Transducers

- **Speaker: Rich Hickey**
- **Conference: Strange Loop 2014 - Sept 2014**
- **Video: https://www.youtube.com/watch?v=6mTbuzafcII**

[Time 0:00:00]

```
slide:

[Clojure logo]

Transducers

Rich Hickey
```

Transducers.

[Time 0:00:03]

```
slide title: What are They?

[ Photo of burrito. ]
```

Figure 1: 0:03 What are They?

[Audience laughter]

Of course, everything is just some combination of the same ingredients. The shell is on the outside, the cheese is in, it is on the top, whatever. I am not claiming any novelty here. It is just another

rearrangement of the same old stuff as usual. But, you know, sometimes, the cheese on top tastes better than when it is inside.

[Time 0:00:28]

```
slide title: What are They?

+ extract the _essence_ of map, filter et al
+ away from the functions that transform sequences / collections
+ so they can be used elsewhere
+ recasting them as _process transformations_
```

All right, so what are transducers? The basic idea is to go and look again at map and filter, and see if there is some idea inside of them that could be made more reusable than map and filter, because we see map and filter being implemented over and over again in different contexts. We have map and filter on collections. We have map and filter on streams. We have map and filter on observables. We were starting to write map and filter, and so on, on channels. And there was just no sharing here. There is no ability to create reusable things. So we want to take the essence out and see if we can reuse them.

And the way that we are going to do that is by recasting them as process transformations. And I will talk a lot more about that, but that is essentially the entire idea: recasting the core logic of these what were sequence processing functions as process transformations, and then providing context in which we could host those transformations.

[Time 0:01:30]

```
slide title: 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
```

So when I talk about processes, what am I saying? It is not every kind of process. There are all kinds of processes that cannot be modeled this way, but there are ton of processes that can. And the critical words here are that: if you can model your process as a succession of steps, and if you can talk about a step, or think about a step as ingesting an input, as taking in or absorbing some input, a single input. So something going on, there is an input, we are going to absorb that input into the something going on and proceed. That is the kind of process that we can use transducers on.

And when you think about it that way, building a collection is just one instance of a process with that shape. Building a collection is: you have the collection so far, you have the new input, and you incorporate the new input into the collection, and you keep going.

But that is a specialization of the idea. The general idea is the idea of a seeded left reduce. Of taking something that you are building up, and a new thing, and continually building up. But we want to get away from the idea that the reduction is about creating a particular thing, and focus more on it being a process. Some processes build particular things; other processes are infinite. They just run indefinitely.

[Time 0:02:48]

```
slide title: Why 'transducer'?

+ _reduce_
  'lead back'

+ _ingest_
  'carry into'

+ _transduce_
  'lead across'

+ on the way back / in, will carry _inputs_ across a series
  of _transformations_
```

So we made up words. Actually we did not make up a word. Again, this is *actually* a word. But why this word?

We think it is related to reduce, and reduce is already a programming word. And it is also already a regular word. And the regular word means "to lead back", to bring something back. The word has come to mean over time "to bring something down" or "to make something smaller". It does not necessarily mean that. It just means "to lead it back to some mother-ship", and in this case, we are going to say, this process that we are trying to accomplish.

The word "ingest" means "to carry something into", so it is the same kind of idea, but that is about one bite. Reduction is about a series of things, and "ingest" itself means one thing.

And "transduce" means "to lead across". And the idea basically is: as we are taking inputs into this reduction, we are going to lead them through a series of transformations. We are going to carry them across a set of functions. So we are going to be talking about manipulating input during a reduction.

[Time 0:03:55]

```
slide title: Transducers in the Real World

+ 'put the baggage on the plane'
+ _'as you do that:'_
  + break apart pallets
  + remove bags that smell like food
  + label heavy bags
```

So this is not a programming thing. This is a thing that we do all the time in the real world. We do not call them "transducers". We call them "instructions".

And so, we will talk about this scenario through the course of this talk, which is, "put the baggage on the plane". That is the overall thing that we are doing, but I have this transformation that I want you do to the baggage.

While you are putting the baggage on the plane, break apart the pallets. We are going to have pallets, big wooden things with a pile of luggage on it that is sort of shrink-wrapped. We want to break them apart, so now we have individual pieces of luggage.

We want to smell each bag and see if it smells like food. If it smells like food, we do not want to put it on the plane.

And then we want to take the bags and see if they are heavy. And we want to label them.

That is what we have to do. So we are talking to the luggage handlers, we say, “that is what you are going to do”. And they all say, “Great, I can do that”.

[Time 0:04:59]

```
slide title: Conveyances, sources, sinks are _irrelevant_

+ And _unspecified_
+ Does baggage come / go on trolleys or conveyor belts?

  _Rules don't care_
```

One of the really important things about the way that was just said, and the way you talk to luggage handlers and your kids and anybody else you need to give instructions to, is that the conveyance and the sources and the sinks of that process are irrelevant. Do the luggage handlers get the bags on a conveyor belt or on a trolley? We did not say. We do not care. In fact, we really do not want to care.

We do not want to say to the luggage guys, “Today, there is going to be luggage on a trolley, do this to it and then put it on another trolley”. And then tomorrow when we switch to conveyor belts, have them say, “We did not know what to do”. “It came on a conveyor belt and like, you did not . . . I have rules for trolleys”.

[Audience laughter]

So the rules do not care. The instructions do not care. This is the real world.

[Time 0:05:56]

```
slide title: Transformation in the Programming World

+ Collection function composition:

  (comp
    (partial map label-heavy)
    (partial filter non-food?)
    (partial mapcat unbundle-pallet))
```

Then we have programming. What do we do in programming? We have collection function composition. We are so cool. We have lists. We have functions from lists to lists. So we can compose our functions.

We are going to say, well, labeling the heavy bags is like mapping. Every bag comes through and it gets a label or does not. But for every bag that comes through, there is a bag that comes out. Maybe it has a label or it does not.

And keeping the non-food bags is a filter. It is analogous to filter. Is it food? We do not want it. If it is not food, we are going to keep it. So we may or may not have an input, depending on this predicate.

And unbundling the pallets is like mapcat. There is some function that given a pallet, gives you a whole bunch of individual pieces of luggage.

So we already know how to do this. We are done. We are finished. Programming can model the real world.

Except there is a big difference between this, and what I just described happens in the real world.

[Time 0:06:56]

```
slide title: Conveyances are _everywhere_

+ map, filter, mapcat are functions of sequence -> sequence
+ 'rules' only work on sequences
+ creates sequences between steps
```

Because "map" is a function from whatever, collection to collection, or sequence to sequence. Pick your programming language. But it is basically a function of aggregate to aggregate. And so is "filter" and so is "mapcat".

And the rules that we have only work on those things. They are not independent of those things. When we have something new, like a channel or stream or observable, none of the rules we have apply to that.

