CLOJURE TRANSDUCERS WORKSHOP

1 DAY, 4 UNITS AND RELATED LABS

HTTPS://GITHUB.COM/USWITCH/USWITCH-ACADEMY/TREE/MASTER/CLOJURE-TRANSDUCERS

AGENDA

1. Transducers basics
2. Custom transducers
3. Transducers and core.async
4. Parallelizing transducers

1. THE BASICS

WHAT ARE TRANSDUCERS?

- A model for sequential processing
- An abstraction on top of `reduce`
- A functional pattern
- A computational recipe

WHAT ARE THEY NOT?

- A library
- A replacement for sequences
- A performance optimization (not only)
- Reducers (but they share a similar design)

QUICK COMPARE

```clojure
(reduce +
  (filter odd?
    (map inc
      (range 10))))
```

```clojure
(transduce
  (comp
    (map inc)
    (filter odd?))
  + (range 10))
```

MORE COMPARE

```clojure
(->>
  (range 10)
  (map inc)
  (filter odd?)
  (reduce +))
```

```clojure
(transduce
  (comp
    (map inc)
    (filter odd?))
  + (range 10))
```

CLEAR DIFFERENCES

```
**Sequential**                    **Transducers**

* No `comp`                       `comp` (remove nesting)
* Nested calls                    `transduce` instead of `reduce`
* Using `reduce`                  Transforms are "grouped"
```

SUBTLE DIFFERENCES

```
**Sequential**                      **Transducers**
* n-intermediate sequences          * No intermediate sequences
* Isolated Transforms               * Composed Transforms
* n-sequential scans                * `transduce` uses `reduce`
* Transform on sequential scan      * Transform evaluates lazily
```

WHY DO WE CARE?

- Transforms are isolated from sequential mechanism
- Transforms are more composable/reusable
- Single pass iteration boost performances
- Sequential iteration is someone else's responsibility

ALWAYS POSSIBLE?

- Some transforms aren't immediate to translate:

```
(->> [[0 1 2] [3 4 5] [6 7 8]] (apply map vector))
```

- Scenarios involving infinite laziness:

```
(take 3 (sequence (mapcat repeat) [1])) ;; boom!
```

- Realising intermediate results unnecessarily:

```
(first
  (sequence
    (comp (mapcat range) (mapcat range))
  [3000 6000 9000])) ;; boom!
```

ALWAYS FASTER?

- Avoid on small collections
- Avoid for just few transforms
- Avoid for trivial transforms

When in doubt, measure

MAIN API

- `transduce`: eager, single pass. All input evaluated.
- `sequence`: delayed, chunked (32), cached.
- `eduction`: delayed, chunked (32) no caching.
- `into`: eager. `transduce` "into" another data type.

CURRENT LINE-UP

```
mapcat, remove, take, take-while, take-nth
drop, drop-while, replace, partition-by, halt-when
partition-all, keep, keep-indexed, map-indexed
distinct, interpose, dedupe, random-sample, cat
```

RESOURCES

- **Transducers presentation** by Rich
- Transducers official **reference guide**
- Article about the Transducers **functional abstraction**

LABS INTRO

```
git clone https://github.com/uswitch/uswitch-academy.git
cd clojure-transducers
lein repl
```

It is assumed you can evaluate the code in the labs with your favourite IDE.

DRIVING EXAMPLE

- App receiving regular updates of fin products
- Users can search for the best product.
- Each update contains +10k products as Clojure maps.
- We want to process the data in a modular/timely manner.
- Open `src/transducers_workshop/lab01.clj` to get started

2. CUSTOM XFORMS

J. Functional Programming **9** (4): 355–372, July 1999. Printed in the United Kingdom

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

University of Nottingham, Nottingham, UK
`http://www.cs.nott.ac.uk/~gmh`

Abstract

In functional programming, *fold* is a standard operator that encapsulates a simple pattern of recursion for processing lists. This article is a tutorial on two key aspects of the fold operator for lists. First of all, we emphasize the use of the *universal property* of fold both as a *proof principle* that avoids the need for inductive proofs, and as a *definition principle* that guides the transformation of recursive functions into definitions using fold. Secondly, we show that even though the pattern of recursion encapsulated by fold is simple, in a language with tuples and functions as first-class values the fold operator has greater *expressive power* than might first be expected.

