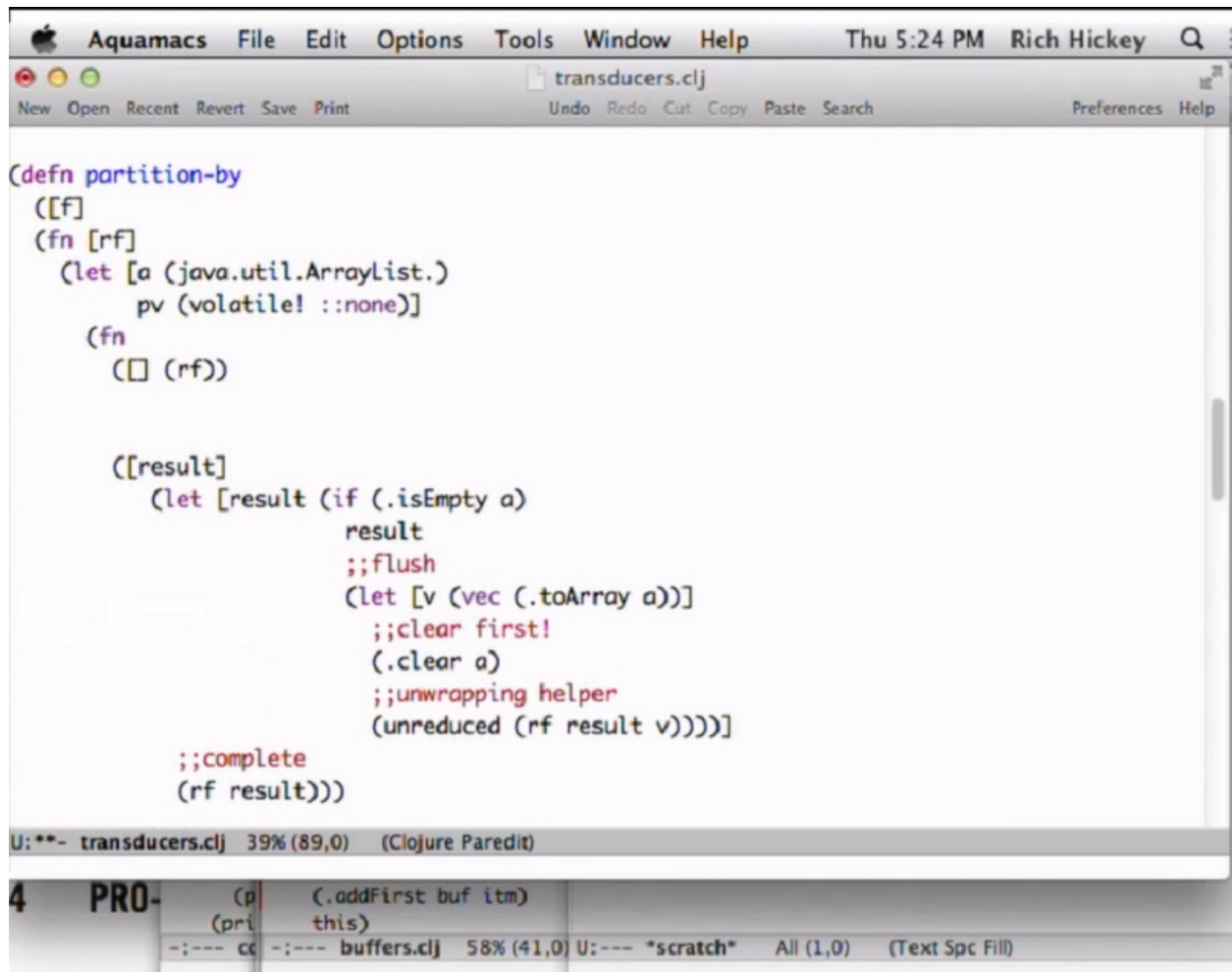Figure 12: 00.16.51 screenshot - build slide

Figure 13: 00.20.09 screenshot - build slide

things. Again, it has state, so we're going to create that state anew each time we're asked to wrap a reducing function. This is the first case in which we're going to be building up some intermediate results that need to be flushed during completion, so this is the first transducer I'm showing you that has an implementation body of any merit inside complete.

Partition-by, if you don't recall, is a function that takes a function and it calls it on every input. And, as long as it's returning the same value, it keeps that stuff in the same results collection. And, as soon as it returns a different value, it yields that collection that's been building up so far and starts building up a new collection. So it sort of snips your data into collections every time the value of f returns a different result on the input that is passed.

Eventually, if the process that you're transforming terminates, now it may not. There are kinds of transformations that never stop, in which case partition-by is going to yield as this function changes, but if it never changes again, you'll never get that last segment. But if it's a transduction process that terminates, like reduce, it's given a finite collection, it will terminate. It will end. That means that this arity will get called saying, "Okay, there's no more input. Time to wrap it up."

And partition-by says, "Whoa! I have leftovers." Right? "The last time this function returned a value, I snipped it off. I returned a collection. Now I've been building up a new collection, and you're telling me we're done. What am I supposed to do with this stuff?"

And the answer is, "Now is your chance." You get one more use of the result. You can use the 2-arity version as much as you need, so it ends up that what's going to happen in partition-by is it's going to take the collection that's been building up. And you don't have to see the details of that, but it's going to pass it to the wrapped reducing function one more time with a value.

I have one more value. I've been building up these partitions. I made one last partition. I'm going to call you with a two-argument version saying here's the results so far and here's a value. That's going to be typical. If you've been building up results, you're going to make one more or maybe you'll reduce, but you're going to make a set of calls to the step function, the two-argument step function, in order to flush your results. And you have to be aware of the fact that that may itself call return reduce saying, "I've seen enough from you. I don't want anymore input."

Then, finally, you have to complete. If you've been building up a result, and you're asked to complete, you can call the wrapped reducing function as many times as you want, ordinarily, to flush out what you've been building up. Then you must call the reduced argument on that, the reduced flavor or arity of this to complete with the final result. Okay? So that will be typical of a transducer that builds up over time.

Then some tips on your actual step function, so partition-by is also interesting in that it's incorporating input and it's building this interim result. It ends up every time it produces an output, it clears the array it's been building up so far. But it may, when it's passed that down, have the function it's wrapping say, "I've seen enough." Right? It returns reduced.

At that point, we don't want to add anymore to the input. In other words, if you're accumulating, as soon as the function under you, the function you're transforming has told you it's done, it's seen enough, in other words it's terminated early, you shouldn't accumulate anymore. You should put yourself in a state so that when you're asked to complete, you say, "I don't have anything to flush," because you know the function underneath you does not want to see it. It doesn't want to see any more from you. It said I'm done, so that's what this bit does here. So that would be typical of that.

Again, no one is forcing you to write stateful reducing functions or transformers, transducers. They're tricky, so you can just use the ones other people write. But if you're interested in doing it, these are the caveats and tips.

All right, look, that was code.

