

The Elements of a Functional Mindset

Training the habits of functional thinking
Eric Normand - PurelyFunctional.tv

**What is uniquely
human?**







Prelude
Op. 28, No. 7

Frederic Chopin

Andantino

Piano

p dolce

con pedale

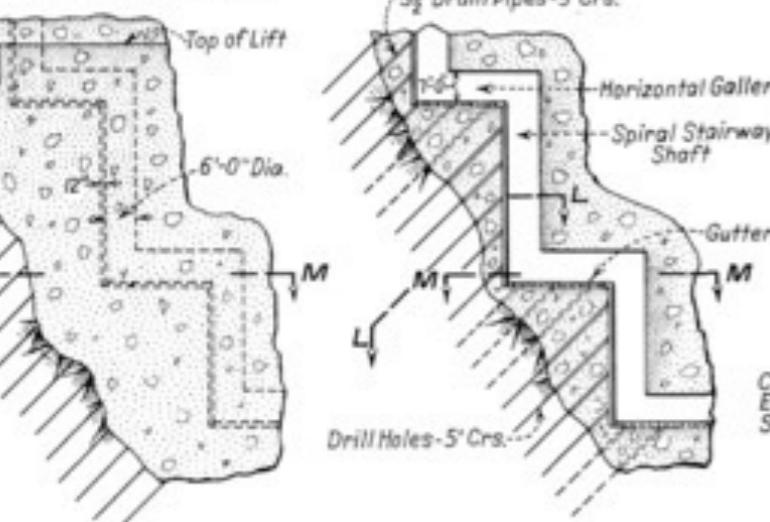
8

mp

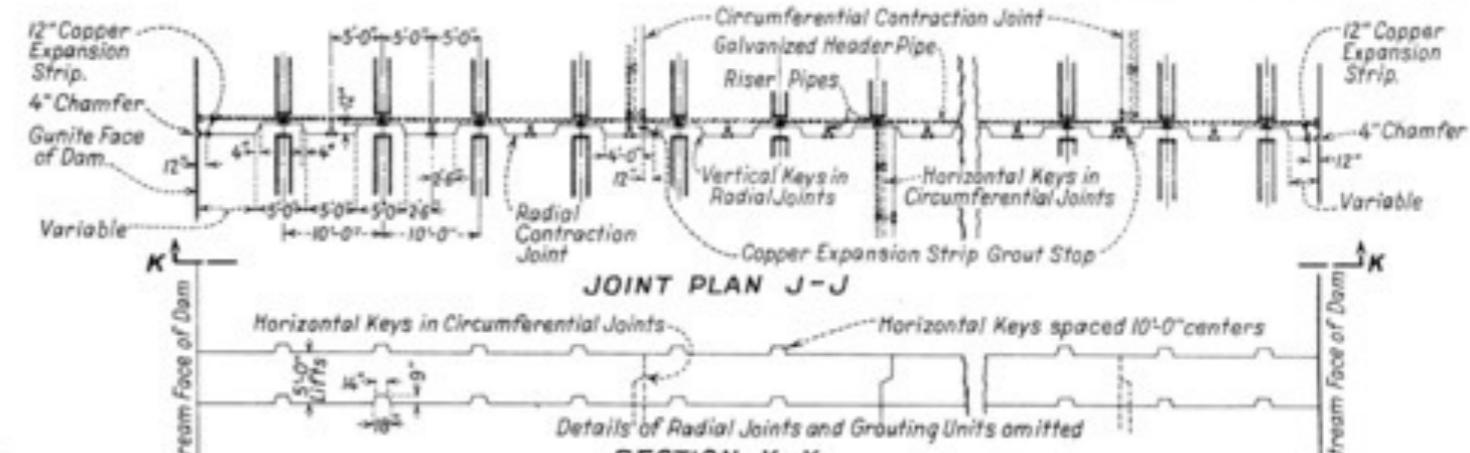
mp

rit. e dim. - - pp

-Area to be grouted from one Level
to be determined in Field.



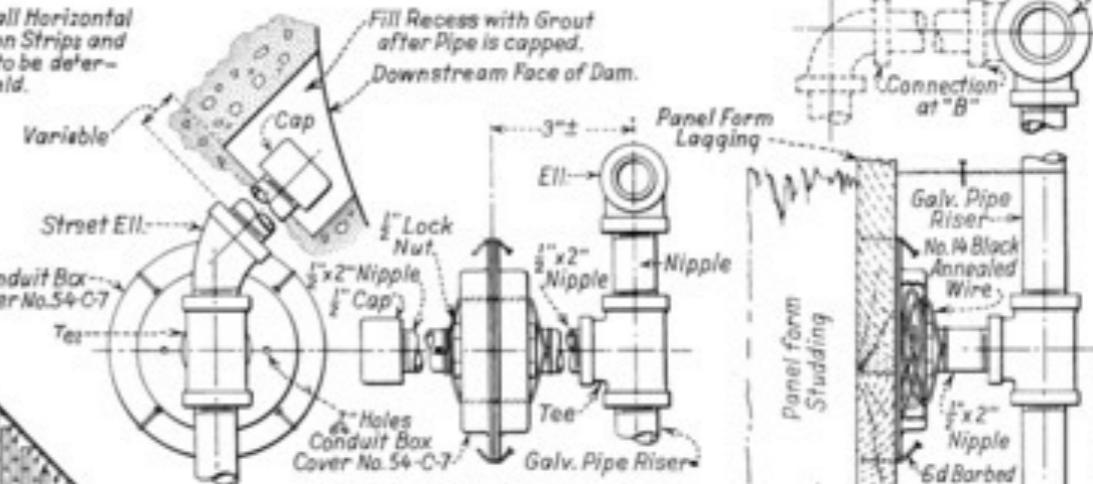
*Note:- For Header Layout
see Drawing No.
45-D-927*