[Time 0:07:24]

```
slide:

[ Photo of airport luggage holders, repeated four times, with big
arrows from one to the next in a sequence. ]
```

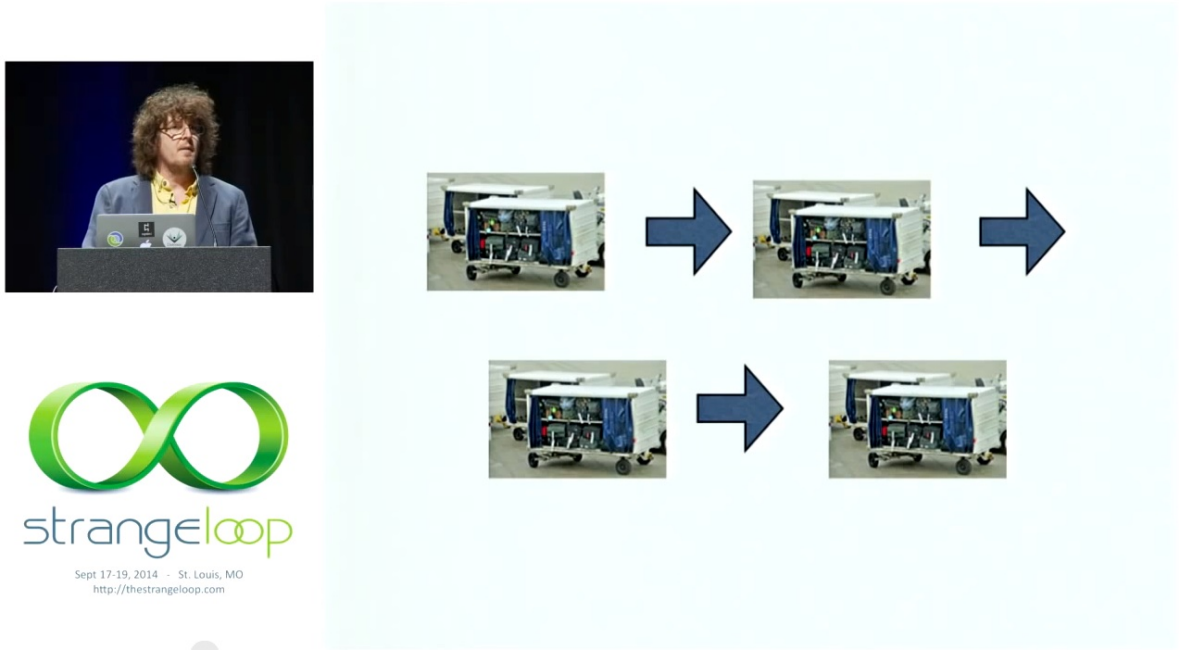

Figure 2: 07:24 {slide of trolleys}

And in addition, we have all this in-between stuff. It is as if we said to the luggage guys, "Take everything off the trolley, and unbundle the pallet, and put it on another trolley. And then take it off that trolley. And then see if it smells like food. And if it does not, put it on another trolley. And then take it off that trolley, and if it is heavy, put a label on it and put it on another trolley". This is what we are doing in programming. This is what we do all the time.

[Time 0:07:50]

```
slide:

[ Same slide, but now with a big red circle with a red slash through
it meaning "no". ]
```

And we wait for a sufficiently smart supervisor to come and say, “What are you guys doing?”.

[Audience laughter]

So we do not want to do this anymore.

[Time 0:08:02]

```
slide title: No reuse

+ Every new collection / process defines its own versions of map, filter,
  mapcat et al
  + MyCollection -> MyCollection
  + Stream -> Stream
  + Channel -> Channel
  + Observable -> Observable ...
+ Composed algorithms are needlessly _specific and inefficient_
```

We do not have any reuse. Every time we do this, we end up writing a new thing. You write a new kind of stream, we have a new set of these functions. You invent ... Rx. Boom! There is a 100 functions.

We were starting to do this in Clojure. We had channels, and we are starting to write map and filter again. So it is time to say, time out. Can we do this?

Because there are two things that happened. One is, all the things we are doing are specific. And the other is, there is a potential inefficiency here. Maybe there are sufficiently smart compilers, and maybe for some context, they can make the intermediate stuff go away. Maybe they cannot.

The problem is: our initial statement really does not like what we normally do. It is not general. It is specific. We are relying on something else to fix it.

[Time 0:08:54]

```
slide:

[ Photo of airport luggage holder, then a right arrow to a photo of a
different kind of cart for carrying luggage. ]
```

Then we also have this problem when we are going to go from one kind of conveyance to another. And now all of a sudden “map” is from x to x. How do we fix this? And I know what everybody is thinking, of course.

[Time 0:09:06]

```
slide:

[ Same photos as previous slide. ]

def map[B, That](f: A => B)(implicit bf:
CanBuildFrom[Repr, B, That]): That
```

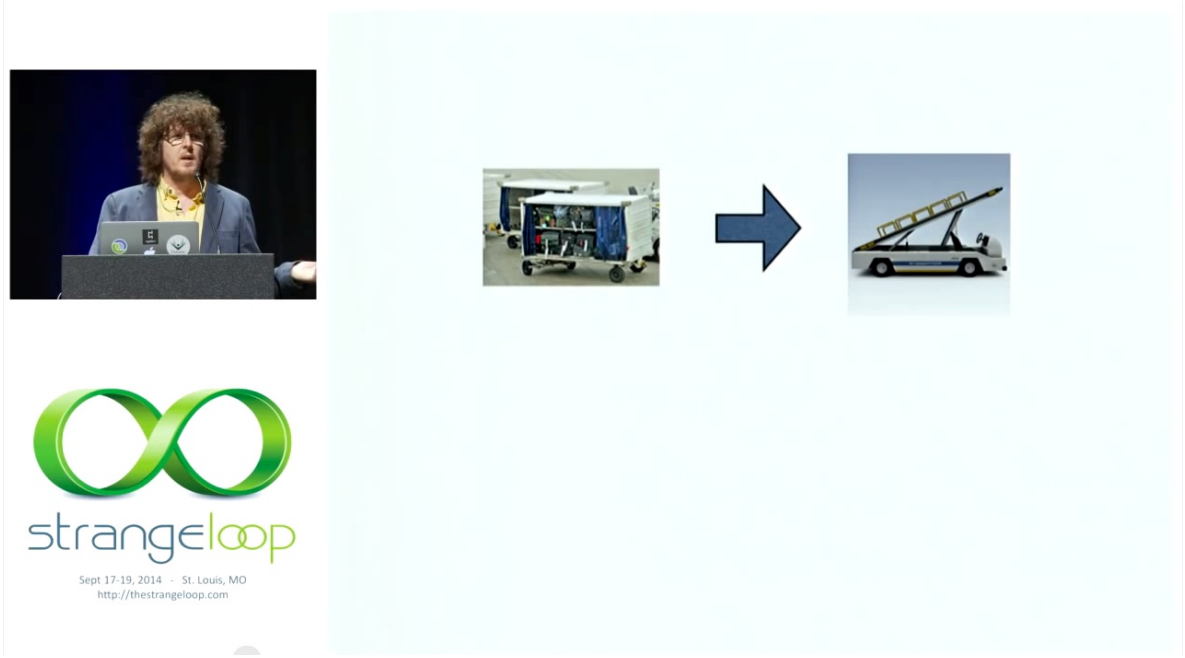

Figure 3: 08:54 {trolley to stair car}

[Audience laughter, then applause]

Yeah.

[Time 0:09:19]

```
slide:

[ Same photo and code snippet as previous slide, with big red circle
and a slash across it meaning "no".  Also a new line of text at the
bottom. ]

Stop talking about entire job
```

So that may fix some of this, but in general it does not solve the problem. And the problem is mostly about the fact that we are talking about the entire job. Those instructions, they were about the step. They were not about the entire job. The entire job was around it. While are you doing this thing, here is what you going to do to the inputs. Here is how you are going to transform them while you are doing the bigger thing, which could change. We could change from conveyor belts to trolleys and stuff like that.

So we want to just take a different approach. If we have something that is about the steps, we can build things that are about the whole jobs, but not vice-versa.

