Reducers

Rich Hickey

Motivation

- Performance
  - via reduced allocation (vs seqs)
  - via parallelism (leverage fork/join)
- Computer clock speeds are stuck
- Matters

Inspiration

- Haskell Iteratees
  - http://www.haskell.org/haskellwiki/Enumerator_and_iteratee
- Guy Steele's ICFP 2009 Talk
  - Organizing Functional Code for Parallel Execution or, foldl and foldr Considered Slightly Harmful
  - http://vimeo.com/6624203

Where We Are

- FP History
  - Primacy of lists and recursion
- Clojure has seqs and laziness
- *Inherently* sequential

Where We are Going

- More cores
- Speed *must* come from parallelism
- New programming model required?

Model Evolution

- Loops
- Higher-order functions on lists
- HOFs on Collections
- Collection independence
- Order independence

map et al Do Too Much

```
(defn map [f coll]
  (cons (f (first coll)) (map f (rest coll))))
```

- Recursion
- Order
- Laziness
- Consumes List
- Builds list

reduce Lets Collection Drive

```
(defn reduce
  ([f init coll]
    (clojure.core.protocols/coll-reduce coll f init)))
```

- Ignorant of collection structure
- Can build anything
- Not lazy
- Still ordered, left fold with seed

Reducing Function

- (f result input) -> result
- Applied to init + first value
- then result + second value etc

How To Make map et al Collection Ignorant?

- Build on reduce
- *Without* depending on order
  - because map/filter don’t, fundamentally
- What to build? - *Nothing!*

Reduction Transformers

- Instead of making new concrete collection
- Change what reduce means for collection
- By modifying the supplied reducing function

Transformers

```
(defn mapping [f]
  (fn [f1]
    (fn [result input]
      (f1 result (f input)))))

(defn filtering [pred]
  (fn [f1]
    (fn [result input]
      (if (pred input)
        (f1 result input)
        result))))

(defn mapcatting [f]
  (fn [f1]
    (fn [result input]
      (reduce f1 result (f input)))))
```

Transformers

```
(defn mapping [f]
  (fn [f1]
    (fn [result input]
      (f1 result (f input)))))

(defn filtering [pred]
  (fn [f1]
    (fn [result input]
      (if (pred input)
        (f1 result input)
        result))))

(defn mapcatting [f]
  (fn [f1]
    (fn [result input]
      (reduce f1 result (f input)))))
```

Reducers

```
(reduce ((mapping inc) +) 0 [1 2 3 4]) ;meh
```

- We want fn of collection -> collection
- Minimize definition of collection == *reducible*

```
(defn reducer
      ([coll xf]
       (reify
        CollReduce
        (coll-reduce [_ f1 init]
          (coll-reduce coll (xf f1) init)))))

(reduce + 0 (reducer [1 2 3 4] (mapping inc))
```

Same Model

```
(defn rmap [f coll]
  (reducer coll (mapping f)))

(defn rfilter [pred coll]
  (reducer coll (filtering pred)))

(defn rmapcat [f coll]
  (reducer coll (mapcatting f)))

(reduce + 0 (rmap inc [1 2 3 4]))
;=> 14
(reduce + 0 (rfilter even? [1 2 3 4]))
;=> 6
(reduce + 0 (rmapcat range [1 2 3 4 5]))
;=> 20
```

But...

- reduce still sequential
- Some perf gains due to less allocation
- Where's the cake?