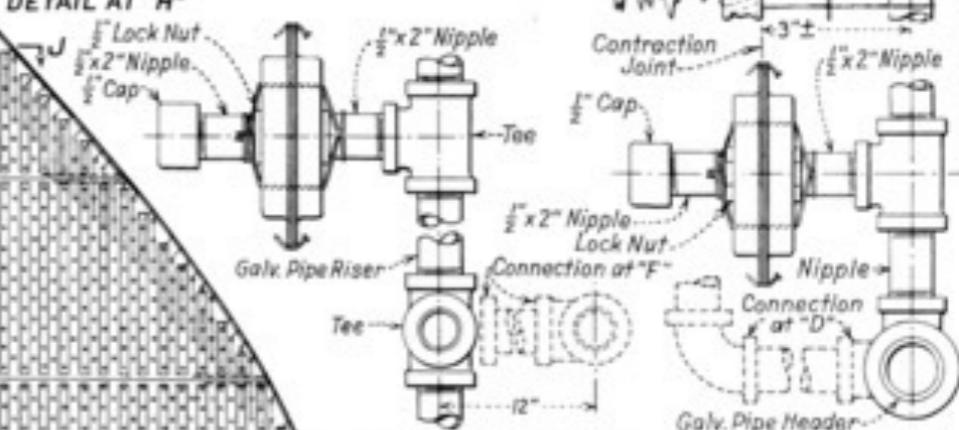
SECTION K-K
TYPICAL DETAIL OF JOINTS AND KEYS

Note:-Elevation of all Horizontal Copper Expansion Strips and Grout Headers to be determined in the Field.

www.browntrout.com

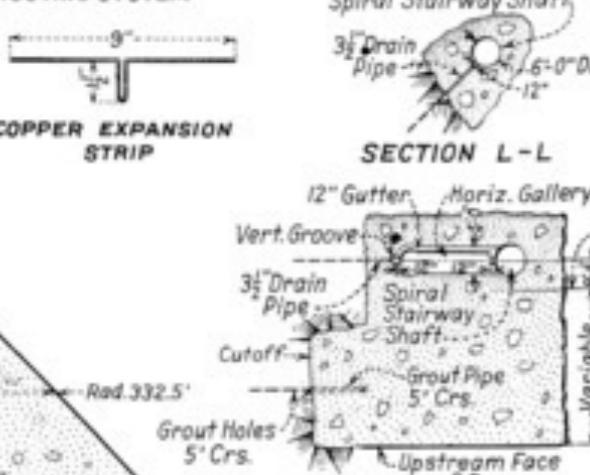


DETACH AT "M"