[Time 0:09:56]

```
slide title: Creating Transducers

(def process-bags
```

```
  (comp
    (mapcatting unbundle-pallet)
    (filtering non-food?)
    (mapping label-heavy)))

+ mapcatting, filtering, mapping _return_ _transducers_
+ process-bags _is a_ _transducer_
+ transducers _modify_ a process by transforming its reducing function
```

OK, this is just going to be some usages here, and then I will explain the details in a little bit. Because usually I do it the opposite way, and people are like, "Oh, my brain hurt for so long and, like, 40 minutes in, you showed me the thing that made it all valuable".

So, here is the value proposition. We make transducers like this. We say: I want to make a transducer. I want to make a set of instructions. I am going to call it process-bags.

I am going to compose the idea of mapcatting using unbundled-pallet as the function. So I want to unbundle the pallets.

Then I want to keep the non-food, filter out the food.

And I want to map, labeling the heavy bags.

And we are going to compose those functions with "comp", which is Clojure's ordinary function composition thing.

So mapcatting, filtering, and mapping return transducers. And process-bags, which is the composition of those things, is itself a transducer. So we are going to call mapcatting, call filtering, call mapping, get 3 transducers, compose them, and make another transducer.

Each transducer takes a process step, its reducing function, and transforms it, changes it a little bit. It says, "before you do that step, do this". I will explain why that seems backwards in a little bit.

[Time 0:11:23]

```
slide title: Using Transducers

                                        Concrete reuse!
;; build a concrete collection
(into airplane process-bags pallets)

;; build a lazy sequence
(sequence process-bags pallets)

;; like reduce, but takes transducer
(transduce
  (comp process-bags (mapping weigh-bag))
  + 0 pallets)

;; a CSP channel that processes bags
(chan 1 process-bags)

;; it's an open system
(observable process-bags pallet-source)
```

Having made those instructions, we can go into completely different contexts, and reuse them.

Amongst the several contexts that we are supporting in Clojure in the first version, is supporting transducers in "into". And "into" is Clojure's function that takes a collection and another collection and pours one into the other. Instead of having more object oriented collections that know how to absorb other collections with buildFrom, we just have this standalone thing called "into", but it is the same idea. Your source and destination could be different.

So we want to pour the pallets into the airplane, but we are going to take them through this process-bags transformation first. So this is collection building. "into" is already a function in Clojure. We just added an additional arity that takes transducers.

Then we have "sequence". "sequence" takes some source of stuff, and makes a lazy sequence out of it. "sequence" now additionally takes a transducer, and will perform that transformation on all the stuff as it lazily produces results. So we can get laziness out of this.

There is a function called "transduce", which is just like "reduce" except it also takes a transducer. So that takes a transducer, an operation, an initial value, and a source, so the transducer is a modification of process-bags I will talk about in a second. The operation is sum, the initial value is 0, and the source are the pallets.

[Time 0:12:53]

So what does this composition do? What is this going to do? It is going to sum the weight of the bags. This is the weight of all the bags. So it is cool. We can take the process that we already had, and modify it a little bit. We can add "weighing the bags" at the end of that set of instructions. And that gives us a number, and we can use that number with plus to build a sum. So that is "transduce".

The other thing we can do is go to a completely different context now. So we have some channels. We are going to be sending pallets of luggage across channels. They do not really fit, but the idea is there. This is a very different context. Channels run indefinitely. You can feed them stuff all the time and get stuff out of the other end on a continuous basis.

But the critical thing here is that these things are not parameterized. They are not "I am a thing that you can tell me later ... you are going to tell me if it is trolleys or conveyor belts". This is the exact same process bags I defined here [on previous "Creating Transducers" slide], this concrete thing, being reused in a completely different context. So this is concrete reuse, not parameterization.

So we can use transducers on channels. The channel constructor now optionally takes a transducer, and it will transduce everything that flows through. It has its own internal processing step, and it is going to modify its inputs accordingly with the transducer it is given.

And it is an open system. I can imagine, but I did not get time to implement, that you could plug this into RxJava trivially. And take half of the RxJava's functions and throw them away. Because you can just a build a transducer and plug it into one observable function that takes an observable and a transducer and returns an observable. And that is the idea.

[Time 0:14:43]

```
slide title: Transducible Processes

+ into, sequence, transduce, chan etc accept transducers
+ use the transducer to transform their (internal, encapsulated)
  reducing function
+ do their job with the transformed reducing function
```

So we call all of these things – into, and sequence, and transduce, and chan - transducible processes. They satisfy that definition of process we gave before, and they accept a transducer.

So transducers sort of have two parts. You make functions that create transducers. And in contexts where they make sense, you start accepting transducers. Then you have these two orthogonal LEGOs you can put together.

Inside each process, they are going to take that transducer and their internal processing function. So what is the internal processing function of "into"? The thing that adds one thing to a collection. In Clojure, it is called "conj" for conjoin.

Similarly, inside lazy sequences, there is some func mechanism that produces a result on demand and then waits to produce the next thing. So that has a step inside of it that can be transformed this way.

Channels also take inputs. Somewhere inside channels is a little step function that adds an input to a buffer. That step function has exactly the same shape as conj, and as laziness. So it can transform its fundamental internal operation. But the operation remains completely encapsulated. The transducible context takes the transducer, modifies its own step function, and proceeds with that.

[Time 0:16:03]

```
slide title: Deriving Transducers

         Lectures on
Constructive Functional Programming

            by
          R. S. Bird

                    A tutorial on the universality and
                         expressiveness of fold

                               Graham Hutton

http://www.cs.ox.ac.uk/files/3390/PRG69.pdf
http://www.cs.nott.ac.uk/~gmh/fold.pdf
```

So as I said before, there is nothing new. Two papers I find useful for helping you think about these things are this Lectures on Constructive Functional Programming, which is a lot closer to the source of when people started thinking about folds and their relationship to lists.

And the second Graham Hutton paper is sort of a summary paper, which sort of just summarizes the current thinking at the time it was written. So they are both really good. But now I am going to take you through "how do we get to this point?". How do we think about these things?

- Lectures on Constructive Functional Programming by R. S. Bird
- A tutorial on the universality and expressiveness of fold by Graham Hutton

[Time 0:16:37]

```
slide title: Many list fns can be defined in terms of _foldr_

+ encapsulates the recursion
+ easier to reason about and transform

(defn mapr [f coll]
```

```
  (foldr (fn [x r] (cons (f x) r))
         () coll))

(defn filterr [pred coll]
  (foldr (fn [x r] (if (pred x) (cons x r) r))
         () coll))
```

So one of the fundamental things that the Bird paper and the work that preceded it talk about is the relationship between these list processing operations and fold. In fact, there is a lot of interesting mathematics that shows that they are the same thing. That you can go backwards and forwards between a concrete list and the operations that constructed it. They are sort of isomorphic to each other.

So many of the list functions that we have can be redefined in terms of fold. There has already been a definition of "map" in several talks here, I think, but the traditional definition of "map" says: if it is empty, return an empty sequence. If you are getting a new input, cons that input onto the result of mapping to the rest of the input. It is recursive and calls itself. But map does that, filter does that, mapcat does that. They all sort of have these structures.

But filter is a little bit different. It has a predicate inside. It has a conditional branch, and then it recurses in two parts of the branch with different arguments.

So what this earlier work did was say, you *can* think about all these things as folds. And if you do, you get a lot of regularity, and things that you can prove about folds, which are now all uniform, will apply to all these functions that otherwise look a little bit different from each other. So there is a lot of value to this.

"fold" encapsulates the recursion, and it is easier to reason about. So if we look at a redefinition of "map" – it is not often defined this way – but if we look at a redefinition of "map" in terms of "fold", then we say: we are going to fold this function that cons-es the first thing onto the rest. And we start with an empty list. So this is fold right. And we do that over a collection.