`fold`

- The idea: redefine sequential operations as folds
- `fold` is a class of recursive-iterative algorithms
- Results are part of the args, no stack consumption
- Transform "f" also knows how to accumulate results
- `reduce` is fold-left: "folds" items from left to right
- Effects: isolate transforms and accumulation details

CORE MAP/FILTER

```clojure
(defn map [f coll]
  (lazy-seq
    (when-let [s (seq coll)] ; terminating condition
      (cons
        (f (first s))        ; applying transform
        (map f (rest s))))))) ; non tail-recursive

(defn filter [pred coll]
  (lazy-seq
    (when-let [s (seq coll)]
      (let [f (first s) r (rest s)]
        (if (pred f)
          (cons f (filter pred r))
          (filter pred r))))))
```

FOLD-LEFT (`aka` REDUCE)

```
(defn fold
 ([f coll]                    ; arity to setup an "init"
  (fold f [] coll))
 ([f result coll]
  (if coll                    ; termination condition
   (recur                     ; tail recursive
    f                         ; "f" decides how to accumulate
    (f result (first coll)) ; transform applied here
    (next coll))              ; move on next element
   result)))                  ; return results when no more items
```

MOVING TO FOLD

- Express `map`, `filter`, etc. as `fold`
- We need to move from linear recursive to iterative.
- We need "f" to gradually build results.
- We need an "init" result to start from.
- It can't be lazy (there is no seq building)

STEP 1: SHAPE-UP

```
(defn map [f result coll]
  (if coll
    (map f (f result (first coll)) (next coll))
    result))

;; Example: (map inc (range 10))
(map #(conj %1 (inc %2)) [] (range 10))

(defn filter [f result coll]
  (if coll
    (filter f (f result (first coll)) (next coll))
    result))

;; Example: (filter odd? (range 10))
(filter #(if (odd? %2) (conj %1 %2) %1) [] (range 10))
```

STEP 2: RENAME

`map` and `filter` are the same! Rename to transform.

```clojure
(defn transform [f result coll]
  (if coll
    (transform f (f result (first coll)) (next coll))
    result))

;; Example: (map inc (range 10))
(transform #(conj %1 (inc %2)) [] (range 10))

;; Example: (filter odd? (range 10))
(transform #(if (odd? %2) (conj %1 %2) %1) [] (range 10))
```

STEP 3: TRANSFORM==REDUCE

`transform` **is** `reduce`!

```
;; Example: (map inc (range 10))
(reduce #(conj %1 (inc %2)) [] (range 10))

;; Example: (filter odd? (range 10))
(reduce #(if (odd? %2) (conj %1 %2) %1) [] (range 10))
```

STEP 4: ANON TO FUNCTION

Let's extract those anons into fns.

```
(defn mapping [result el]
  (conj result (inc el)))

;; Example: (map inc (range 10))
(reduce mapping [] (range 10))

(defn filtering [result el]
  (if (odd? el)
    (conj result el)
    result))

;; Example: (filter odd? (range 10))
(reduce filtering [] (range 10))
```

STEP 5: EXTRACT ACCUMULATION

`conj` is specific accumulation logic. Extract param.

```
(defn mapping [rf]
  (fn [result el]
    (rf result (inc el))))

;; Example: (map inc (range 10))
(reduce (mapping conj) [] (range 10))

(defn filtering [rf]
  (fn [result el]
    (if (odd? el)
      (rf result el)
      result)))

;; Example: (filter odd? (range 10))
(reduce (filtering conj) [] (range 10))
```