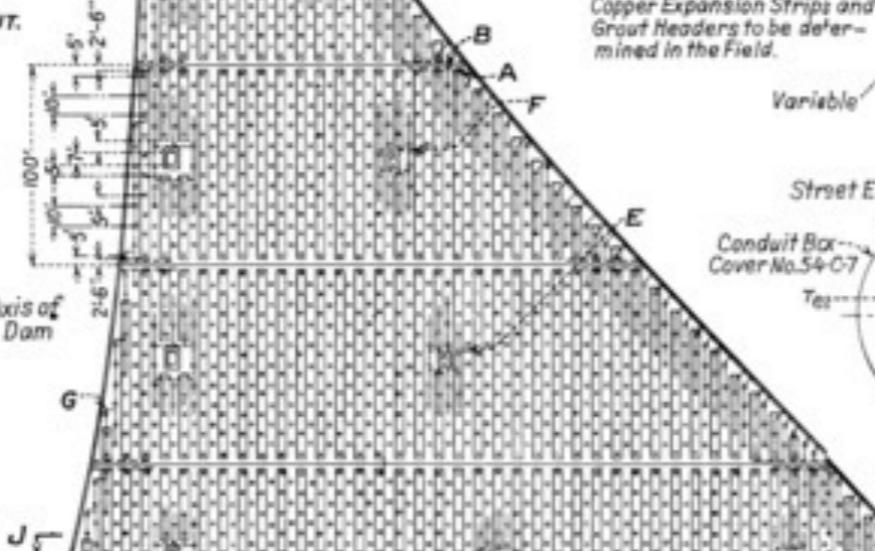


DETACH AT SEAM AND USE **DETACH AT ONE ANGLE**

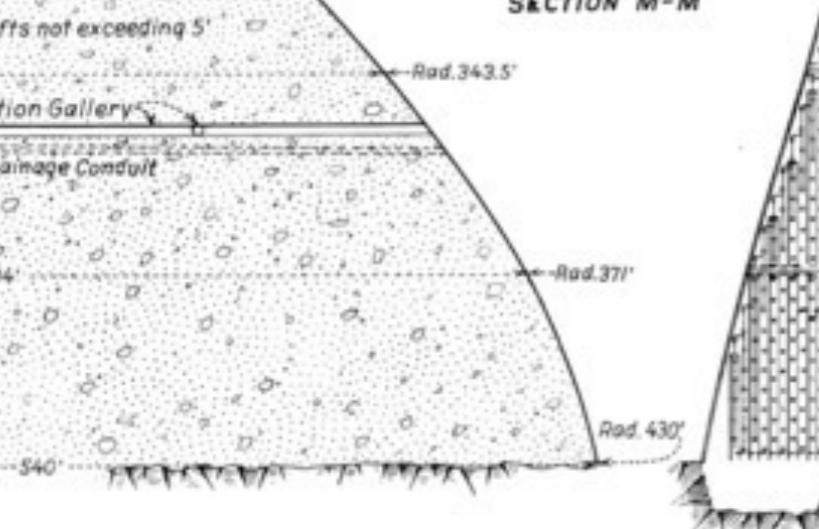
**TYPICAL SECTIONAL ELEV. ABUT
CUTOFF GROUTING SYSTEM**



**TYPICAL SECTIONAL ELEV. ABU
CUTOFF DRAINAGE SYSTEM**



SECTION M-



TYPICAL ELEVATION CONTRACTION JOINT

DEPARTMENT OF THE INTERIOR
BUREAU OF RECLAMATION
BOULDER CANYON PROJECT

HOOVER DAM
MAXIMUM SECTION OF DAM
GROUT AND DRAINAGE SYSTEM

DRAWN & C.M. CR. SUBMITTED
TRACED & A.A. RECOMMENDED *H. Sted*
CHECKED & P.D.O.L. APPROVED *R. F. Mather*

24124 DENVER, COLO., DEC. 1, 1930 45-D-924

A close-up portrait of Edsger Dijkstra, an elderly man with a full, bushy grey beard and mustache. He has short, receding hair and is wearing gold-rimmed glasses. He is dressed in a white collared shirt under a red ribbed sweater vest. He is looking slightly to his right with a thoughtful expression.

Edsger Dijkstra

The purpose of abstraction
is not to be vague, but to
create a **new semantic
level** in which one can be
absolutely **precise**.

```
(def record-sum    (atom 0))
(def record-count (atom 0))

(defn average-records []
  (doseq [record (fetch-records)]
    (swap! record-sum  + (:score record)))
  (swap! record-count + 1))
(/ @record-sum @record-count))
```

```
(def record-sum (atom 0))
(def record-count (atom 0))

(defn average-records []
  (doseq [record (fetch-records)]
    (swap! record-sum + (:score record)))
  (swap! record-count + 1))
(/ @record-sum @record-count))
```

```
(def record-sum (atom 0))
(def record-count (atom 0))

(defn average-records []
  (doseq [record (fetch-records)]
    (swap! record-sum + (:score record)))
  (swap! record-count + 1))
(/ @record-sum @record-count))
```

```
(def record-sum (atom 0))
(def record-count (atom 0))

(defn average-records []
  (doseq [record (fetch-records)]
    (swap! record-sum + (:score record)))
  (swap! record-count + 1))
(/ @record-sum @record-count))
```

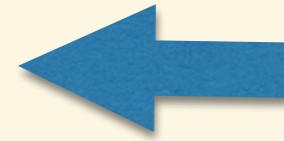
```
(def record-sum (atom 0))
(def record-count (atom 0))

(defn average-records []
  (doseq [record (fetch-records)]
    (swap! record-sum + (:score record)))
  (swap! record-count + 1))
(/ @record-sum @record-count))
```

```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



There are a lot of records, so this calculation takes a long time. We want to get the **current average** of all records done so far before it finishes. Can we express that?

```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
  (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



There are a lot of records, so this calculation takes a long time. We want to get the **current average** of all records done so far before it finishes. Can we express that?

```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
  (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



No :(

There are a lot of records, so this calculation takes a long time. We want to get the **current average** of all records done so far before it finishes. Can we express that?

```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



Current average: $754/100 = 7.54$

Read average: ???/100 = ???

There are a lot of records, so this calculation takes a long time. We want to get the **current average** of all records done so far before it finishes. Can we express that?

```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```

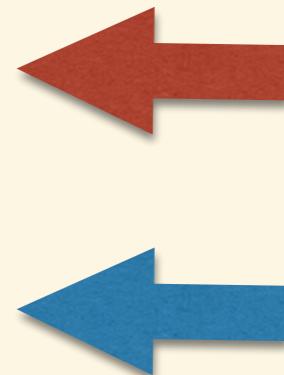


Current average: $754/100 = 7.54$

Read average: ???/100 = ???

There are a lot of records, so this calculation takes a long time. We want to get the **current average** of all records done so far before it finishes. Can we express that?

```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



Current average: $754/100 = 7.54$; $801/101 = 7.93$

Read average: $801/100 = 8.01$

Problem:
“In-between” state.