We can similarly define "filter" this way. And what is really interesting about these things is that the "foldr", the empty list, and the coll, that is all boilerplate. It is exactly the same. "map" and "filter" are precisely the same in those things. All that is different is what is inside the inner function definition. And even there, there is something the same.

[Time 0:18:45]

```
slide title: Similarly, via reduce (foldl)

+ returning eager, appendy vectors

(defn mapl [f coll]
  (reduce (fn [r x] (conj r (f x)))
          [] coll))

(defn filterl [pred coll]
  (reduce (fn [r x] (if (pred x) (conj r x) r))
          [] coll))

(defn mapcatl [f coll]
  (reduce (fn [r x] (reduce conj r (f x)))
          [] coll))
```

So it ends up that you can similarly define these functions in terms of "foldl". And "foldl" is just left reduce. And here are some what-if definitions of "map" and "filter", and we added "mapcat", that are left folds that use left reduce. So the trade-off between left reduce and right reduce is: right reduce sort of puts you on the laziness path, and left reduce puts you on the loop path. It ends up that the loop path is better and faster and more general for the kinds of things we want to apply this to, especially if we can get laziness later, which I just said we kind of could. So we like that.

So this means we can turn these things into loops. Because reduce becomes a loop. But the same thing. We have the boilerplate. We have reduce. These definitions use vectors, which in Clojure are like arrays, but their fundamental conj-ing operation adds at the end. So this has the same shape I want to talk about for the rest of the talk.

We have something that we are building up, a new input, and we produce a new thing, and sort of the stuff is coming from the right and getting added to the right hand side. So it just makes more sense here. So these are eager and they return vectors. But it is the same idea. We are reducing. We have a function that takes the vector so far, and a new value. We are conjoining the new value, having applied f to it.

That is the idea of mapping. There is an idea behind mapping that luggage handlers understand. Put the label on everything that comes through. It is very general. That is mapping. They get that. We get that. We are all human beings. We understand the same thing.

As programmers, we have mucked this up, because look at what is happening here. "map" says: there is this fundamental thing that you do to everything as it comes through. "filter" says: there is this fundamental tiny thing that you do to everything as it comes through. And "mapcat" says: there is this fundamental tiny thing that you do to everything as it comes through.

What is the problem?

[Time 0:20:44]

```
slide:

[ Same as previous slide, except with the three occurrences of "conj"
having red arrows pointing to them from the new text in red: "We want
to get rid of these". ]
```

conj! conj is basically like saying "to the trolley", or "to the conveyor belt". It is something about the outer job that is leaked, or it is inside the middle of the idea. Inside the middle of the idea of mapping is this conj. It does not belong. Inside the middle of the idea of filter is this conj. It should not be there. Same thing with mapcat.

This is specific stuff in the middle of a general idea. The general idea is just take stuff out. We do not want to know about conj. Maybe we want to do something different.

So again, we have a lot of boilerplate. We have these essences. And the other critical thing is: the essences can be expressed as reducing functions. Each of these little inner functions is exactly the same shape as conj. It takes the result so far, and new input; returns the next result.

[Time 0:21:41]

```
slide title: Transducers

+ modify a process by transforming its reducing function

  (defn mapping [f]
```

```
    (fn [step]
      (fn [r x] (step r (f x)))))

  (defn filtering [pred]
    (fn [step]
      (fn [r x] (if (pred x) (step r x) r))))

  (defn cat
    (fn [step]
      (fn [r x] (reduce step r x))))

  (defn mapcatting [f]
    (comp (map f) cat))
```

So to turn those inner functions into transducers, we are just going to parameterize that conj. We are going to parameterize the old fashioned way: with a function argument. We are not going to have anything higher-order blah blah blah. We are going to take an argument, which is the step.

So right in the middle body of this mapping – this is the same as it was in the last slide. This is where it said conj. Now we say step. We put that inside a function that takes the step. So this is a function.

"mapping" takes the thing that you are going to map, label the baggage. And it returns something that is a function that expects a step. What are we doing? Putting stuff on conveyor belts. What are we doing? We are putting stuff on trolleys.

And it says, before I do that, I am going to call f on the luggage. I am going to put a label on the luggage. But I do not know about luggage anymore. The step, you are going to tell me later. What are we doing today? Conveyor belts or trolleys? Conveyor belts, cool. I got the rules, I understand how to do mapping and filtering and mapcatting.

So same thing: filter. And what is beautiful about this is: what is the essence of filtering? Apply a predicate. Then maybe you do this step, or maybe you do not. There is no stuff here. It is a choice about activity. It is a choice about action.

Same thing with concatenate, cat. What does it do? It basically says: do this step more than once. I am giving you an input that is really a set of things. Do it to each thing. And mapcatting is just composing map and cat. Which it should be. OK.

[Time 0:23:27]

```
slide title: reduce-based map et al redux

(defn mapl [f coll]
  (reduce ((mapping f) conj)       <----
          [] coll))

(defn filterl [pred coll]
  (reduce ((filtering pred) conj)  <----  conj is now
          [] coll))                       an argument

(defn mapcatl [f coll]
  (reduce ((mapcatting f) conj)    <----
          [] coll))
```

So we can take these transducer returning functions. So "mapping" returns a transducer. "filtering" returns a transducer. "cat" *is* a transducer. And "mapcatting" returns a transducer.

And we can then plug them into the code we saw before. Like how could we define "map", now that we have made "mapping" into this abstract thing that does not really know about lists or vectors anymore?

And what we do is: we just call "mapping". That gives us a transducer. It says: If you give me a step function, I will modify it to do f first on the input. And we say: OK, here is the step function: conj. Now I rebuilt the functions I had before, except conj is not inside mapping and filtering and mapcatting anymore. It is an argument. Woohoo!! We now have the essence of these things, a la carte. And that is the point.

[Time 0:24:19]

```
slide title: Transducers are Fully Decoupled

+ Know nothing of the process they modify
  + reducing function fully encapsulates
+ May call step 0, 1 or more times
+ Must pass previous result as next r
  otherwise _must know nothing of r_
+ can transform input arg
```

Transducers are *fully* decoupled. They do not know what they are doing. They do not know what process they are modifying. The step function is completely encapsulated.

They have some freedom; they can call the step function not at all, once exactly per input, or more than once per input. But they do not really know what it does, so that is what they are limited to doing: using it or not using it.

That is pretty much it, except they do have access to the input. So when we said mapcat unbundle-pallet, the function we are supplying there is something that knows about pallets. It does not know about conveyor belts. It does not know what the overall job is, but it knows about pallets. It is going to know how to turn a pallet into a set of pieces of luggage.

There is a critical thing about how they use that step function that they have been passed, and it goes back to that successor notion I mentioned before. They *must* pass the previous result from calling the step function, as the next first argument to the next call to the step function. That is the rule for step functions and their use. And no others. They *can* transform the input argument, the second argument.

[Time 0:25:29]

```
slide title: Backwards comp?

(comp
  (mapcatting unbundle-pallet)
  (filtering non-food?)
  (mapping label-heavy))

+ No, composing the transformers yields input transformations
  that run left->right

[ Figure with step in the middle, label-heavy surrounding it,
non-food? surrounding that, and unbundle-pallet surrounding that. ]
```

So let us talk a little bit about the backwards part, because this is a frequent question I get. What did you do? "Do transducers change comp?" is the first thing. They ruin comp, or something like that.

And so what we have to do is look at what transducers do. A transducer function takes a function, wraps it, and returns a new step function. That is still happening right to left. This is ordinary comp, and it works right to left.

So mapping gets run first. We are going to have some operation, "put stuff on trolley", or conj. Mapping will be the first thing that happens. It is going to make a little modified step that labels the heavy bags before it calls "put it on the airplane".