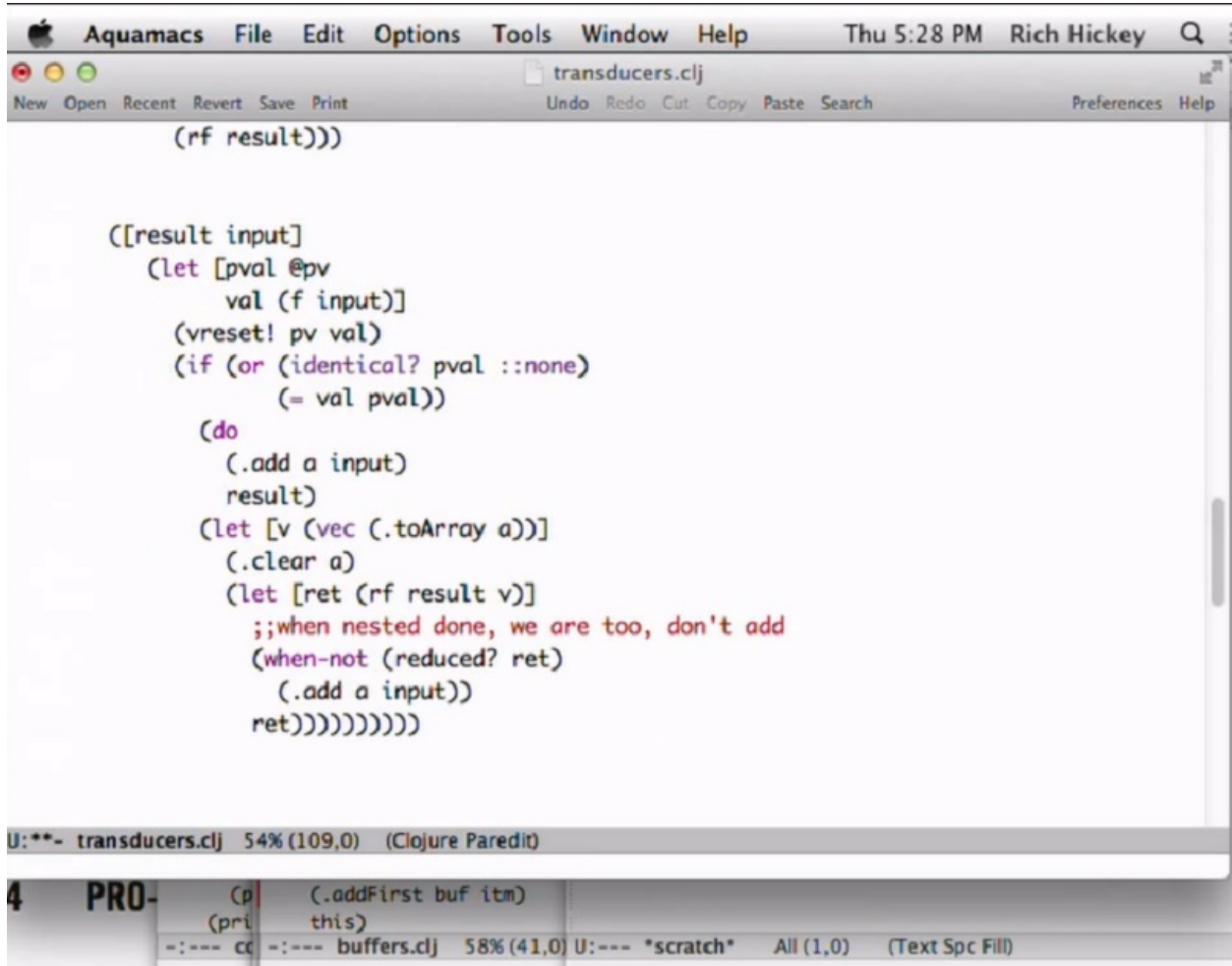Okay, so that's writing a transducer. What about writing a transducible context? What does that even mean?
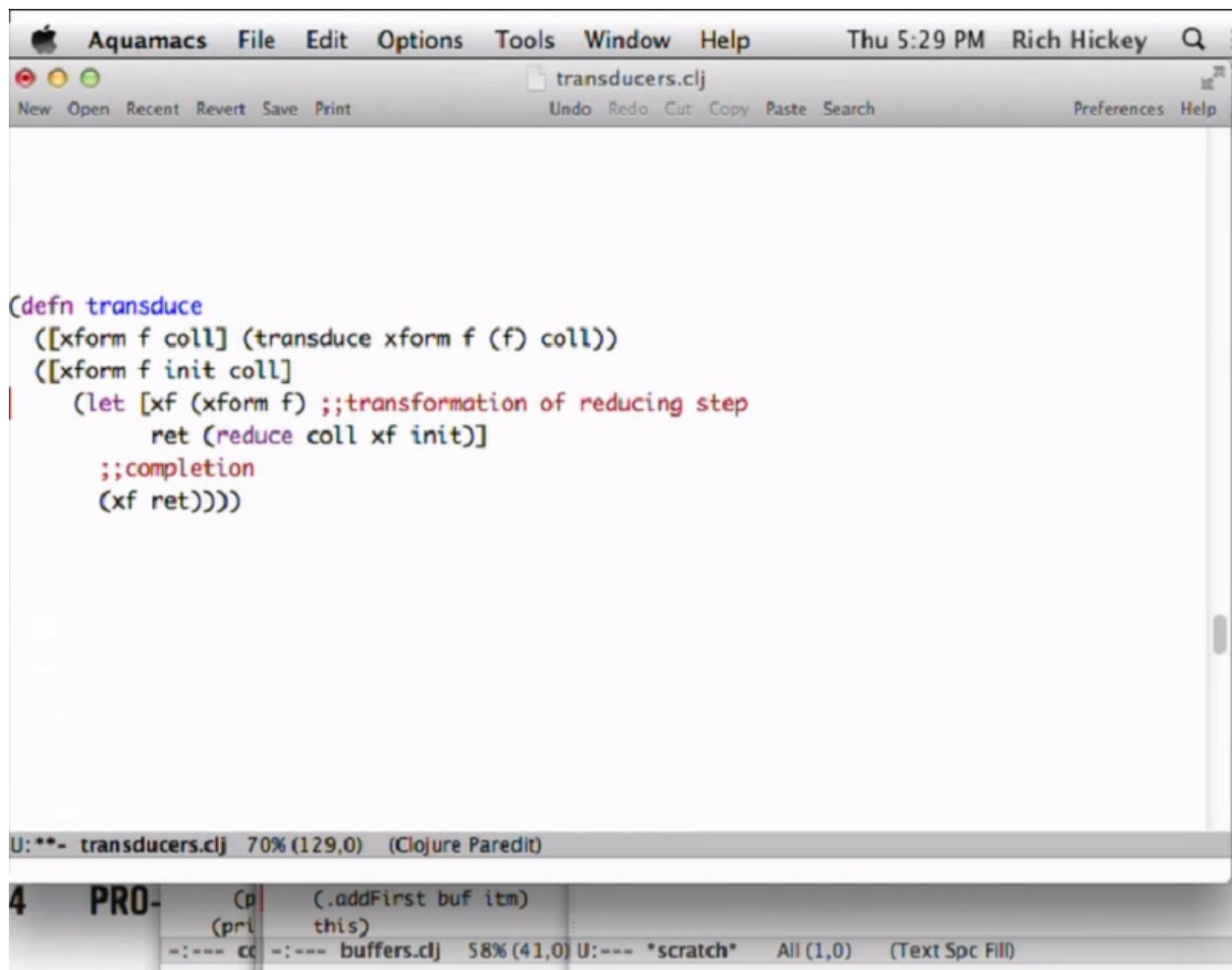
Figure 14: 00.23.49 screenshot - build slide

Figure 15: 00.25.11 screenshot - build slide

Figure 16: 00.25.17 Transducible Contexts

Well, a transducible context is some thing, some process that can incorporate a transducer. We're adding a bunch of these to Clojure. One is a new function called transduce. It's analogous to reduce, but it takes a transducer, and it will transduce the reduction process with whatever transducer you pass it.

Into has been enhanced to take a transducer. Into just used to just say, "Dump this collection into that collection," you know, pour it in. Now you can pour it in and stick a transducer in the middle of that, so everything coming from one collection gets transduced and then put in.

Sequence will take a collection and produce a lazy view of that collection, having pulled it through a transducer. And, finally, there's channels that will accept transducers, so these are all examples of transducible contexts. Again, it's even less likely you're going to write your own transducible context, but you should be able to and hopefully seeing the insides of these will help you.

Figure 17: 00.26.22 Implementing

What do you need to do if you're implementing a transducible context? You have to figure out some way to talk about what you're doing as if there was a step function. Now, a lot of times there will be a step function, and it'll be plain what it is. Other times, you may have to just think a little bit and say, "Is there a way for me to think about what I'm doing as if it was a function of some result so far and a new input producing a result next?" As long as you can do that, you can transduce it. So select a step function with that shape.