STEP 6: EXTRACT TRANSFORM

`inc` and `odd?` are specific transform logic. Extract param.

```
(defn mapping [f]
  (fn [rf]
    (fn [result el]
      (rf result (f el)))))

;; Example: (map inc (range 10))
(reduce ((mapping inc) conj) [] (range 10))

(defn filtering [pred]
  (fn [rf]
    (fn [result el]
      (if (pred el) (rf result el) result))))

;; Example: (filter odd? (range 10))
(reduce ((filtering odd?) conj) [] (range 10))
```

STEP 7: ENCAPSULATE CALL

`mapping` and `filtering` need preparation for use. Extract that complexity away in new function `wrapper` "init" can be obtained from `(rf)`

```
(defn wrapper [xf rf coll]
  (reduce (xf rf) (rf) coll))

;; Example: (map inc (range 10))
(wrapper (mapping inc) conj (range 10))

;; Example: (filter odd? (range 10))
(wrapper (filtering odd?) conj (range 10))
```

STEP 8: FINAL TOUCHES

- We reached our desired form.
- Can you guess how "wrapper" was officially named?
- And what about "mapping" or "filtering"?
- What mapping/filtering have in common?

ADDITIONAL DETAILS

- `mapping`/`filtering` are very similar to map/filter
- Same for `wrapper` which was named `transduce`
- The stdlib also implements setup/tear-down behaviors
- This is why `map` and `filter` xform have more arities
- A "good transducer" also need to behave correctly

DESIGNING A TRANSDUCER

- Deal with the end of the reduction in 1-arg arity
- Provide an initial value in 0-arg arity (currently unused)
- Where to initialize state (for stateful xforms)
- How to terminate early (if needed)
- Surrounding xforms awareness (mandatory calls)

RESOURCES

- A tutorial on the **universality and expressiveness of fold**
- uSwitch Labs **transducers articles**

LAB 02

- Task 1: create a "logging" transducer to print useful info.
- Task 2: create a stateful moving average transducer
- Open `src/transducers_workshop/lab02.clj` to get started

3. XF CORE.ASYNC

WHERE IT ALL STARTED

- core.async was shaping up back in 2012-2013
- Having `(map f in out)` was desirable feat.
- But a channel is not a sequence
- Don't want to reimplement it all over:

```
(defn map [f in out]
 (go-loop []
  (let [val (<! in)]
   (if (nil? val)
    (close! out)
    (do (doseq [v (f val)] (>! out v))
        (when-not (impl/closed? out)
         (recur)))))))
```

REUSE THE SAME XFORMS

- No need to reimplement all over.
- Reuse the same xforms! 3 options:
- `a/channel`
- `a/transduce`
- `a/pipeline`

a/channel

```clojure
(defn consumer [xf]
  (let [in (a/chan (a/sliding-buffer 64) xf)]
    (a/go-loop [x (a/<! in)]
      (when x
        (println "consumer received data" x)
        (recur (a/<! in)))) in))

(defn producer [xs & chs]
  (a/go (doseq [x xs ch chs]
      (a/<! (a/timeout 1000))
      (a/>! ch x))))
```

```clojure
(producer
  "hello"
  (consumer (map (comp keyword str)))
  (consumer (map int)))

;; consumer received data 104
;; consumer received data :e
…
```

A/TRANSDUCE

- Accepts a channel as input
- Returns a channel with the reduction results

```
(a/<!!
  (a/transduce
    (comp (map inc) (filter odd?))
    + 0
    (a/to-chan (range 10))))
;; 25
```

A/PIPELINE

```
(let [out (a/chan (a/buffer 100))]
  (a/pipeline 4 out
    (comp (map inc))
    (a/to-chan (range 10)))
  (a/<!! (a/into [] out)))
;; [1 2 3 4 5 6 7 8 9 10]
```

RESOURCES

- Communicating Sequential Processes (CSP) **paper**
- Core.async **walk-through**
- **Brave Clojure guide**

LAB 03

- Task 1: create channels and orchestrate them so incoming products end up in a local cache.
- Task 2: use the "prepare-data" transducer from `lab01` to transform incoming products.