Then filtering gets called. It does go right to left. It says, "Give me that step. I will make you a new step that first sees if it is food. If it is food, I am going to throw it away. If it is not food, I am going to use it".

Then mapcatting runs, or the result of mapcatting runs. And that says, "Give me a step and I will take its input, presume it is a pallet, unbundle it, and supply each of those arguments to the nested thing".

So the composition of the transformers runs right to left. But it builds a transformation step that runs in the order that they appear: left to right in the comp. In other words, comp is working ordinarily. It is building steps right to left. The resulting step runs the transformations left to right. So when we actually run this, we will unbundle the pallets first, call the next step, which is to get rid of the food. Call the next step, which is to label the heavy bags. So that is why it looks backwards.

[Time 0:27:14]

```
slide title: Transducers are Fast

+ Just a stack of function calls
  + short, inlinable
+ No laziness overhead
+ No interim collections
+ No extra boxes
```

OK. So the other nice thing about transducers is that there is no intermediate stuff. They are just a stack of function calls. They are short. Potentially they could be inlined. There is no laziness overhead. There is no laziness required. There is no laziness utilized.

There is no interim collections. We are not going to have you make everything into a list. So you can say, "an empty list is nothing". No. Nothing is nothing. An empty list is an empty list. And one thing is one thing. A list of one thing is a list of one thing. These are not the same.

So you use the step function, or you do not. And there is no extra boxes required, or boxing for communicating about the mechanism.

[Time 0:27:56]

```
slide title: Transducer Types, Thus Far
```

So the other thing that was sort of interesting was ... I started talking about transducers, and a lot of people in Haskell were trying to figure out what the actual types were, because I had shorthand in my blog post.

And I am not going to get into that right now, except to say that I think it is a very interesting type problem, and I am very excited to see how people do with it in their various languages. I have seen

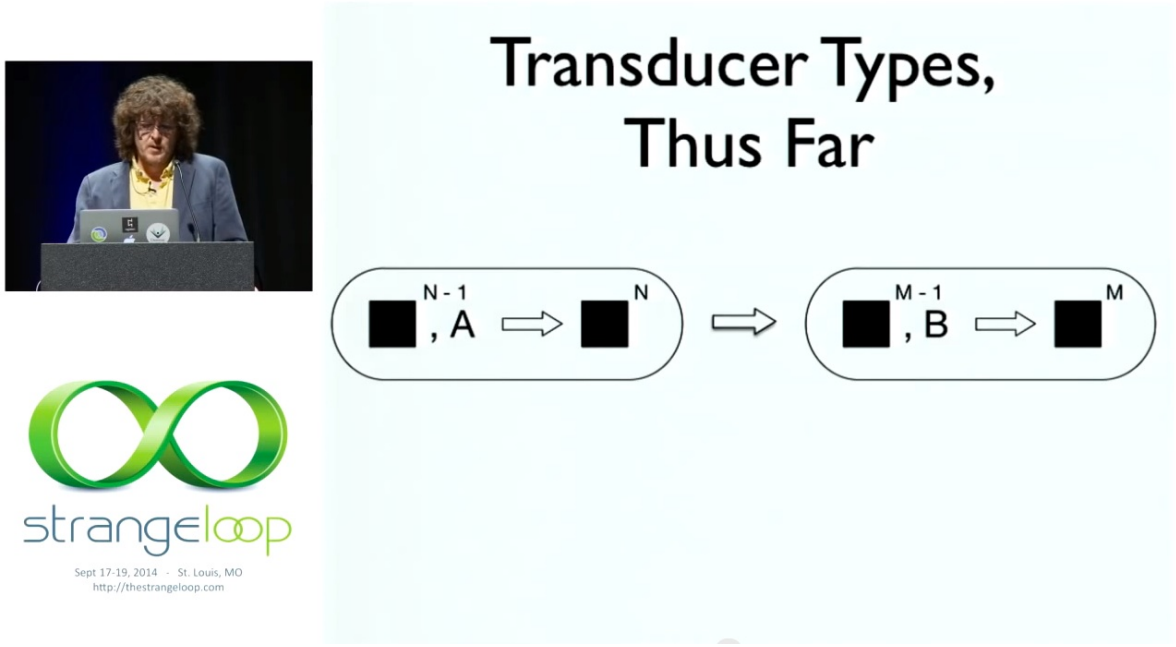

Figure 4: 27:56 Transducer Types, Thus Far

results that were sort of, "It works pretty well" to "Man, these types are killing me", depending on whether the user's type system could deal with it.

But let us just try to capture what we know so far graphically, and somebody who reviewed these slides for me said these should have been subscripts. But computers are so hard to use, I could not switch them in time. So they are superscripts.

But the idea is that if you are trying to produce the next process N, you *must* supply the result from step N-1 as the input. If you try to model this in your type system saying R to R, that is wrong. Because I can call the step function 5 times, and then on the 6th time, take the return value from the first time and pass it as the first thing. That is wrong. So you have got to make your type system make that wrong. So figure that out.

Also, if you make the black box and the black box the same thing, that is also arbitrarily restrictive. You could have a state machine that every time it was given X, returned Y. Every time it was given Y, returned Z. Every time it was given Z, returned X. That is a perfectly valid step function. It has 3 *separate* input types and 3 separate output types. It only happened at particular times. There is nothing wrong with that state machine. It is a perfectly fine reducing function. It may be tough to model in a type system. And do not say X or Y or Z. Because it does not take X or Y or Z and return X or Y or Z. When it is given X, it only returns Y. It never returns Z.

So seems like a good project for the bar, later on.

[Audience laughter]

But the thing that we are capturing here is that the new step function might take a different kind of input. It might take a B instead of an A. Now, our first step does that. It takes a pallet and returns a set of pieces of luggage, but each step returns a piece of luggage.

[Time 0:30:12]

```
slide title: Early Termination
```

```
+ Reduction normally processes all input
+ Sometimes a process has just 'had enough' input, or gotten external
  trigger to terminate
+ A transducer might decide the same

(comp
  (mapcatting unbundle-pallet)
  (taking-while non-ticking?)
  (filtering non-food?)
  (mapping label-heavy))
```

So there are other interesting things that happen in processes. Ordinary reduction processes everything, but we want this to be usable in cases that run arbitrarily long. We are not just talking about turning one kind of collection into another kind of collection. A transducer that is running on a channel has got an arbitrary amount of stuff coming through. A transducer on an event stream has an arbitrary amount of stuff coming through.

But sometimes you want either the reducing process, or somebody, says "Whoa! I have had enough. I do not want to see anymore input. we are done. I want to say we are done now, even though you may have more input".

So we are going to call that early termination. And it may be desired by the process itself, like the thing at the bottom. Or it may be a function of one of the steps. One of the steps may say, "You know what, that is all I was supposed to do. And so I do not want to see any more input".

And the example here will be, we are going to modify our instructions and say, "If the bag is ticking, you are finished. Go home. We are done loading the plane".

[Audience laughter]

So we are going to add that: taking-while non-ticking. And taking while non-ticking needs to stop the whole job in the middle. It does not matter if there is more stuff on the trolley. When it is ticking, we are finished.

[Time 0:31:30]

```
slide title: Reduced

+ Clojure's _reduce_ supports early termination via (reduced result)
+ A wrapper with a corresponding test: reduced?
+ And unwrap / dereference

  (reduced? (reduced x)) -> true
  (deref (reduced x)) -> x
```

So how do we do that? It ends up, in Clojure, we already have support for this idea in reduce. There is a constructor of a special wrapper object called "reduced", which says, "I do not want to see any more input. Here is what I have come up with so far, and do not give me any more input". And there is a predicate called "reduced?" that allows you to ask if there is something in this wrapper. And there is a way to unwrap the thing and look at what is in it.