Then you're going to have to have some ability to say, "Give me a transducer." It's your job as a transducible context to apply the transducer to your step function. That should never be a user space activity. Why is

that?

What is a transduced function like? It's got this whole stack of stuff. All right, what happens if you transduce with partition-by?

That transduced step function is now potentially a stateful thing, which means you do not want more than one person to have that, which means you should never hand it to somebody else. You should never hand a transduced reducing function around, which means all transducible contexts should accept a transducer and, inside them, you call the transducer on the step function and make the transformed step function. And you just keep that to yourself. Right?

Then, every time you have a new input, maybe you're proactive. Right? You've got a collection in hand. Your job is to reduce it, so you just have your stuff already. Or maybe you're reactive, right? A channel is reacting to stuff coming in from somewhere else. Well, every time there's a new thing, you're just going to call the step function, the transform step function on it. Maybe it's data source or it may be an event.

If, and only if – you do not have to do this – if you have a notion of being done. If you're processing a collection, you will have the notion of being done. If you're dealing with events from the outside world, you may not have a concept of being done. You're like an infinite process. You're done when they pull the plug on the computer or something.

But if you do have the notion of being done, when you're done, call the arity-1 to allow any flushing to occur. Call the completing arity.

All right, so let's look inside a couple of these. Transduce is the most basic. It's just like reduce, except it has a transducer and it transforms the reducing function. It also has to add the completion step at the end. It does imply that the reducing function must support arity-1, which it may not do.

Let's look at code. Do I have code? Oh, look - code.

All right, code. I talked too much about this before showing it. Okay, so what does transduce do? It calls reduce. All right, but before it calls reduce, so it supports the same form of reduce here, but it actually has different semantics of reduce.

Who knows what the semantics of reduce are when you call it with a collection and no initial value?

No one, right. No one knows. It's a ridiculous, complex rule. It's one of the worst things I ever copied from Common List was definitely the semantics of reduce. It's very complex. If there's nothing, it does one thing. If there's one thing, it does a different thing. If there's more than one thing, it does another thing. It's much more straightforward to have it be monoidal and just use f to create the initial value. That's what transduce does, so transduce says, "If you don't supply me any information, f with no arguments better give me an initial value."

Note though, there's no requirement that you support that. You don't have to have that arity. You don't have to allow people to call transduce with no initial value because sometimes there's just no good made up from nothing initial value. Somebody needs to think about an initial value or supply some inputs to your process to get a starting value. Not everything can be made from nothing. It's easy to come up with zero from nothing. But it's not easy to come up with a channel from nothing or other kinds of things, you know, event systems. You don't have to support this.

What will happen is if somebody gives you an f and it doesn't support this arity, it won't work. That's okay. But the normal form takes an initial value in a collection, so it's just the same shape as reduce, except it has this new first argument: transform. This is the reason. This different treatment of f is the reason why we didn't just slap transform into reduce. That's why we have transduce.
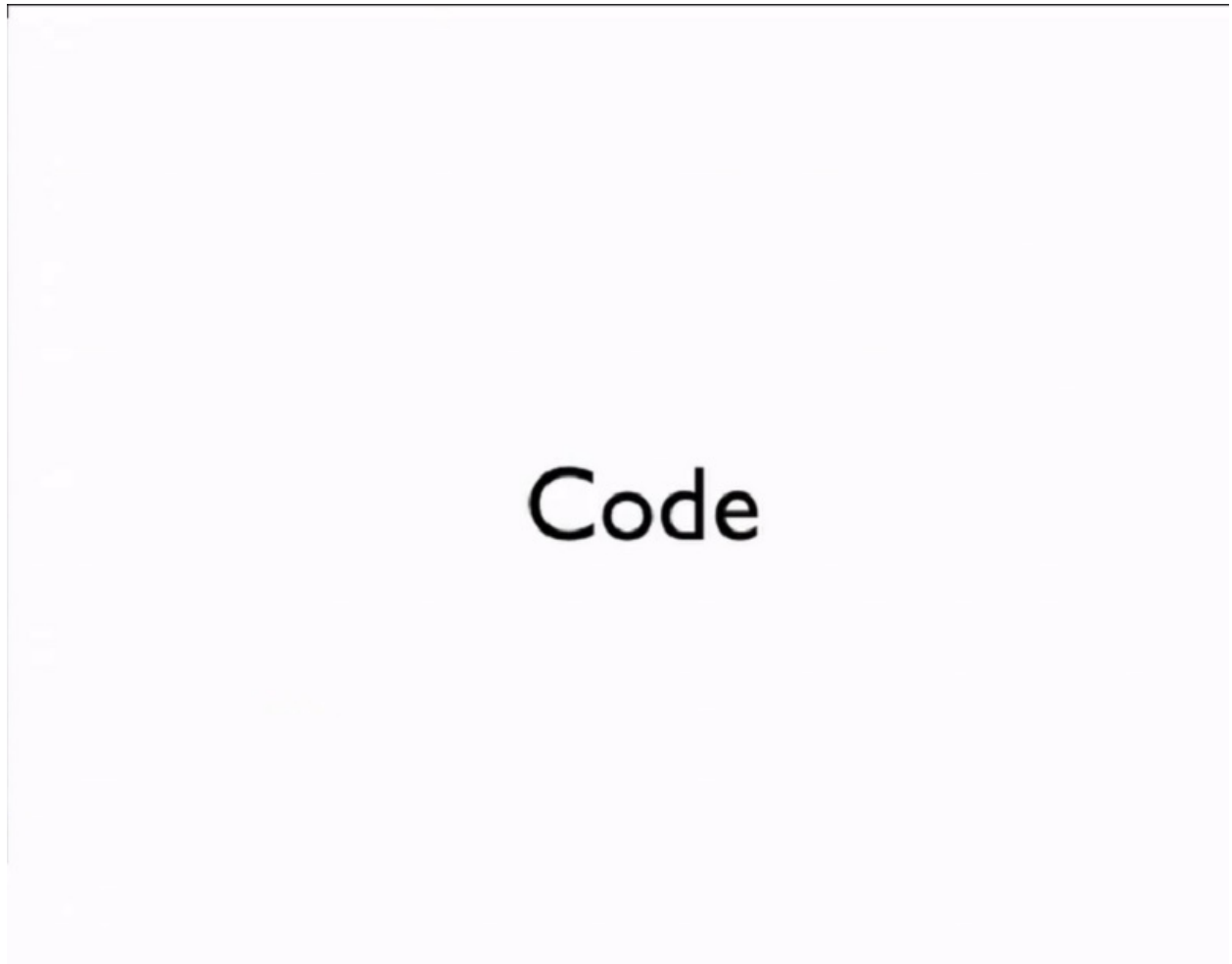
Figure 18: 00.28.50 transduce
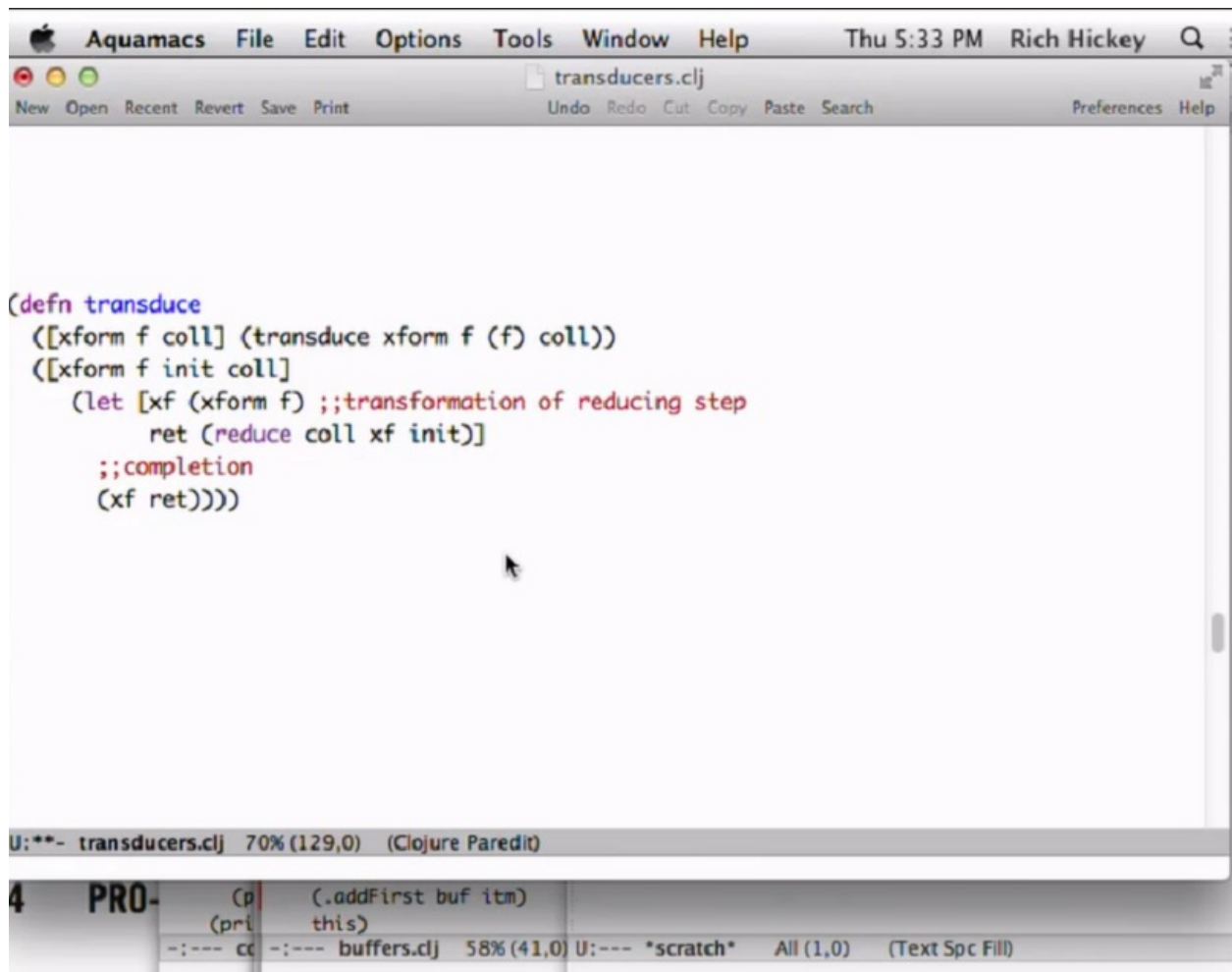
Figure 19: 00.29.10 Code