4. GOING PARALLEL

PARALLELISM

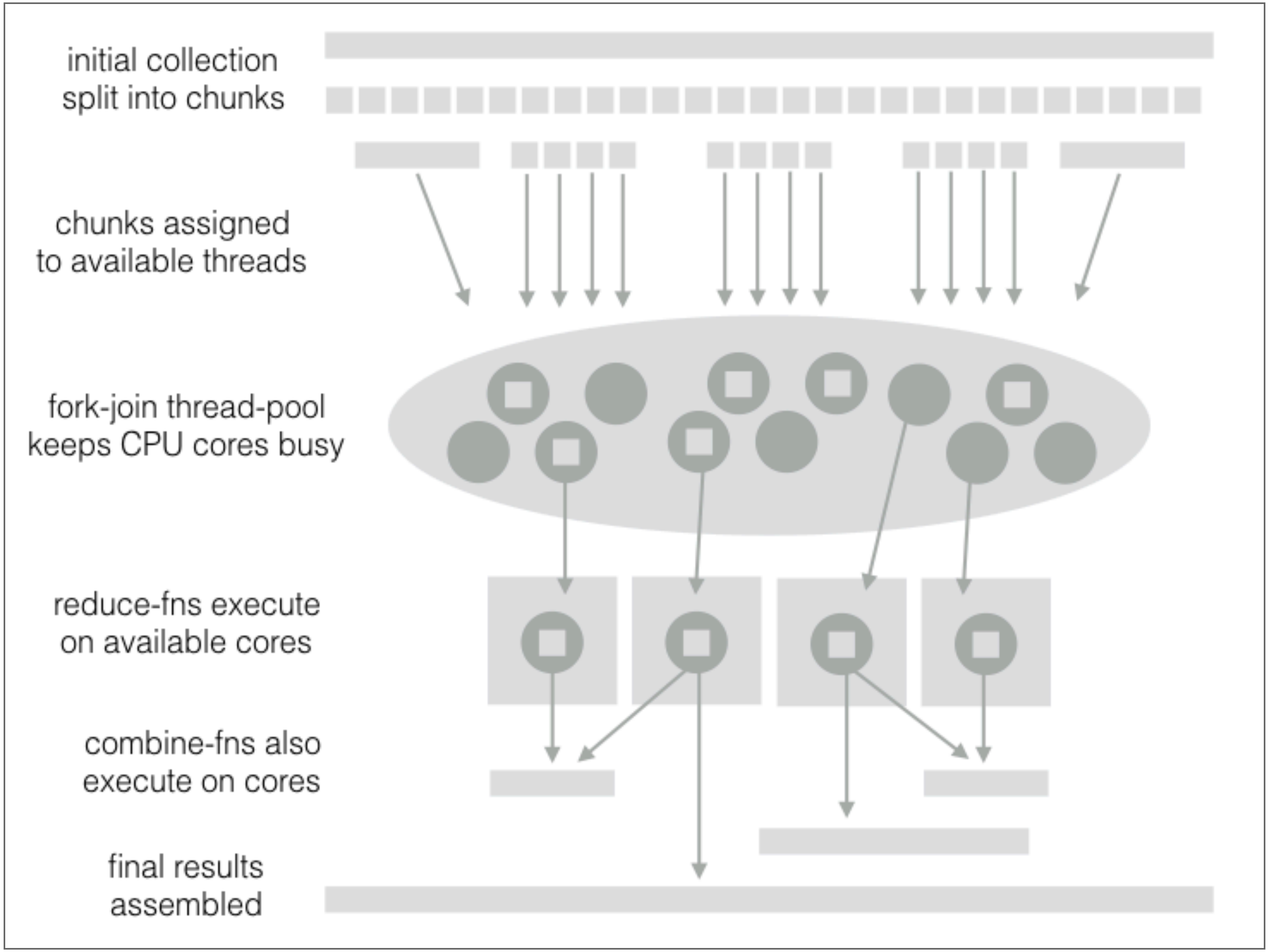
- Present in Clojure in different forms:
- `pmap` (lazy, sequential, constrained)
- `r/fold` (work-stealing, fork-join)
- Custom (`future`, `agent`, etc.)