So you can say, is the reduced thing reduced? That will always return true. And you can deref a reduced thing and get the thing that is inside it.

This is not the same thing as “Maybe”. Because “Maybe” also wraps the other things that are not reduced. Or “either”, or all of those other boxy kind of things. So we do not do that. We only wrap when we are doing this special termination.

[Time 0:32:23]

```
slide title: Transducers Support _reduced_

+ step functions can return (reduced value)
+ If a transducer gets a _reduced_ value from a nested step call,
  it must never call that step function again with input

  (defn taking-while [pred]
    (fn [step]
      (fn [r x]
        (if (pred x)
      (step r x)
      (reduced r)))))
```

So like reduce, transducers also must support reduced. That means that the step functions are allowed to return a reduced value. And that if a transducing process or a transducer gets a reduced value, it must never call the step function with input again. That is the rule. Again, implement the rule in your type system, have at it. But that is the rule.

So now we can look at the insides of taking-while. It takes a predicate. It takes a step that it is going to modify. It runs the predicate on the input. If it is OK, it runs the step. If it is not OK, it takes what has been built up so far and says, “we are finished. Reduced result”. That is how we bail out. But notice the ordinary result is not in a wrapper.

[Time 0:33:13]

```
slide title: Processes Must Support Reduced

+ If the step function returns a _reduced_ value, the process must
  not supply any more input to the step function
+ the dereferenced value is the final accumulation value
+ the final accumulation value is still subject
  to _completion_ (more later)
```

And so the reducing processes must also play this game. The transducer has to follow the rule from before, and the reducing process similarly has to support reduced. If it ever sees a reduced thing, it must never supply input again.

The dereferenced value is the final accumulated value. But the final accumulated value is still subject to completion, which I am going to talk about in a second. So there is a rule for the transducers as well. They have to follow this rule.

[Time 0:33:43]

```
slide title: Transducer Types, Thus Far
```

So now we get new pictorial types in the graphical type language that is OmniGraffle.

[Audience laughter]

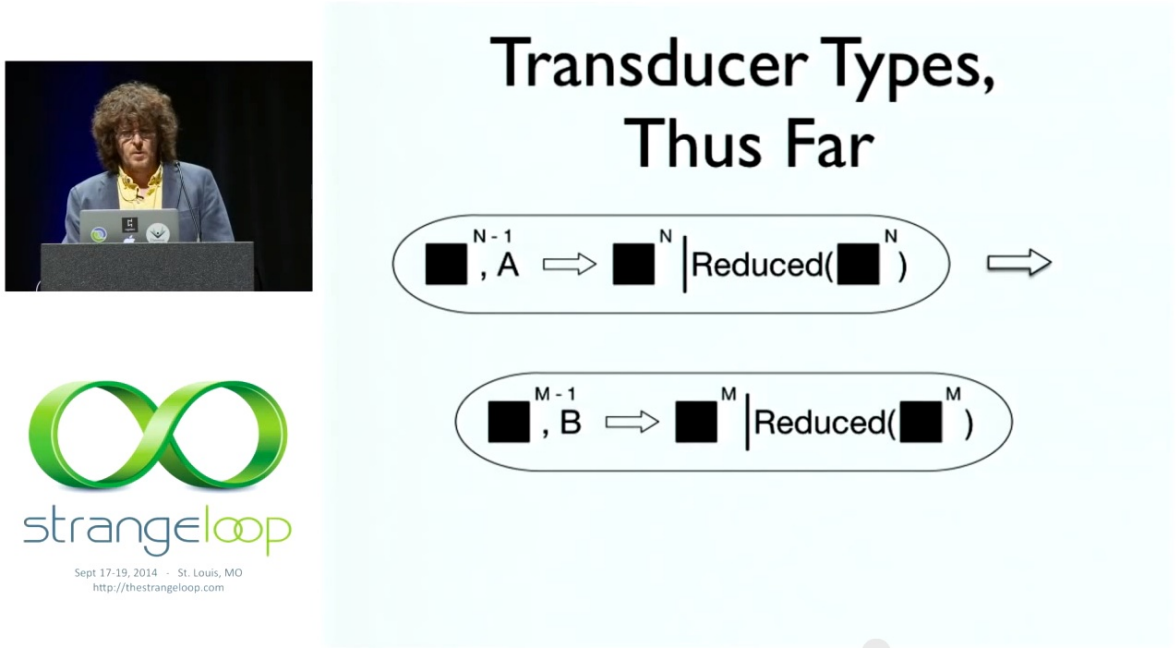

Figure 5: 33:43 Transducer Types, Thus Far

So we can have a process that takes some black box at the prior step and an input and returns a black box at the next step, or maybe it returns a reduced version of that. So one of those two things can happen. Vertical bar [|] is "or".

And it returns another step function that similarly can take a different kind of input, a black box, returns a black box or a reduced black box. Same rules about successorship apply. All right.

[Time 0:34:26]

```
slide title: State

+ Some transducers require state
  e.g. take, partition-*
+ Must create unique state every time they are called upon
  to transform a step fn
+ Thus, once applied to a process, a transducers yields
  another (potentially stateful) process, which should not be aliased
+ Pass transducers around and let processes apply them
```

So some interesting sequence functions require state. And in the purely functional implementations, they get to use the stack or laziness to put that state. They get somewhere in the execution machinery a place to put stuff.

Now we are saying, "I do not want to be in the business of specifying we are lazy or not lazy or recursive. I am not going to give you space inside the execution strategy, because I am trying to keep the execution strategy from you".

And that means the state has to be explicit when you have transducers. Each transducer that needs state, must create it.

So examples of sequence functions that need state are take, partition-all, partition-by and things like that. They are counting, or they are accumulating some stuff to spit it out later. Where is that going to go?

And it has to go inside the transducer object. They have to make state. And there is some rules about that. If you need state as a transducer author, you have to create it every time uniquely. And again, every time you are asked to transform a step function. So anew you are going to create state every time you transform a step function.

That means that if you build up a transducer stack, some of which are stateful transducers, and you apply it – not when you build it, no state exists then. Now, after you have called comp, there is no state. When you have applied it, you now have a new process step. But as we should be thinking about all transducer process steps, including the ones at the bottom, that may be stateful. You do not know that the very bottom process has not launched stuff into space. So you should *always* treat an applied transducer stack as if it would return a stateful process, which means you should not alias it.

What ends up happening in practice is: all of the transducible processes, they do the applying. It is not in the user's hands to do it. You pass around a transducer, and input to the job, to the job. The job applies the transducer to its process. It gets a fresh set of state when it does that, and there is no harm. But you do have to do this by convention.

[Time 0:36:36]

```
slide title: A Stateful Transducer
```

```
(defn dropping-while [pred]
  (fn [step]
    (let [dv (volatile! true)]
      (fn [r x]
    (let [drop? @dv]
      (if (and drop? (pred x))
        r
        (do
          (vreset! dv false)
      (step r x))))))))
```

So here is an example of a stateful transducer: dropping-while a predicate is true. So we start with our flag that says it is true. As long as it is still true, we are going to drop.

When we see that it is not true, we are going to reset it and continue with applying the step. And from then on forward, we are going to apply the step. So that is not the prettiest thing.