Figure 20: 00.29.13 screenshot

This is just what I said verbally before. First take the function you're supposed to be using. In the case of transduce, f is the step function. It's the reducing function. It's just like reduce. Somebody gave you a step to use. So you just transform it. You haven't told anybody else about this.

XF is now a private transducer stack that's been constructed around this step function. Then transduce just pawns off the job to reduce to do this work and passes the initial value in the collection. It gets a return value, but this is another piece that's different. It must call completion. Reduce does have a notion of being done. This collection is going to become exhausted. Therefore, it should call completion, so one last call.

This does imply, however, that the step function support arity-1, which again not all existing step functions do.

transduce

- basic transformation of reducing fn

- reduce, but adds completion call at end

- implies reducing fn must support arity-1

  - completing - helper that adds it

Figure 21: 00.32.13 transduce

We have a helping function called completing, which will take any function that's just a plain reducing function, so it only has the 2-arity: result, input, returning result. And it will add arity-1 with an implementation that's just identity. Given this result, return the same result. That's a way to sort of take an existing function that's not ready to have complete called on it and make it possible to do that.

All right.

So let me just show you one more thing here, which is the real implementation of transduce is a little bit more involved, but I wanted to just introduce this because this is another piece of novelty in Clojure that people are wondering about what is this reduce init thing.

IReduceInit

- IReduce does too much (no init)
- IReduceInit still does (eats reduced)
  eventually a variant that flows reducing out
  but would be breaking to change IReduce
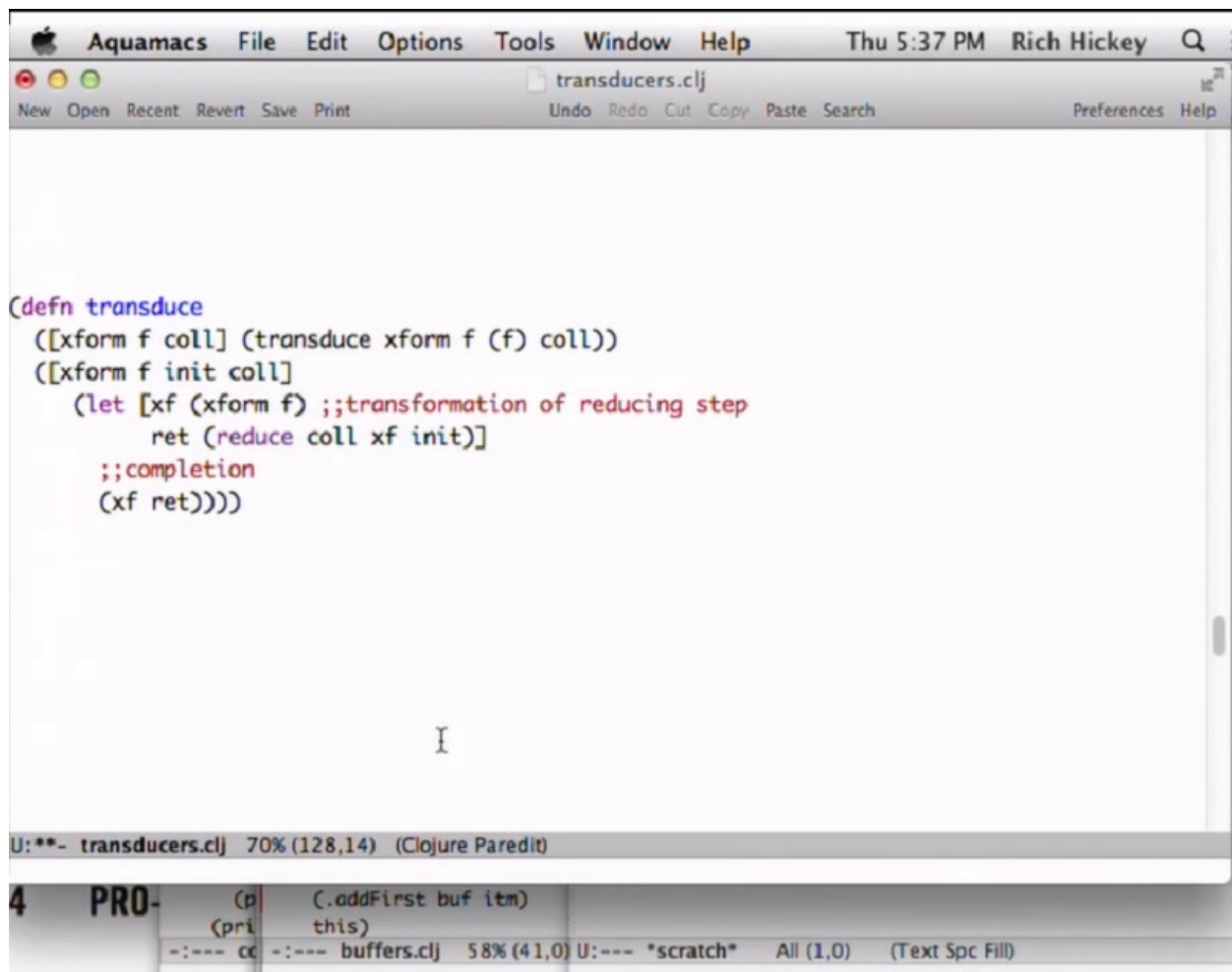
Figure 22: 00.32.43 IReduceInit

Figure 23: 00.32.44 screenshot

How many people know about IReduce? Not too many. IReduce is an interface that supports the semantics of common list reduce. It has two methods: one that's called reduce and it takes no init value, and the other is reduce that takes an init value. It's a way for a collection to say, "I know how to reduce myself. Instead of asking me for a seek to walk through my stuff, just let me do it because I have a faster way to run through all my own stuff."