Solution:

Make mutation atomic.

```
(def record-sum (atom 0))  
(def record-count (atom 0))  
  
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record)))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



```
(def record-average (atom {:sum 0  
                           :count 0}))
```

```
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



```
(def record-average (atom {:sum 0  
                           :count 0}))
```

```
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-sum + (:score record))  
    (swap! record-count + 1))  
  (/ @record-sum @record-count))
```



```
(def record-average (atom {:sum 0  
                           :count 0}))
```

```
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-average  
           (fn [{:keys [sum count]}]  
             {:sum (+ sum (:score record))  
              :count (+ count 1)})))  
  (/ @record-sum @record-count))
```



```
(def record-average (atom {:sum 0  
                           :count 0}))
```

```
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-average  
           (fn [{:keys [sum count]}]  
             {:sum (+ sum (:score record))  
              :count (+ count 1)})))  
  (/ @record-sum @record-count))
```



```
(def record-average (atom {:sum 0  
                           :count 0}))
```



```
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-average  
           (fn [{:keys [sum count]}]  
             {:sum (+ sum (:score record))  
              :count (+ count 1)})))  
  (let [{:keys [sum count]} @record-average]  
    (/ sum count)))
```

There are a lot of sets of records that we want to calculate at the same time in different threads. [Can we express that?](#)

```
(def record-average (atom {:sum 0  
                           :count 0}))
```



```
(defn average-records []  
  (doseq [record (fetch-records)]  
    (swap! record-average  
      (fn [{:keys [sum count]}]  
        {:sum (+ sum (:score record))  
         :count (+ count 1)})))  
  (let [{:keys [sum count]} @record-average]  
    (/ sum count)))
```

There are a lot of sets of records that we want to calculate at the same time in different threads. Can we express that?

```
(def record-average (atom {:sum 0  
                           :count 0}))
```



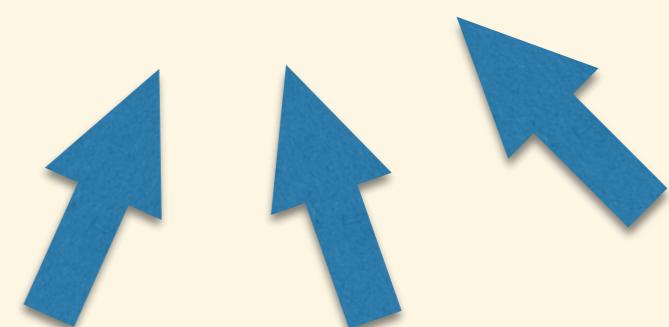
```
(defn average-records [id]  
  (doseq [record (fetch-records id)]  
    (swap! record-average  
           (fn [{:keys [sum count]}]  
             {:sum (+ sum (:score record))  
              :count (+ count 1)})))  
  (let [{:keys [sum count]} @record-average]  
    (/ sum count)))
```

There are a lot of sets of records that we want to calculate at the same time in different threads. [Can we express that?](#)

```
(def record-average (atom {:sum 0  
                           :count 0}))
```



```
(defn average-records [id]  
  (doseq [record (fetch-records id)]  
    (swap! record-average  
           (fn [{:keys [sum count]}]  
             {:sum (+ sum (:score record))  
              :count (+ count 1)})))  
  (let [{:keys [sum count]} @record-average]  
    (/ sum count)))
```



Problem:

**Threads will write over each
other in global state.**

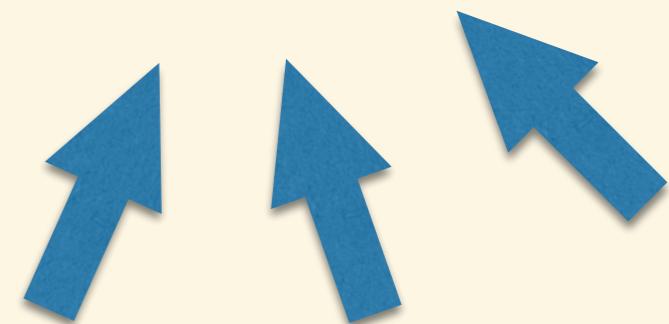
Solution:
Make state local.

There are a lot of sets of records that we want to calculate at the same time in different threads. [Can we express that?](#)

```
(def record-average (atom {:sum 0  
                           :count 0}))
```

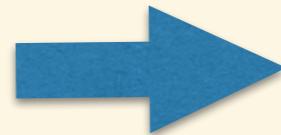


```
(defn average-records [id]  
  (doseq [record (fetch-records id)]  
    (swap! record-average  
           (fn [{:keys [sum count]}]  
             {:sum (+ sum (:score record))  
              :count (+ count 1)})))  
  (let [{:keys [sum count]} @record-average]  
    (/ sum count)))
```



There are a lot of sets of records that we want to calculate at the same time in different threads. Can we express that?

```
(def record-average (atom {:sum 0  
                           :count 0}))
```