[Time 0:37:00]

```
slide title: Completion

+ Some processes complete, and will receive no more input
+ A process might want to do a final transformation of the value built up
+ A stateful transducers might want to flush a pending value
+ All step functions _must_ have an arity-1 variant
  that does not take an input
```

I talked before about completion. So we have the idea of early termination. The other idea that transducers support is completion. Which is that, at the end of input, which may not happen. There

will be plenty of jobs that do not complete. They do not have ends. They are not consuming a finite thing like a collection. They are processing everything that comes through a channel, or everything that comes through an event source. There is no end.

But for things that have an end, there is a notion of completion, which is to say: if either the innermost process step wants to do something finally when everything is finished, they can. Or if any of the transducers have some flushing they need to do, they can do it.

So the process may want to do a final transformation on the output. Any stateful transducer, in particular a transducer like partition, it is aggregating to return aggregates. You say, partition 5 and it collects five things and spits it out. If you say "we are done", and it has got three things, it wants to spit out the three things. But you need to be able to tell it, "we have exhausted input".

In order to do that, the way that is implemented in the Clojure implementation of transducers is that all the step functions must have a second operation. So there is the operation that takes a new input, and the accumulated value so far, and returns a new accumulated value, or whatever. It is up to the process what the meaning of the black box is. But there must be another operation which takes just the accumulated value and no input. So an arity-1 operation. So that is required.

[Time 0:38:39]

```
slide title: Completion Operation

+ A completing process _must_ call the completion operation on
  the final accumulated value, exactly once

+ A transducer's completion operation _must_ call its nested completion
  operation, exactly once, and return what it returns

+ A stateful transducer _may_ flush state (using the nested step function)
  prior to calling the nested complete*.  _partition-all_ and
  _partition-while_ are examples.
```

So we will talk about what that does, or how that gets used. If the process itself, if the overall job has finished, if it has exhausted input, or it has a notion of being finished – this is not bailing out; this is like there is nothing more to do; there is no more input ordinarily – it must call a completion operation exactly once on the accumulated value. So there is no more inputs, I am going call you once with no input. Do whatever you want.

Each transducer must do the same thing. It has to have one of these completion operations, and it must call its nested completion operation.

It may, however, before it does that, flush. So if you have something like partition that has accumulated some stuff along the way, it can call the ordinary step function, and then call complete on the result. And that is how we accomplish flushing.

There is just one caveat here, which is that if you are a stateful thing like partition, and you have ever seen reduced come up, the earlier rule says you can never call the input function. So you just drop whatever you have hanging around, because somebody bailed out on this process. There is going to be no ordinary completion.

[Time 0:39:47]

```
slide title: Transducer Types, Thus Far
```

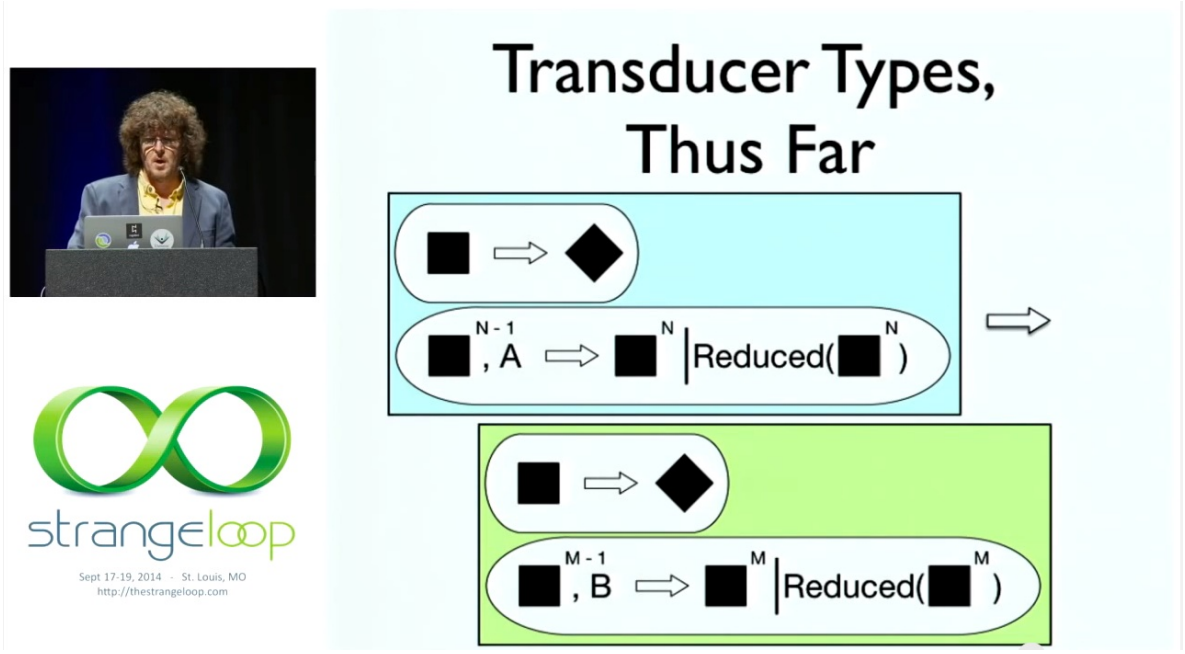

Figure 6: 39:47 Transducer Types, Thus Far

So we can look at our types again in OmniGraffle 2000, the latest programming innovation. And think about a reducing function as a pair of operations. They will be different in each programming language. It is not really important. In Clojure, it ends up a single function can capture both of these arities, but whatever you need to do to take two operations.

The first one up there that takes no input is the completion operation. And the second is the step operation we have been seeing so far. It takes a pair of those things and returns a pair of those things. That is it.

And again, we do not want to concretely parameterize the result type there either. You have got to use rank 2 polymorphism, or something, because if you concretely parameterize that, you will have something that only knows about transducing into airplanes, as opposed to the general instructions.

[Time 0:40:45]

```
slide title: Init

+ A reducing function _may_ support arity-0, which returns
  an initial accumulation value
+ Transducers _must_ support arity-0 init in terms of
  a call to the nested init

  user=> (+)
  0
  user=> (+ 21)
  21
  user=> (+ 21 21)
  42
```

OK, there is a third kind of operation that is associated with processing in general, which is Init. We

have had talks before that mention monoids and things like that. The basic idea is just, sometimes it is nice for a transformation operation to carry around an initialization capability. It need not be the identity value or anything like that. It does not matter.

What does matter is that a reducing function is allowed to, may, support arity-0. In other words, given nothing at all, here is an initial accumulator value. From nothing.

Obviously, a transducer cannot do that because it is a black box. The one thing it definitely does not know how to do is to make a black box out of nothing. Cannot do it. So all it can ever do is call down to the nested function. So transducers must support arity-0, init, and they just define it in terms of a call to the nested step. They cannot really do it, but they can carry it forward so that the resulting transducer also has an init, *if* the bottom transducer has an init.

I have talked about the arity overloading, and so here is an example. Plus [+] from Lisp. This is old. This is older than transducers. Lisp programmers have been doing this for a while. Sorry Currying fans, this is what we do. Plus with nothing returns the identity value for plus: 0. Multiplication with nothing returns 1. It implements plus of an accumulated result as identity, and the binary operation that does the work.