Internal reduce is sort of the Java interface version of call reduce. It already existed, and other people implemented it, but most of the existing implementation of reduce is done by call reduce, which has the same exact semantics. But it ends up that it would be nice to have an interface that just represented the second part of IReduce. Remember, I said common list, who knows what the no init value thing does in reduce - nobody. And so to force everyone to keep implementing both where the first one has these tricky semantics about no values and one value was tough. IReduceInit only has the version of reduce that takes an initial value, and it's implemented by collections. It's going to come up later because we're going to see educe use it. All that's happening inside transduce is, it says I'm trying to take advantage of the fastest possible way to reduce you, which is what reduce does now as well.

IReduceInit

- IReduce does too much (no init)
- IReduceInit still does (eats reduced)

  eventually a variant that flows reducing out

  but would be breaking to change IReduce

Figure 24: 00.34.44 IReduceInit

All right, so IReduce did too much, right? It has the no init flavor, which I really don't like, but I can't just change it because people have implemented it and they have code that depends on it, so I can't change its semantics. So all I did was split IReduce into two halves. IReduceInit takes an init and IReduce derives from it and adds the no init flavor.

But IReduceInit still does something I wish it didn't, which is that it eats reduce. In other words, if there's a reduce returned by the function internally, it eats it. It says, "Okay, I should stop," and it stops, but it returns the final result. It ends up that a lot of times if you're using reductions in nested contexts, whether you're processing a tree or you're doing transduction, which is sort of a tree of or at least a list of nested reductions, you might want to a reduction in an internal step. You need to know if that bailed out so that you could bail out.

So I'd like a variant that actually returns reduced, but we haven't had time to integrate that yet. So you see IReduceInit, it's just an interface that implements the reduce version that takes an init.

sequence
(LazyTransformer)

- (sequence xform coll)

- builds linked list (thus caches)

- transformed step is list append

- tricky bit is that a transducer might not yield any output for a particular input

  - thus to satisfy pull, must loop

Figure 25: 00.36.00 sequence

All right, another transducing context is sequence. We always had sequence. I don't know if anybody ever used it. It basically was just a way to explicitly say I want to get a sequence out of this collection. Now it takes a transformer and it has become substantially more interesting because the collection might not be lazy, but the return value from sequence is, and its use of the transducer is also sort of lazy. And we'll see what sort of means in a second.

What is this step of sequence? Essentially it's building a linked list. The transform step is sort of list append. We don't really use list append, and usually list append is really expensive, so it must be doing some fancy modifying version that's efficient of list append that actually can add to the end of a list, and that is what it does, and that's what lazy seek did as well. Lazy seek just knew how to attach things to the end of the last cell.

The tricky bit, if you think about making a lazy list out of a transduced collection is the fact that that collection might sometimes return nothing. Right? Maybe one of the transducers is filter, which means I

pulled, right, because I'm going to be eating a sequence by pulling. I'm pulling. I say give me something.

If you just consumed one input from the collection and it filtered it out, what would you hand to the user? You've got nothing yet. You actually have to keep going until you see something. Inside, if you look at the implementation of lazy transformer, which is tricky, you'll see that there has to be a loop there because you have to keep consuming input until you can produce output.

Lazy vs Pushy

- Transducers are fundamentally push

- sequence implements pull via introducing inputs

  - lazy in input consumption, not output production

- thus sequence not as lazy as lazy-seq

  but still usefully lazy in practice

Figure 26: 00.37.52 Lazy vs Pushy

The other thing that's tricky about this is it sort of changes the notion of what it means to be lazy. And I don't really want to change that. I just want you to think about another notion of lazy. Transducers are fundamentally push. You supply input and then the logic runs. They're not fundamentally pull like lazy sequences are.

Sequence, the function, implements pull by going around the other end and saying, well, you wanted something, so I'm going to take another input and feed it to the step function. Now we just said one tricky bit is you may ask for something and I put it through the step function, and the step function gives me nothing, in which case I've got to keep doing this until I have one thing to return to. But what else could this have done?

[Audience response]

It could have returned one thing each time I ask. It could return nothing. It also could return what?

[Audience response]

More than one thing. Where is that going to go? I asked for one thing. You called the step function. It gave you, like, six things. What's going to happen?

30

[Audience response]

Yeah, I'm just going to attach them to the linked list, which means that it's not quite this laziness we're used to. We're used to saying, "Don't run any little part of the calculation until I ask for one thing, and then just do exactly that much work," which ends up being a huge amount of overhead. Now we're saying you're going to ask for one thing. What's actually lazy is not your consumption, but the production. In other words, we're lazily going to add only as much to this to produce an output to give you, but each step might produce more than one thing, so it's kind of pushy.

It's lazy in the input consumption, which means if you have something that expands through an infinite sequence, it's not going to work. If you have something that, in a single step, expands to something that consumes all of your memory, that's also not going to work. In particular, it's not going to work because, even if your result produces something lazy, it's going to be eagerly consumed. Nothing inside any of the transducers uses laziness at all because we want to be able to have these other semantics.

That's something to be aware of. A lot of people initially trip up over, you know, trying to use something lazy in a step and are surprised that it got fully realized. But that's the nature of this.

I still think this is a useful granularity for laziness. I don't think people are consuming all their memory in a single step. Otherwise, this gives you the best of both worlds because you get very high performance and only as much caching as you need per step, and it still is sort of windowed, so I think it would work for most things, but it's something to be aware of.

Figure 27: 00.40.34 educe

Okay. I don't think I showed any – yeah, I don't show you the code for that.

All right, the other thing that's new in here is educe. This is just basically I saw this word and I'm like, I have to figure out how to make a language feature out of this.

[Audience laughter]

That's not actually true. There were many, many different names. It was a thing, and then there was the name, not the other way, but it is. Educe, it's so – this is so cool, right?

Educe means lead out. We had reduce, which is to lead back, like, towards the thing you're making. We have transduce, which is to lead across this transformation. We have educe, which is just to sort of lead out.

To where? We're like, where is it going? It doesn't say. That's what's really cool. It's actually sort of a word game too because it's like, if you've ever seen the definition of reduce, it says it takes R and I, and it returns R. It's like reduce without the R. We don't have the – we don't know what we're going to do with this yet. We don't have the result. We're not there yet. We just take the R off. We still end up with an English word, and we're good.

[Audience laughter]

When you educe a collection, basically you're taking a transducer. You're going to make the recipe for transducing this collection, but you're not going to do anything right then. Nothing happens when you call educe except the recipe is created, and you return something that is called an eduction, or a leading out of this data.

When does it actually happen? It happens when you use it. It ends up that eduction implements a bunch of important interfaces. It implements IReduceInit. So it knows how to reduce itself, which means it knows how to transduce itself and reduce and anything that uses IReduceInit.

It is iterable, so you can use it anywhere you need something that's iterable. It's seq-able, so you can use it. You can call seq on it. Get a seq, and use it in existing code that's expecting lazy sequences. And it's sequential, which is just a tag thing because it is a sequence of things.

The most important thing about eductions and educe is that every time you ask for a seq, it'll walk through it, or every time you ask for an interator and use it, and every time you transduce or reduce it, every single time the work is going to be done across the entire collection or as much of it as you consume with the process. There's no caching like there is for seq. So, if you had side effects, you'd seem them go off over and over again. You don't have side effects. This is not really what this stuff is for.