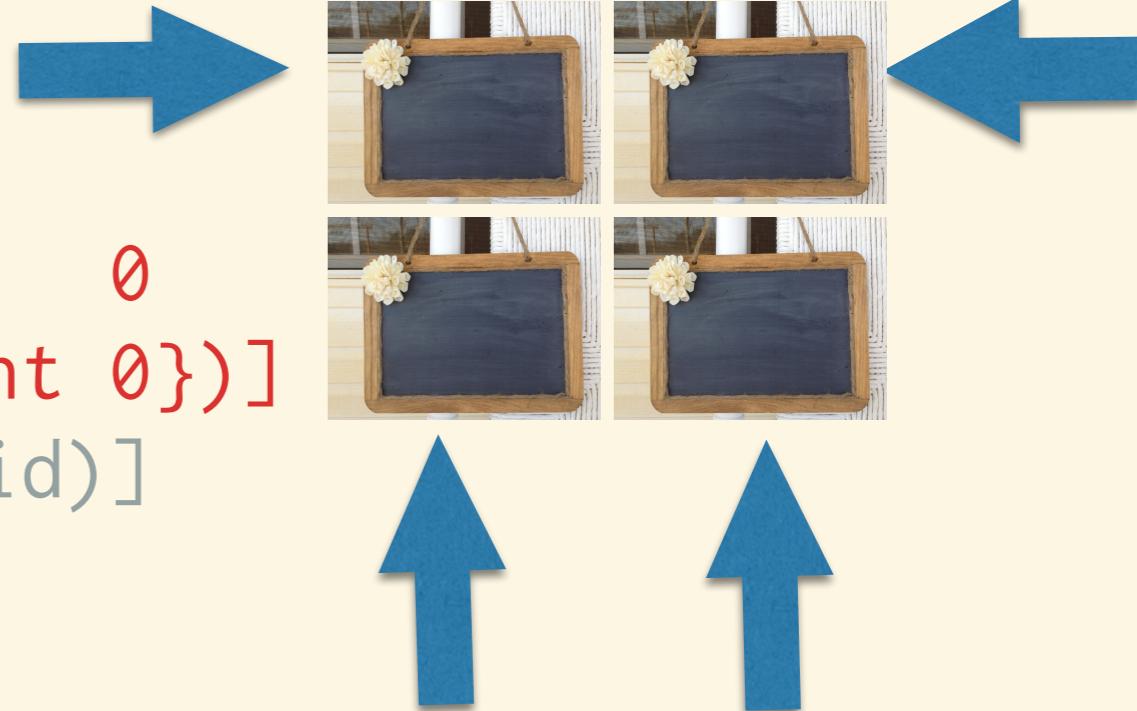


```
(defn average-records [id]  
  (doseq [record (fetch-records id)]  
    (swap! record-average  
           (fn [{:keys [sum count]}]  
             {:sum (+ sum (:score record))  
              :count (+ count 1)})))  
  (let [{:keys [sum count]} @record-average]  
    (/ sum count)))
```



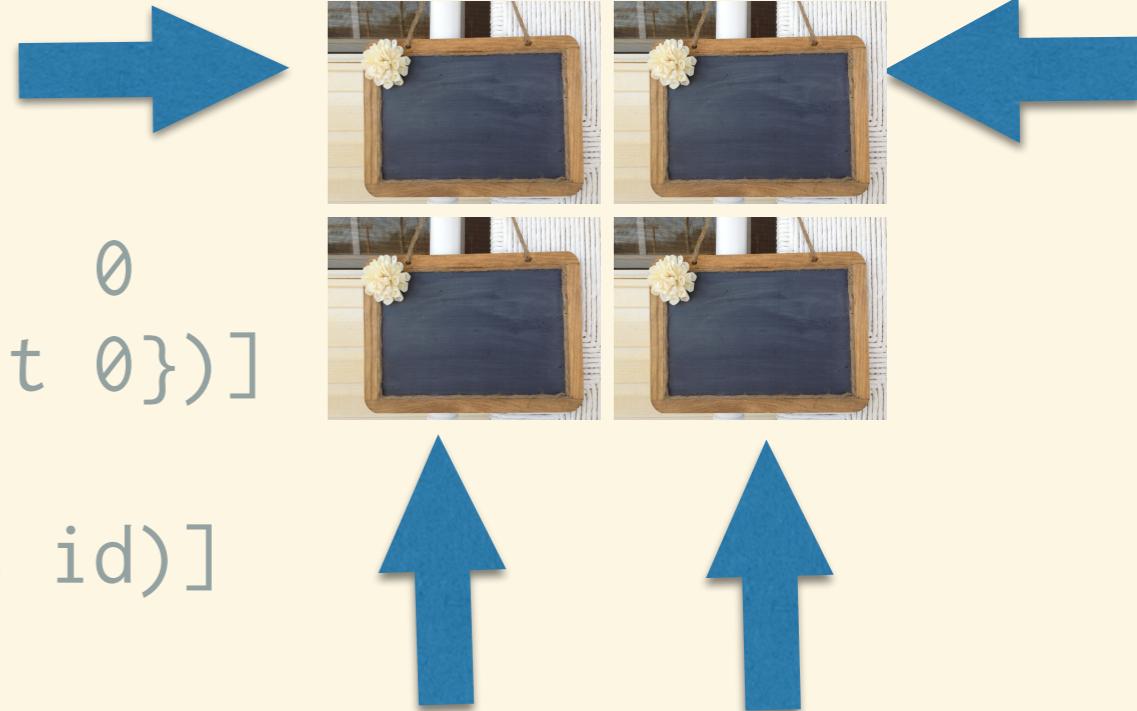
There are a lot of sets of records that we want to calculate at the same time in different threads. Can we express that?