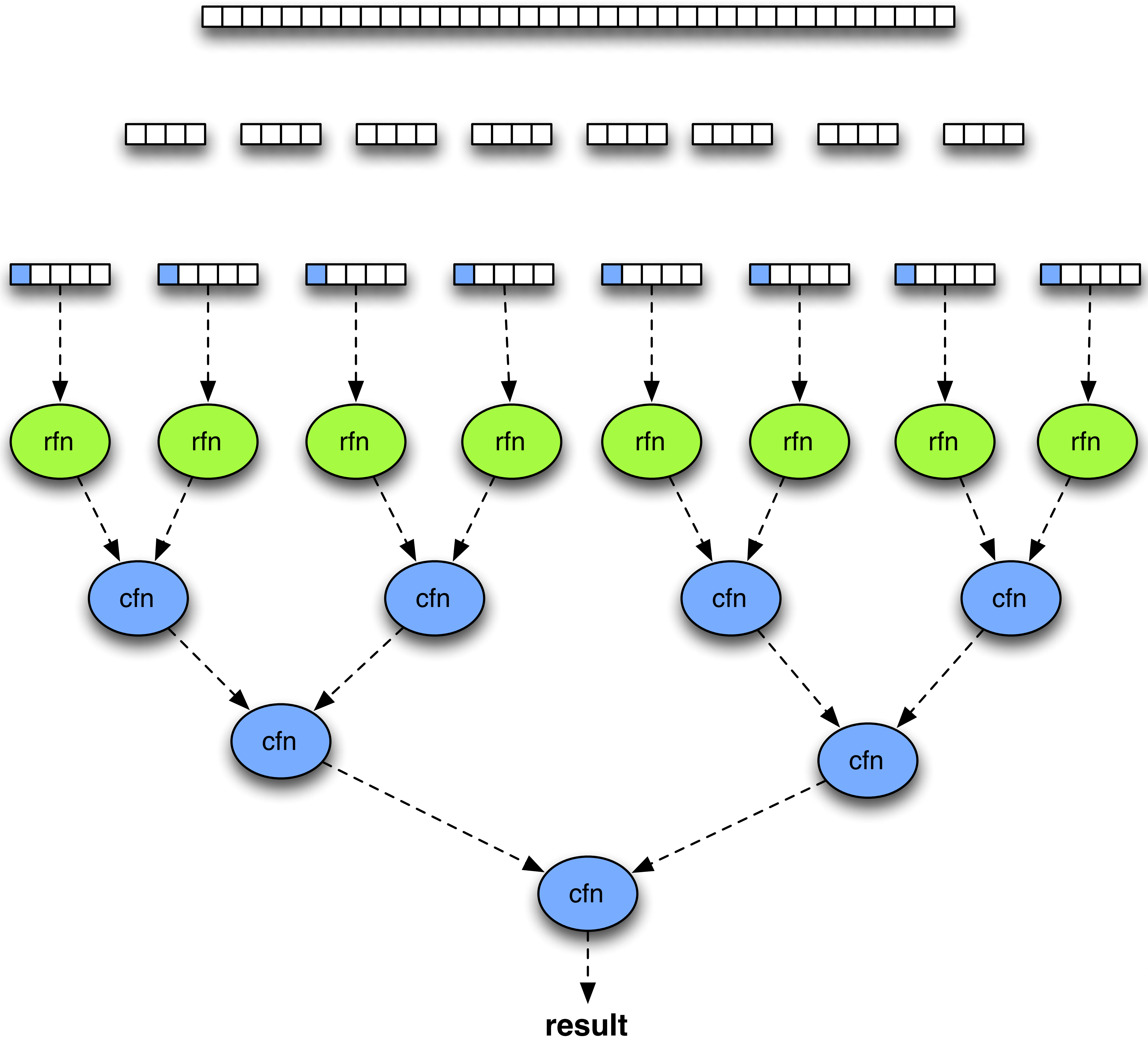
fold

- Takes the order out of foldl, foldr, reduce
  - a (potentially) parallel reduction
- Uses a reduce+combine strategy
  - fork/join under the hood
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

fold

```
(defn fold
  ([combinef reducef coll]
     (coll-fold coll n combinef reducef)))
```

- Like reduce, asks collection to do the work
  - via protocol
- Segment the collection
- Run multiple reduces in parallel
- Use combine fn to process reduce results


rfn
rfn
rfn
rfn
rfn
rfn
rfn
rfn
cfn
cfn
cfn
cfn
cfn
cfn
cfn
result

Reducing Leaves

- Breaks free from ordered single-pass
- Multiple seeds - from where?
  - (combinef) ;; no args
  - must return ‘identity’
  - a la (+) == 0

Folders

- If collection is foldable, so is reducer
- as long as transformer doesn't care about order
  - map/filter etc don't, take does

```
(defn folder
  ([coll xf]
     (reify
      ;;extend CollReduce as before
      CollFold
      (coll-fold [_ n combinef reducef]
        (coll-fold coll n combinef (xf reducef))))))

(defn rmap [f coll]
  (folder coll (mapping f)))
```

Composition

```
(def transform (comp (r/map inc) (r/filter even?)))

(r/fold + (transform v))
```

reduce/combine vs. map/reduce

- No collection-ification
- identity value vs fn
- granularity

Summary

- Build map, filter et al as reducers
- Now independent of collection and order
- So, if fold is parallel, so are the ops
    - No parallel-collections
    - No parallel-ops
- fold + collection + reducers - simple!

Demo & Questions