[Audience laughter]

So the work happens with every use. And this is one of those tradeoffs. You probably don't want to hand around an eduction and have ten copies where the work is really expensive. You should probably pour that into a collection and share the results. But you may very well want to give away an eduction if it's pretty inexpensive and you don't know how much of it is going to be consumed by each user. Then it's worth doing.

Also, the nice thing is you can have an eduction as long as you're not really ready to use it. The cool thing about eductions is that if you subsequently transduce them, then the recipes fold together, and it's as if you put them, as if you composed both sets of transducers together. And there's no intermediate stuff created. And you can do that over and over again.

You can say, I know about how much of this much of the work to do. I had the source material, so I can't just give you a transducer. I know what the source was. I had the initial set of transformations. But now I'm handing it to another stage that I don't want to know about, and it doesn't want to know about me. It knows it can transduce that some more, hand it to the next guy who can transduce it some more, and finally someone can pour it into collection, reduce it, or something like that. All those intermediate operations are going to fold together, and there will be no intermediate data created.

It's just all work.
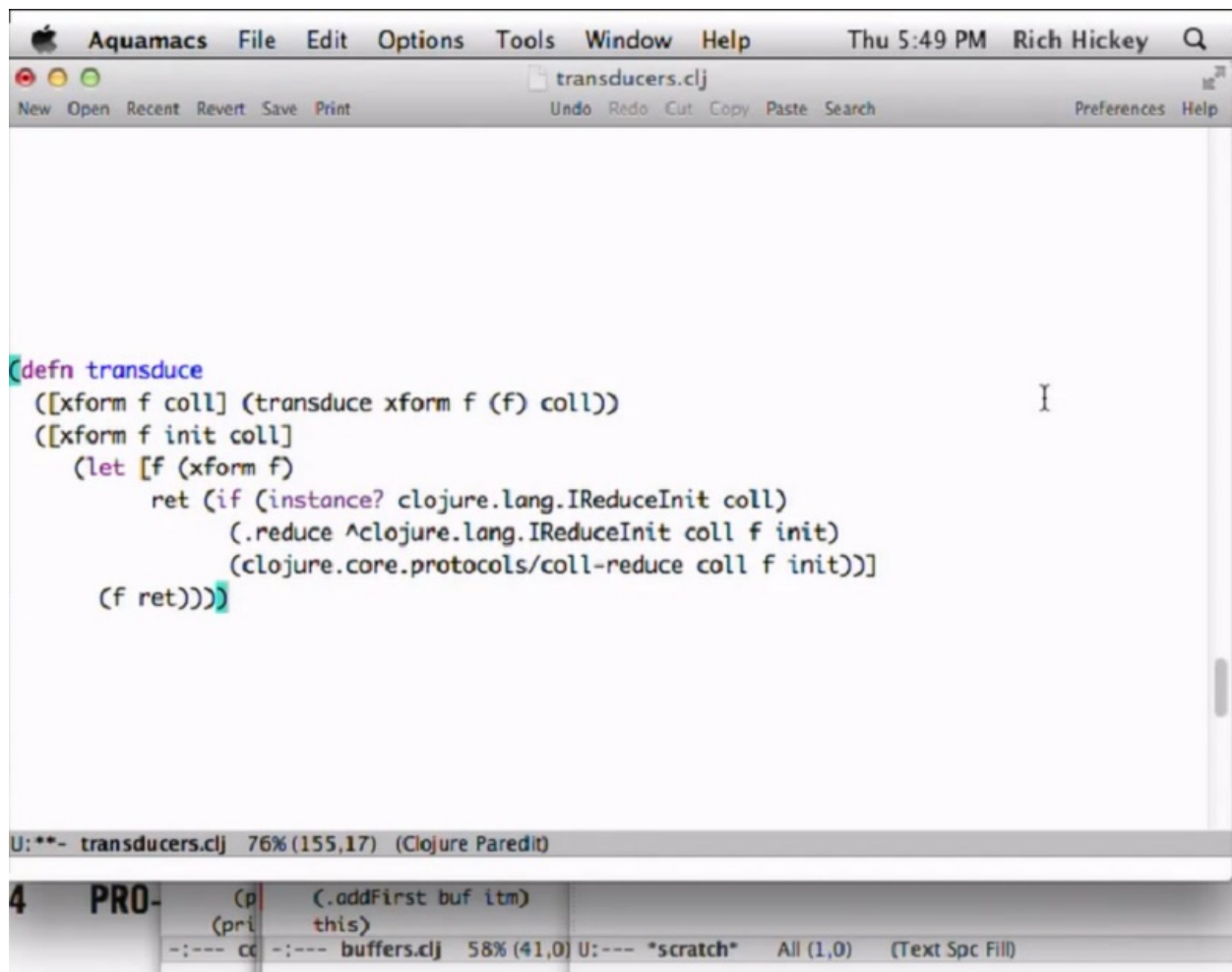
So, we can look at this.
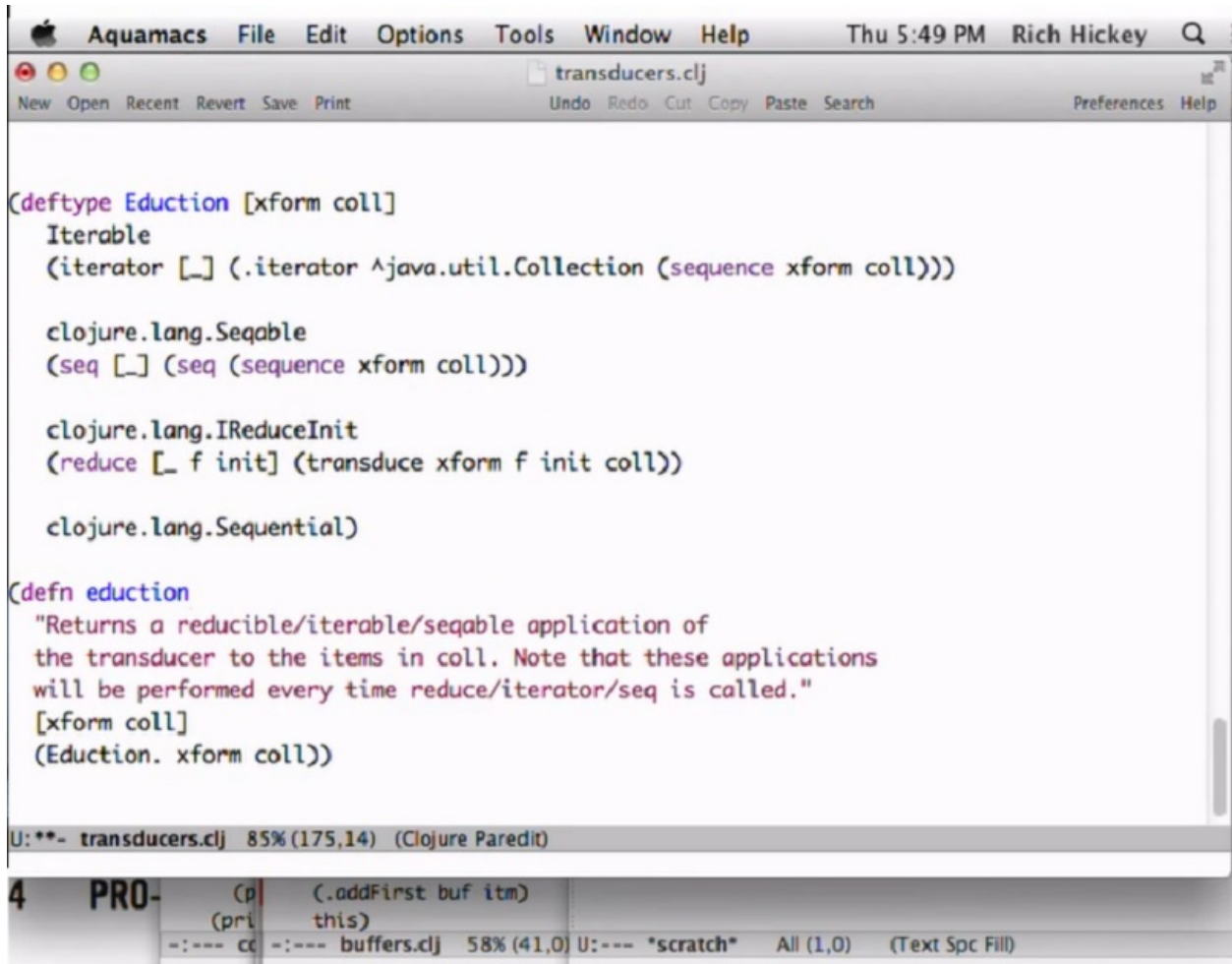
Figure 28: 00.44.42 Code

Figure 29: 00.44.43 screenshot

Figure 30: 00.44.45 screenshot - build slide

That's what an educt does. This is all of it right here. Eduction is just a deftype. It implements iterable. Now the key thing here is it just uses sequence to do that. It implements Seqable. It also used sequence to do that. It implements IReduceInit. It used transduce to do that.