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0})]
    (doseq [record (fetch-records id)]
      (swap! record-average
             (fn [{:keys [sum count]}]
               {:sum (+ sum (:score record))
                :count (+ count 1)})))
    (let [{:keys [sum count]} @record-average]
      (/ sum count))))
```



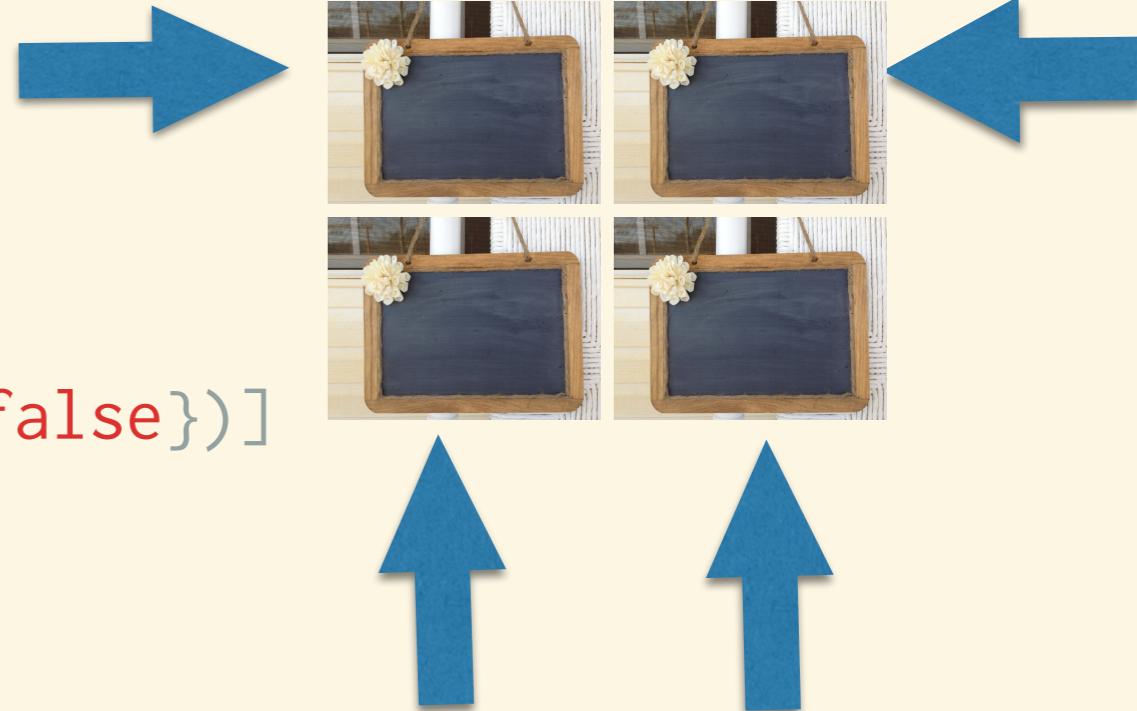
There are a lot of sets of records that we want to calculate at the same time in different threads. Can we express that?

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0})])
  (future
    (doseq [record (fetch-records id)]
      (swap! record-average
        (fn [{:keys [sum count]}]
          {:sum (+ sum (:score record))
           :count (+ count 1)}))))
  (fn []
    (let [{:keys [sum count]} @record-average]
      (/ sum count)))))
```



There are a lot of sets of records that we want to calculate at the same time in different threads. Can we express that?

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future
      (doseq [record (fetch-records id)]
        (swap! record-average
          (fn [{:keys [sum count]}]
            {:sum (+ sum (:score record))
             :count (+ count 1)
             :finished false})))
      (swap! record-average assoc :finished true)))
  (fn []
    (let [{:keys [sum count finished]} @record-average]
      [(/ sum count) finished]))))
```



```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future
      (doseq [record (fetch-records id)]
        (swap! record-average
          (fn [{:keys [sum count]}]
            {:sum (+ sum (:score record))
             :count (+ count 1)
             :finished false})))
      (swap! record-average assoc :finished true)))
  (fn []
    (let [{:keys [sum count finished]} @record-average]
      [(/ sum count) finished]))))
```

state:

non-atomic -> atomic

global -> local



more meaningful
more precise
more general

```
(defn average-records [id]
  (let [record-average (atom {:sum    0
                               :count  0
                               :finished false})]
    (future
      (doseq [record (fetch-records id)]
        (swap! record-average
          (fn [{:keys [sum count]}]
            {:sum    (+ sum (:score record))
             :count  (+ count 1)})))
      (swap! record-average assoc :finished true)))
  (fn []
    (let [{:keys [sum count finished]}
          @record-average]
      [(/ sum count) finished]))))
```

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future
      (doseq [record (fetch-records id)]
        (swap! record-average
          (fn [{:keys [sum count]}]
            {:sum (+ sum (:score record))
             :count (+ count 1)})))
      (swap! record-average assoc :finished true)))
  (fn []
    (let [{:keys [sum count finished]} @record-average]
      [(/ sum count) finished]))))
```

```
(defn accumulate-average [record-average id]
  (doseq [record (fetch-records id)]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))
```

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average id)))
  (fn []
    (let [{:keys [sum count finished]}]
        @record-average]
      [(/ sum count) finished]))))
```

```
(defn accumulate-average [record-average id]
  (doseq [record (fetch-records id)]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))
```

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average id)))
  (fn []
    (let [{:keys [sum count finished]}]
        @record-average]
      [(/ sum count) finished]))))
```

```
(defn accumulate-average [record-average id]
  (doseq [record (fetch-records id)]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average id))
    (fn [] (calculate-average @record-average))))
```

Some sets of records are in the Database (fetched by fetch-records). But some are stored in memory. We want to calculate the average of both types. [Can we express that?](#)

```
(defn accumulate-average [record-average id]
  (doseq [record (fetch-records id)]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average id))
    (fn [] (calculate-average @record-average))))
```

Some sets of records are in the Database (fetched by fetch-records). But some are stored in memory. We want to calculate the average of both types. Can we express that?

```
(defn accumulate-average [record-average id]
  (doseq [record (fetch-records id)] ←
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average id))
    (fn [] (calculate-average @record-average))))
```

Problem:
Side effect “buried” in
logic.