[Time 0:42:25]

`slide title: Transducer Types`

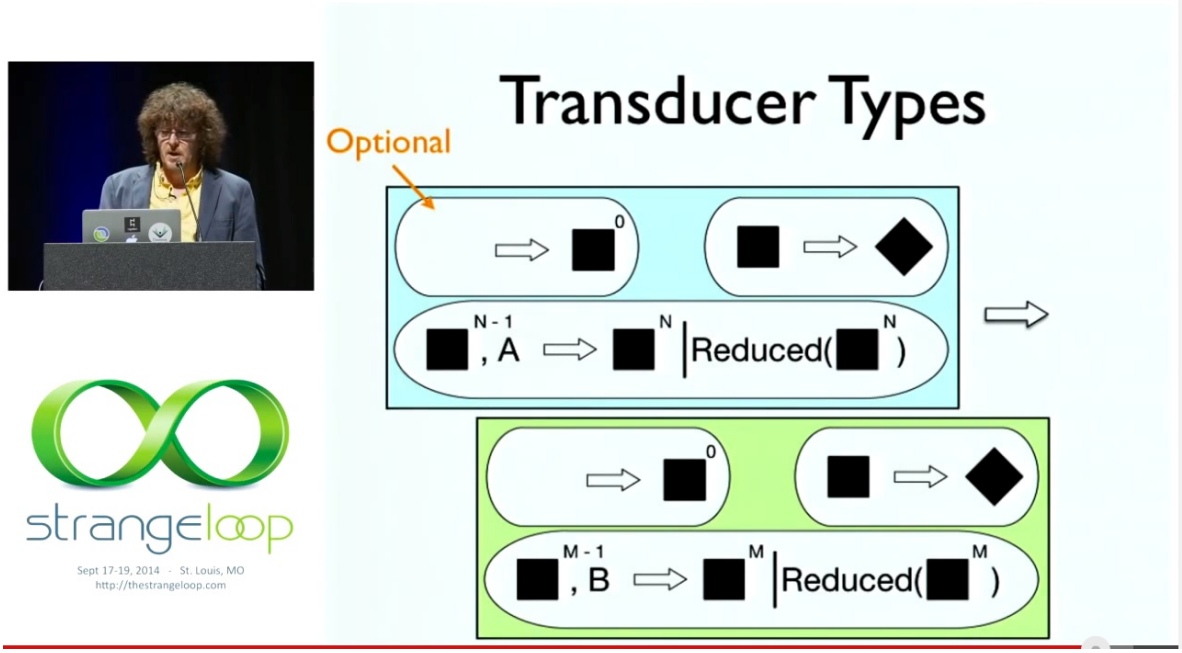

Figure 7: 42:25 Transducer Types

So here are the types again. We now have an optional init from nothing. And we are taking a set of three operations and returning a new set of three operations.

[Time 0:42:36]

`slide title: Clojure Implementation`

```
+ Reducing fns are just arity 0, 1, 2 functions

+ Transducers take and return reducing fns

+ Core sequence functions' collectionless arity now returns a transducer:
  (mapping f) == (map f)

+ map, mapcat, filter, remove, take, take-while, drop, drop-while,
  take-nth, replace, partition-by, partition-all, keep, keep-indexed,
  cat, dedupe, random-sample ...
```

In Clojure, we just used arity to do this. A transducer enclosure then is just something that takes a reducing function and returns one, where a reducing function has these three arities.

We have not actually called the reducing functions "mapping" and "filtering" and -ing this and -ing that. I think that is an English-ism that is not going to carry over very well. And we have available to us arity overloading because we do not have currying. So "map" of f with no collection argument returns the transducer. And we have modified, so far, all of these sequence functions to do that.

[Time 0:43:09]

```
slide title: Filter, returning a Transducer
```

```
  (defn filter
    ([pred]
      (fn [rf]
        (fn
      ([] (rf))
      ([result] (rf result))
      ([result input]
        (if (pred input)
          (rf result input)
          result)))))
    ([pred coll]
      (sequence (filter pred) coll)))
```

So this is the final example of "filter" returning a transducer. It takes a predicate and returns us a step modifying function, which takes a reducing function, which presumably has these three arities, and defines a function with three arities.

init, which just flows it through, because it does not know what it could possibly do.

complete: filter does not have anything special to do, so it just flows that through.

And then the result and input one, which is the one we have seen before.

Now we can see, we can define the collection-implementing one by just calling sequence with this transducer. And that is true of all of these functions. You can define the collection version exactly like this, which shows that transducer is more primitive than the other.

[Time 0:43:52]

```
slide title: The Goal
```

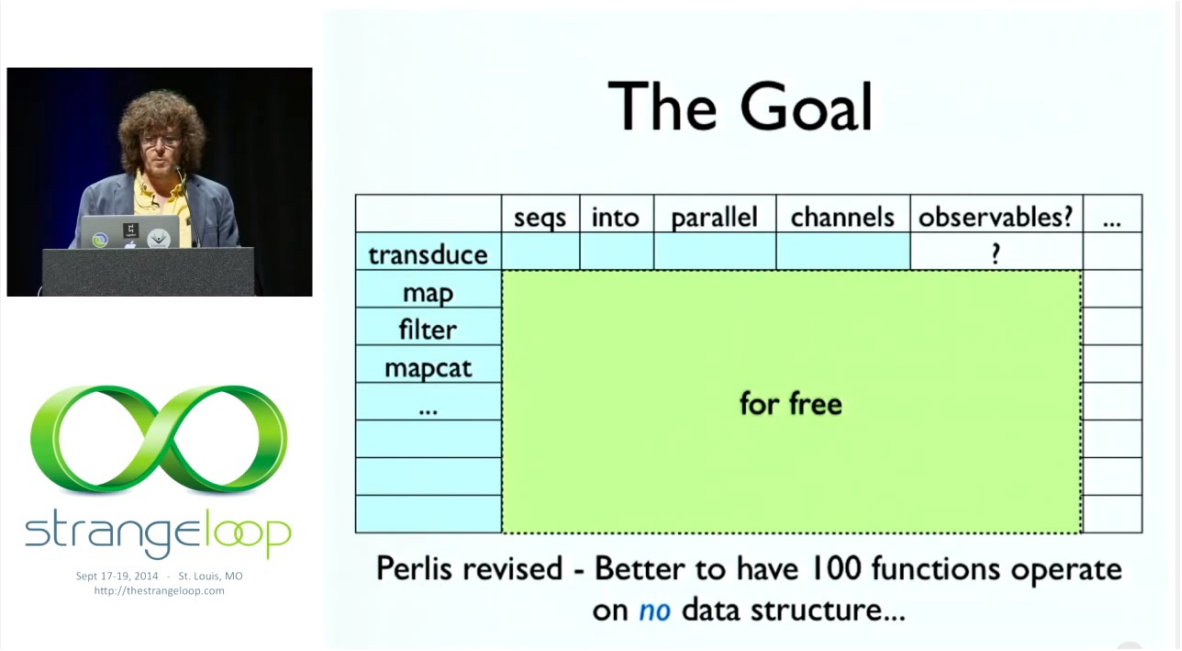

Figure 8: 43:52 The Goal

So this is what we are trying to accomplish. You define a set of transducers once. You define all your new cool stuff. So channels today, observables tomorrow, whatever the next day. You just make it accept transducers, and every specific implementation of these things, you get for free. And every recipe that somebody creates, that is a composition of those transducing operations, works with your thing right away.

That is what we want, right? we are going to take Perlis and just say it is even better. We want a hundred functions with *no* data structure.

- in reference to "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures." - Alan Perlis

[Time 0:44:27]

```
slide title: Transducers

+ Transducers support context-independent definitions
  of data transformations
+ Reusable across a wide variety of contexts
+ Support early termination and completion
+ Composable via ordinary function composition
+ Efficient
+ Tasty
```

So, transducers are context-independent. There is tremendous value in that. They are concretely reusable. So somebody can make this and not how you are going to use it. That has tremendous value. It is much stronger than parameterization. Because you can flow it.

They support early termination and completion. You can compose them just as easily as you can compose the other ones. They are efficient and tasty.

Thanks.

[Audience applause]

[Time 0:44:56]