What is this … weird?

Oh, yeah, it uses transduce to do that, right? And it's a tag. Sequential is just a tagging interface, so you see there's nothing inside this.

But other tricky aspects of this is it's not a collection. Unfortunately, Java.util.collection is this giant interface and satisfying it doesn't make sense for this class, but it's kind of unfortunate because Java built a lot of things saying, for instance, all collection constructors take other collections, so they know how to initialize themselves from another collection. It would have been vastly better had they said, "I take an iterable," because that's all they really need to know. So that's a little bit tricky, but we are enhancing, say, vec, so that it will be able to quickly and efficiently with both. Using reduce will build up a value quickly given an eduction. Eduction just makes one of these things, so nothing to it.

Figure 31: 00.46.12 channels

All right, and finally, we have channels as the last example of transducible contexts. This is really the whole point. A lot of what I've talked about already are variants with some performance enhancements, possibly, over what you can do with sequences and pull. But the fact that you can use transducers in those contexts and in completely push contexts like channels, that's the Holy Grail of this thing. That's what this is all about, the fact that you can make a transducer stack and pass it to reduce the most on-demand, do it now,

eager thing you can think of, and also pass it to a channel, which doesn't even have any stuff to do until later when some asynchronous thing comes by with some input.

These are the opposite parts of programming. They're opposite. But transducers connect the two. You don't need a different transducer. You can use the same transducer.

This is the point of transducers is the fact that they cross across this. So if we look at transducing a channel, we need to think about channels as if they were a step function. And it ends up in the bottom of channels is this function that looks like this. It takes a buffer and an input and returns a buffer. It's just a matter of sort of looking at that function and saying, "That's a reducing function." Reduce wasn't called inside channels because it doesn't have all the stuff. It's getting one thing at a time, but there is this step, and it's called sequentially. It follows all the rules I talked about before, so it's perfect.

What happens in a channel is this buffer add function gets transformed by the transducer. The one trick of this is the same one we saw earlier with sequence. Buffers have a notion of being full. When you're one away from being full, are you full?

[Audience response]

No. Something is going to ask the buffer, "Are you full?" It's going to say, "No, I've got room." It used to be it was saying, "I'm not full," and expecting one more thing to get ended and be asked again, "Are you full now?"

Well, what happens in a transduction? Well, you're going to get one more input, right? You're going to let one pending write occur. That's going to go through a transducer and what could come out?

[Audience response]

More than one thing. And it ends up that buffers, now the semantics are, if you're not yet full, you're going to accept one or more things. And all the buffers have been modified to do that. It ends up that that is, again, sort of a nice fit. If you're going to consume all of memory, it won't work. But otherwise, the notion – as soon as you've done that with one step, it's going to report full until it drains all the way down to what its target was.

You can't use fixed sized buffers, but, on the other hand, I think you can implement the logic of saying the activity of this buffer is gated by its size. It's just not limited to that size at any one point in time. So we've added the semantic to all of those.

All right, there are many places we want to take this. Certainly I'm hoping to get to the point where we're going to take the parallelism that we built for reducers and sort of bring it into this model, so it takes transducers. Should be very straightforward to do. You saw a lot of repetitiveness in the implementations. You know, a lot of them dummy out: 0-arg, 1-arg, variance. If you look at the implementation of reducers, there was a macro that made that go away. I'll probably implement that so other implementors will sort of be able to focus on just what's unique about their transducer.

We have ideas about supporting primitives. There are open questions about multiple arity. Map, in fact, does support multiple arity, but not all the other ones do, and it doesn't really make sense for a lot of the others, but it might for filter.

Reducers had a notion of being able to fold maps using a key value function as opposed to a function of key and value. That is very efficient. It would be nice to find a way to talk about that and make it explicit.

Maybe we can talk about the last bit.

All right, I'm running out of time, but I just want to take you quickly through some of the new things that are coming into core.async. They're in these categories: better integration with code that already exists, using channels as promises, making it easier and better to use channels for, like, RPC kind of endpoints.

And just all this stuff is in progress. So if you're following along with the alphas, you're seeing some of this stuff happen.

The first thing is that channels are going to become deref-able. This is very useful, again, for existing code. How many people have code where the contract of the code is, "Give me something deref-able"? And you're

More...

- parallel transduce

- supportive implementation macros

- primitives?

- multi-arity?

- kv-transduce?

- functional stateful transducers?

Figure 32: 00.49.20 More...

Figure 33: 00.50.20 more.async

Channels become deref-able

- Integrates with existing code

- deref reads from channel

  will block until available

- timeout flavor of deref also supported

  will read *or* timeout (not both)

Figure 34: 00.50.40 Channels

kind of independent then of whether or not that thing is actually one of the reference types or a promise. There are lots of things that are deref-able. People have made new deref-able things.

This would make it so that a channel is a valid thing to be deref-able. That will have blocking semantics, so deref will be blocking. It will be like promises that way. And when you deref, it will read from the channel. But, because they're blocking, these channels will also support the timeout flavor of deref. Everybody know that deref can take a timeout and evaluate a return that the timeout occurred? Yes?

[Audience response]

No? Yeah? Okay. Well, deref can take a timeout, and it's useful when the thing that you're derefing can block. And so the cool thing about channels when you deref them with the timeout is that they will either successfully read and return that value, or time out, but not both, so they will not time out and then later have a side effect of reading because of some pending read. The read will be killed. You'll either get the time out or the read value, but not both. I think this helps integrate to existing code and has useful semantics.

Promise channels

- create with promise-chan

- write once, read many

- buffer of one

  always write-ready

- first value written 'wins'

  subsequent dropped

- all readers unblocked by first write

  always readable once written

Figure 35: 00.52.05 Promise channel

The other thing that's coming are promise channels. Promise is cool, but channels are cooler because they support alt and things like that. And, of course, you can think of a channel as if it was a promise. But there's some more you'd want to see to make sure that using a channel as a promise wasn't a pile of convention that you had to get right over and over again.

We're going to have a function called promise-chan, which will return a promise channel. It's a write-once read-many channel. As soon as somebody has written a value to it, that value will be returned over and over

and over again, permanently. So it has a buffer-1 already by default, so you can't get that wrong. One of the things people don't want is for you to have a promise that blocks them when they fill it. Right? That's no fun, so we're not going to do that. So that will always be built in.

The first value that gets written, 'wins,' that's the same as promises. Subsequent ones will just get dropped. And the other thing that's really neat about it, compared to using channels raw, is the fact that all of the readers will unblock together.

Mult-channels, like the ones we have, all their readers are racing for the next value, and that's not what you want in a promise. You want everybody who is awaiting that value to succeed, not just the first one or the lucky one to succeed. That's a new semantic. It's not actually a different semantic of the channel; it's the semantic of the buffer that's used to create it, which is kind of neat. In other words, we implement the promise channels just by making a new kind of buffer, so that's coming.