Solution:
**Separate side effect, call it
elsewhere, and pass result as
argument.**

Some sets of records are in the Database (fetched by fetch-records). But some are stored in memory. We want to calculate the average of both types. Can we express that?

```
(defn accumulate-average [record-average id]
  (doseq [record (fetch-records id)] ←
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average-records [id]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average id))
    (fn [] (calculate-average @record-average))))
```

Some sets of records are in the Database (fetched by fetch-records). But some are stored in memory. We want to calculate the average of both types. Can we express that?

```
(defn accumulate-average [record-average records]
```

```
  (doseq [record records]
```

```
    (swap! record-average
```

```
      (fn [{:keys [sum count]}]
```

```
        {:sum (+ sum (:score record))}
```

```
        :count (+ count 1)
```

```
        :finished false}))
```

```
  (swap! record-average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
```

```
  [(/ sum count) finished])
```

```
(defn average-records [id]
```

```
  (let [record-average (atom {:sum 0
```

```
                           :count 0
```

```
                           :finished false})]
```

```
  (future (accumulate-average record-average (fetch-records id)))
```

```
  (fn [] (calculate-average @record-average))))
```

Some sets of records are in the Database (fetched by fetch-records). But some are stored in memory. We want to calculate the average of both types. Can we express that?

```
(defn accumulate-average [record-average records]
  (doseq [record records]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average-records [records]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average records))
    (fn [] (calculate-average @record-average))))
```

side effects:

buried -> separated

```
(defn accumulate-average [record-average records]
  (doseq [record records]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average-records [records]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average records))
    (fn [] (calculate-average @record-average))))
```

We are calculating the average score, but now we need to find the average age. Can we express that?

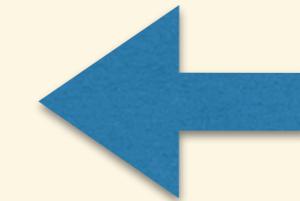
```
(defn accumulate-average [record-average records]
  (doseq [record records]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average-records [records]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average records))
    (fn [] (calculate-average @record-average))))
```

We are calculating the average score, but now we need to find the average age. Can we express that?

```
(defn accumulate-average [record-average records]
  (doseq [record records]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (:score record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))
```



```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average-records [records]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average records))
    (fn [] (calculate-average @record-average))))
```

Problem:
**Our function depends on
internal structure of data.**

Solution:
**Abstract the structure using
a fn argument.**

We are calculating the average score, but now we need to find the average age. Can we express that?

```
(defn accumulate-average [record-average f records]
  (doseq [record records]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (f record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average-records [f records]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average f records))
    (fn [] (calculate-average @record-average))))
```

We are calculating the average score, but now we need to find the average age. Can we express that?

```
(defn accumulate-average [record-average f records]
  (doseq [record records]
    (swap! record-average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (f record))
         :count (+ count 1)
         :finished false})))
  (swap! record-average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average-records [f records]
  (let [record-average (atom {:sum 0
                               :count 0
                               :finished false})]
    (future (accumulate-average record-average f records))
    (fn [] (calculate-average @record-average))))
```

We are calculating the average score, but now we need to find the average age. Can we express that?

```
(defn accumulate-average [average f vals]
  (doseq [val vals]
    (swap! average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (f val))
         :count (+ count 1)
         :finished false})))
  (swap! average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    (future (accumulate-average average f vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate-average [average f vals]
  (doseq [val vals]
    (swap! average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (f val))
         :count (+ count 1)
         :finished false})))
  (swap! average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    (future (accumulate-average average f vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate-average [average f vals]
  (doseq [val vals]
    (swap! average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (f val))
         :count (+ count 1)
         :finished false})))
  (swap! average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    (future (accumulate-average average f vals))
    (fn [] (calculate-average @record-average))))
```

Problem:

**What we calculate is buried
in how we calculate it.**

Solution:
**Dig out structure into one
place.**

Now we need to calculate the sum. Can we express that?

```
(defn accumulate-average [average f vals]
  (doseq [val vals]
    (swap! average
      (fn [{:keys [sum count]}]
        {:sum (+ sum (f val))
         :count (+ count 1)
         :finished false})))
  (swap! average assoc :finished true))
```

```
(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])
```

```
(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    (future (accumulate-average average f vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate-average [accum average f vals]
  (doseq [val vals]
    (accum val))
  (swap! average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})
        accum (fn [val]
                (swap! average
                  (fn [{:keys [sum count]}]
                    {:sum (+ sum (f val))
                     :count (+ count 1)
                     :finished false})))
        future (accumulate-average accum average f vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate-average [accum average f vals]
  (doseq [val vals]
    (accum val))
  (swap! average assoc :finished true))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})
        accum (fn [val]
                (swap! average
                  (fn [{:keys [sum count]}]
                    {:sum (+ sum (f val))
                     :count (+ count 1)
                     :finished false})))
        future (future (accumulate-average accum average f vals))
        fn [] (calculate-average @record-average)))]
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate-average [accum finish average f vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})
        accum (fn [val]
                (swap! average
                      (fn [{:keys [sum count]}]
                        {:sum (+ sum (f val))
                         :count (+ count 1)
                         :finished false})))
        finish (fn [] (swap! average assoc :finished true))]
    (future (accumulate-average accum finish average f vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate-average [accum finish average f vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})
        accum (fn [val]
                (swap! average
                      (fn [{:keys [sum count]}]
                        {:sum (+ sum (f val))
                         :count (+ count 1)
                         :finished false})))
        finish (fn [] (swap! average assoc :finished true))]
    (future (accumulate-average accum finish average f vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})
        accum (fn [val]
                (swap! average
                      (fn [{:keys [sum count]}]
                        {:sum (+ sum (f val))
                         :count (+ count 1)
                         :finished false})))
        finish (fn [] (swap! average assoc :finished true))]
    (future (accumulate accum finish vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn calculate-average [{:keys [sum count finished]}]
  [(/ sum count) finished])

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})
        accum (fn [val]
                (swap! average
                      (fn [{:keys [sum count]}]
                        {:sum (+ sum (f val))
                         :count (+ count 1)
                         :finished false})))
        finish (fn [] (swap! average assoc :finished true))]
    (future (accumulate accum finish vals))
    (fn [] (calculate-average @record-average))))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})}
    accum (fn [val]
            (swap! average
                  (fn [{:keys [sum count]}]
                    {:sum (+ sum (f val))
                     :count (+ count 1)
                     :finished false})))
    finish (fn [] (swap! average assoc :finished true))
    current (fn []
              (let [{:keys [sum count finished]} @average]
                [(/ sum count) finished]))
    (future (accumulate accum finish vals))
    current)))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn average [f vals]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    (fn [val]
      (swap! average
             (fn [{:keys [sum count]}]
               {:sum (+ sum (f val))
                :count (+ count 1)
                :finished false})))
    (fn [] (swap! average assoc :finished true))
    (fn []
      (let [{:keys [sum count finished]} @average]
        [(/ sum count) finished]))]
  (future (accumulate accum finish vals))
  current))
```



Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn average-accumulator [f]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    [(fn [val]
       (swap! average
              (fn [{:keys [sum count]}]
                {:sum (+ sum (f val))
                 :count (+ count 1)
                 :finished false})))
     (fn [] (swap! average assoc :finished true))]
    (fn []
      (let [{:keys [sum count finished]} @average
            [(/ sum count) finished]])))))

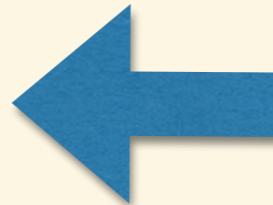
(defn average [f vals]
  (let [[accum finish current] (average-accumulator f)]
    (future (accumulate accum finish vals))
    current))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn average-accumulator [f]
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    [(fn [val]
       (swap! average
              (fn [{:keys [sum count]}]
                {:sum (+ sum (f val))
                 :count (+ count 1)
                 :finished false})))
     (fn [] (swap! average assoc :finished true))]
    (fn []
      (let [{:keys [sum count finished]} @average
            [(/ sum count) finished]])))))

(defn average [f vals]
  (let [[accum finish current] (average-accumulator f)]
    (future (accumulate accum finish vals))
    current))
```



Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))
```

```
(defn average-accumulator []
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    [(fn [val]
       (swap! average
              (fn [{:keys [sum count]}]
                {:sum (+ sum val)
                 :count (+ count 1)
                 :finished false})))
     (fn [] (swap! average assoc :finished true))
     (fn []
       (let [{:keys [sum count finished]}]
         @average
         [(/ sum count) finished]))]))
```

```
(defn average [f vals]
  (let [[accum finish current] (average-accumulator)]
    (future (accumulate (comp accum f) finish vals))
    current))
```

```
(average :score (fetch-records 10))
(average :age (fetch-records 10))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))

(defn average-accumulator []
  (let [average (atom {:sum 0
                        :count 0
                        :finished false})]
    [(fn [val]
       (swap! average
              (fn [{:keys [sum count]}]
                {:sum (+ sum val)
                 :count (+ count 1)
                 :finished false})))
     (fn [] (swap! average assoc :finished true))
     (fn []
       (let [{:keys [sum count finished]}]
         @average
         [(/ sum count) finished]))])))

(defn average [f vals]
  (let [[accum finish current] (average-accumulator)]
    (future (accumulate (comp accum f) finish vals))
    current))
```

Now we need to calculate the sum. Can we express that?

```
(defn accumulate [accum finish vals]
  (doseq [val vals]
    (accum val))
  (finish))
```

```
(defn sum-accumulator []
  (let [average (atom {:sum 0
                        :finished false})]
    [(fn [val]
       (swap! average
              (fn [{:keys [sum]}]
                {:sum (+ sum val)
                 :finished false})))
     (fn [] (swap! average assoc :finished true))
     (fn []
       (let [{:keys [sum finished]}]
         @average
         [sum finished]))]))
```

```
(defn sum [f vals]
  (let [[accum finish current] (sum-accumulator)]
    (future (accumulate (comp accum f) finish vals))
    current))
```

(average :score (fetch-records 10))

(average :age (fetch-records 10))

(sum :score (fetch-records 12))

dependencies:

concrete structure -> abstraction

disparate structure -> consolidated

purelyfunctional.tv/clojure-remote

clojuregazette.com

lispcast.com