PARALLEL TRANSDUCERS

- Transducers could run in parallel for additional perf.
- Approach 1. Divide and conquer: split the input and work in parallel
- Approach 2. `core.async` pipelines

R/FOLD

- `clojure.core.reducers` is part of Clojure
- fork-join is a divide and conquer strategy with work-stealing
- The input can be split into chunks (so no laziness)
- Combining the chunks back must be commutative


initial collection
split into chunks
chunks assigned
to available threads
fork-join thread-pool
keeps CPU cores busy
reduce-fns execute
on available cores
combine-fns also
execute on cores
final results
assembled

HOW

- Transducers are f: rf -> rf
- Call with "+" to obtain the transformed rf
- Use transformed rf in `r/fold` as usual
- To go parallel, you need vector/map
- Or possibly your custom `IFold`
- Works only with stateless transducers

EXAMPLE

```clojure
(require '[clojure.core.reducers :as r])

(r/fold +
  ((comp
    (map inc) (filter odd?)) +)
  (vec (range 1000)))
;; 250000
```

STATEFUL PROBLEM

```
(distinct (for [i (range 1000)]
  (r/fold +
    ((comp
      (map inc) (drop 1) (filter odd?)) +)
    (vec (range 1000)))))
;; (249999 249498 249499)
```

- Results are inconsistent
- Depending on which thread runs `(drop 1)`
- You get different results each run

PIPELINES

- `core.async` provides a `pipeline` construct
- `pipeline` can be further "piped" together
- Each `pipeline` declares the parallelism degree
- Each `pipeline` can apply a different transducer

EXAMPLE

```
(a/pipeline
   (inc (.availableProcessors (Runtime/getRuntime)))
   (a/chan out)
   (comp (map inc) (filter odd?))
   (a/to-chan (range 1000)))
```

- "availableProcessor" is the number of parallel threads
- Adding 1 or 2 threads to keep it buys
- Use in/out channels to pipe more of them together

RESOURCES

- The **parallel** library enables consistent stateful xforms in parallel.
- **A Java fork-join framework** paper by Doug Lea
- **Clojure Applied** book contains chapters dedicated to Transducers with core.async pipelines examples.
- **Standard Library book**, Chapter 7 Reducers and Transducers

LAB 04

- Task1: parallelise the xform with reducers.
- Task2: parallelise the xform with pipelines.
- Task3: different pipelines for different transducers.

- **Clojure Transducers Workshop**
- **Agenda**
- **### 1. The Basics**
- **What are transducers?**
- **What are they not?**
- **Quick compare**
- **More compare**
- **Clear differences**
- **Subtle differences**
- **Why do we care?**
- **Always possible?**
- **Always faster?**
- **Main API**
- **Current line-up**

- Resources
- Labs Intro
- Driving Example
- ### 2. Custom Xforms
- fold
- core map/filter
- fold-left (aka reduce)
- Moving to fold
- step 1: shape-up
- step 2: rename
- step 3: transform==reduce
- step 4: anon to function
- step 5: Extract accumulation
- step 6: Extract transform
- step 7: encapsulate call
- step 8: final touches
- Additional details
- Designing a transducer
- Resources
- Lab 02
- ### 3. Xf core.async
- Where it all started
- Reuse the same xforms
- a/channel
- a/transduce

- a/transduce
- a/pipeline
- Resources
- Lab 03
- ### 4. Going parallel
- Parallelism
- Parallel transducers
- r/fold
-
- How
- Example
- Stateful problem
- Pipelines
- Example
- Resources
- Lab 04