Figure 36: 00.53.42 Endpoints

If you think a little bit about that use case of being a supplier to a promise-chan, you get into the third category I wanted to talk about today, which is just being a consumer of channels in a more RPC-like way or a more API-like way. I think there are sort of two fundamental ways to use channels. One is you're building a flow network in your program, so channels are the interface points of subsystem boundaries, and you say, you know, this subsystem has an input channel, and that's where it gets its stuff from as opposed to calling a function on it. And it uses one or more output channels to distribute its results as opposed to sort of somebody having called the function and getting the results. That decoupling is import for building

systems. That's why we like channels, and you're going to compose a system out of channels orchestrating out of processes, orchestrating the channels to wire them together. But you end up with a stable network of relationships that are used to flow data.

But if you think about something like a promise-chan or using a channel for RPC, that's not that kind of enduring relationship. You can use channels for these things, but the semantics get a little bit trickier. In particular, that thing I just talked about for promise channels exists for RPC-like uses of channels.

How many people are using channels as a way for an asynchronous function to return a value to its caller?

[Audience response]

Yeah. Okay. Look, it's already happening. All right, now how many people are, like, concerned when they put something on that channel that it might block them if the person who made the channel messed up and made an unbuffered channel and hasn't bothered to come back and read it?

[Audience response]

I know I worry about that. And you don't have a great way to communicate about the semantics except, again, by a convention to say, "You better give me a buffered channel or dropping channel because I don't want you to block me." So we're going to have two new functions. The most important one is offer!. What offer! does is it says, "If there's room for this in the channel, that's going to get written to the channel. If there's not, it's just going to return nil. Didn't do it. No. Did not happen.

As an API author, as an author of somebody who is using RPC with channels or filling promises, you're going to tell the user, "I'm going to offer you a value." Now that's a way to tell them quite clearly they better have room for it, or they better be willing to have it get dropped because now you cannot get blocked. If you call offer, it will not block, which means you can use it in go blocks. It won't block.

This is definitely a much better semantic for those kinds of scenarios. You can say, "I'll offer you values," and that's what your documentation should say. This function will offer the value to the channel you provide. That's a lot cleaner.

There's a corresponding logic for poll. This is a lot less justified except for sort of completeness and balance. Here you're saying, "You better have given me a channel with stuff in it already because I'm going to poll it," and either there'll be stuff there or it's just going to give me nil.

Now, how many people think polling is good?

[Audience response]

Nobody. Right? That's why it's called poll. If you put this in a loop, you're polling, and everyone will know because you just said, "Poll." I'm polling. So I think this should be pretty rare, but I wouldn't say that there's no good use for it, but there's no good use for polling, so neither of these block.

I am not going to have enough time to talk about pipeline except to say pipeline is awesome.

[Audience laughter]

But one thing – well, first of all, transducers and channels are awesome, but you should be aware of the fact that a transducer in a channel runs in the most protected part of the channel code, which is in the part that transfers values in and out of the buffer, which is the part that runs under the lock of the channel, which means that you will impede other activity on that channel for the duration of a transducer, which means we're back to the old, "Don't do too much work in your channel transducer or in your callback function." Now it's in your channel transducer. All right, because of these two things.

There's a new set of three functions called pipeline that I think encapsulate a whole bunch of what is pretty hard work to get right, involved with pipelining. Just so everybody is aware, what a pipeline does is it takes a bunch of sequential work. This is usually a stream of inputs and then a job you want to do. It's going to turn those jobs into a set of parallelized work, and then return the results in order out through a serial process.

pipeline

- "don't do too much work in your callback function" is back:

  "... in your channel transducer"

  - runs under the channel lock

  - inhibits other channel ops

- use pipeline instead

Figure 37: 00.57.20 pipeline

pipeline

- Moves items from one channel to another

- subject to a transformation

- with user-specified parallelism (first arg)

- always in-order results

Figure 38: 00.58.07 pipeline - build slide

You can think of P-map sort of does this, but P-map is sequenced sequence. And so pipeline is channel to channel, subject to a transformation using a transducer. And you're also able to control the parallelisms. You can say, "Use this many threads or that many threads and do this many at the same time." Unlike P-map, you have a lot of control. It takes a channel, returns a channel, takes a transducer, but it's parallel in the middle.

There are three flavors. I am going to talk about this.

Figure 39: 00.59.03 pipeline - build slide

I'm going to go five minutes late. There are three flavors. There's pipeline itself, which is for computational purposes - exactly what the transducer was for. Don't block in this thing. Don't call an async function of this thing. You're just here to calculate. It's going to use the right threads internally to do that.

There's pipeline blocking. This says, "In the middle of the step transformation I want to do, I need to block." Well, this would be catastrophic inside the transducer in a channel. Right? You're blocking in the middle of a channel, and now the channel is, like, blocked, so pipeline is a better place to do that.

Then there's pipeline async. Maybe you want to take everything that comes through this channel and make a Web call. You don't know when it's going to come back and then proceed out the other side.

How many people have written pipelining themselves manually?

[Audience response]

Yeah. How many people got it right the first time?

[Audience laughter]

Yeah, and how many people find them doing it over and over again, right, because it's a hard thing to sort of encapsulate. So I think channels give us the ability to sort of capture a pretty high level pattern as a reified programming construct, so we have pipeline async as well.

Pipeline and pipeline blocking both take a transducer, so it takes N, which is the amount of parallelism, the to-channel (Where is this stuff going to go?), the transducer, and the from-channel (Where is this stuff coming from?). Pipeline blocking does the same thing.

pipeline parallelism

- uses appropriate threads for nature of job compute vs blocking vs async

- because pipeline is parallel, transducers are applied per-item, not across items

- so can't pipeline stateful transducers, e.g. partition-by

- can still be filtering or expansive

Figure 40: 01.00.31 pipeline parallelism

Something to be aware of is the fact that because pipeline is parallelized, right? Well, first of all, it's going to do the right thing. For async stuff, it's going to use go blocks and, for compute stuff, it can use – for blocking stuff, it'll use thread, so it will do the right thing.

But because it's parallel, you can't do any stateful transducers, right? Each of these things, you know, you took work like this and turned it into that briefly and then turned – which way did it come in? This way, then that, and back out, right? It keeps the order.

But, in the middle, it's parallel, so no stateful transducers. If you look at the doc strings for transducers, for all the things that return transducers, they all say stateful when they return stateful. You can't use them in parallel contexts. You can't use them in pipeline. You're not going to be able to use them effectively with the parallelism versions of, you know, when we move reducers into a transducer compatible format.

Of course, right? The answer – you know, you should think, "Of course not," right? That's not a problem,

right? They have utility in one context. They don't in another. All right, you know. That's okay. There should still be round holes even though that there are square pegs.

Because of that, your transducer is a one operation at a time transducer, no state allowed. But they can still be filtering or be expansive. We can take one thing and return more than one thing. We can take one thing and ignore it. What we can't do is accumulate any information across, no state.

pipeline-async

- pipeline-async takes channel filling fn

  (pipeline-async [n to-ch af from-ch])

  af is [input result-ch]->nil

- af should return immediately, having launched async work that fills result-ch and closes it

Figure 41: 01.02.04 pipeline-async

All right, and pipeline async, I really won't have time to go into, but it's a little bit different. It does not take a transducer because it needs a way to get an asynchronous result and it uses channels to do that. It will soon tell you it's going to offer your results to the channel, but basically it's going to give you your input and a place to put, a channel to put the result. Then it collapses all that stuff and effectively streams it out the other side. Between the three of these, that should cover all your pipelining needs.

To wrap up, how many people are using transducers?

[Audience response]

All right, how many people are using core.async?

[Audience response]

Yeah, so I think both of these have seen a lot of community uptake early on, especially since transducers have still just been in the alphas. And a lot of great feedback, so please keep it coming. It's really great.

Figure 42: 01.02.33 Thanks

There's a lot more stuff coming for 1.7. I think Alex is going to run a thing on feature expressions later, which is probably the next other big thing, so look forward to that and other stuff, and rock on.

Thank you.

[Audience applause]