

# Folding in Parallel

*manually*

notch1p

August 23, 2024

# fold $\{l,r\}$

- **foldl**:  $(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$
- **foldr**:  $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha \rightarrow [\alpha] \rightarrow \beta$

Examples:

$$\text{foldl } (\cdot + \cdot) 0 \iota.4 = 10$$

$$\text{foldr } \dots = 10$$

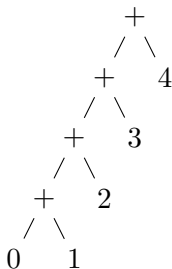
## how do they look?

foldl  $(\cdot + \cdot)$  0  $\iota$ .4

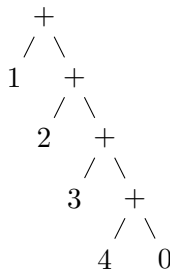
$$\iff (((0 + 1) + 2) + 3) + 4$$

foldr ...

$$\iff 1 + (2 + (3 + (4 + 0)))$$



(a) foldl



(b) foldr

# Sequential BAD

Compare:

|   |                         |                   |                             |
|---|-------------------------|-------------------|-----------------------------|
| ① | $((0 + 1) + 2) + 3 + 4$ | <b>sequential</b> | $O(\log n)$                 |
| ② | $(0 + 1) + (2 + 3 + 4)$ | <b>parallel</b>   | $\Omega(\log n), O(\log n)$ |

In other words, we would like to insert  $+$  between elements.

Languages like APL/J already do this:

**(+ / % #)** 1 2 3 4 5 *NB. 3. uses implicit fork.*

Consider a more general case:

$$((a \text{ op } b) \text{ op } c) \text{ op } d \stackrel{?}{=} (a \text{ op } b) \text{ op } (c \text{ op } d)$$

When does the equation hold?

# monoid

$\text{op} : S \rightarrow S \rightarrow S$  must satisfy  $\forall a, b, c, e \in S$ ,

$$(a \text{ op } b) \text{ op } c = a \text{ op } (b \text{ op } c),$$

Associativity

$$a \text{ op } i = i \text{ op } a = a.$$

Identity

- Monoid: A (carrier) set with an associative binary operation  $\text{op}$  and a unit element.

# reduce

In other words,

```
class Monoid ( $\alpha$ : Type) where
  zero:  $\alpha$ 
  op:  $\alpha \rightarrow \alpha \rightarrow \alpha$ 
```

e.g. for +,

```
instance m_nat_add : Monoid Nat := ⟨0, (· + ·)⟩
```

reduce: A fold-like operation that reduces over a monoid. We expect

$$\begin{aligned} \text{reduce } :: \alpha &\Rightarrow \text{Monoid } \alpha \rightarrow [\alpha] \rightarrow \alpha, \\ \text{reduce } m \text{ nil} &\equiv m.\text{zero}, \\ \text{reduce } m [x] &\equiv [x]. \end{aligned}$$

Then summing over  $\iota.4$  would be

$$\text{reduce } \langle 0, (\cdot + \cdot) \rangle [1, 2, 3, 4] \equiv 1 + 2 + 3 + 4$$

+ in some languages (e.g. CL) is already Monoidic and their implementation of reduce takes advantages from it.

Sequential version of reduce:

```
def reduce [m: Monoid  $\alpha$ ] (xs: List  $\alpha$ ):  $\alpha$  :=
match xs with
| []  $\Rightarrow$  Monoid.zero
| [x]  $\Rightarrow$  x
| x::xs  $\Rightarrow$  Monoid.op x (reduce xs)
```

How about parallel? Split list to smaller list:

```
class ListSlice ( $\alpha$  : Type) where
l: List  $\alpha$ 
start: Nat
finish: Nat
```

# parallel reduce

Parallel:

```
partial def parreduce [Inhabited  $\alpha$ ] (m : Monoid  $\alpha$ ) (xs : ListSli
match xs.finish + 1 - xs.start with
| 0  $\Rightarrow$  m.zero
| 1  $\Rightarrow$  xs.l.get! xs.start
| 2  $\Rightarrow$  m.op (xs.l.get! xs.start) (xs.l.get! (xs.start + 1))
| 3  $\Rightarrow$ 
m.op
(m.op (xs.l.get! xs.start) (xs.l.get! (xs.start + 1)))
(xs.l.get! (xs.start + 2))
| n + 4  $\Rightarrow$ 
let n' := (n + 4) / 2
let first_half := {xs with finish := xs.start + n' - 1}
let second_half := {xs with start := xs.start + n'}
m.op
(parreduce m first_half)
(parreduce m second_half)
```

**No data dependency i.e. Invocations can be done in parallel.**



# compose monoid

Consider `(foldr #'- 0 (iota 4)) ; => ((1- (2- (3- (4- x)))) 0)`,  
`(n-)` can be seen as a function. (CL does have `1- 1+`) Or generally,

$$\text{foldr } (n-) \ z \ l \ \iota.n = (n-)^{\circ n} \ z$$

- how about constructing monoid from function composition...

Obviously,

$$(f \circ g) \circ h = f \circ (g \circ h)$$

$$\text{id} \circ f = f \circ \text{id} = f$$

Thus we obtain

**instance** `compose_monoid` : Monoid `(α -> α)` := `<id, λ f g x => f (g x)>`

Key idea: `o` is associative.

But how do we make  $(n-)$ , or generally, a bivariate function with its lvalue pre-filled?

- *Partial Application*. Very easy in a curried language.

Now `foldr` would be

```
def foldr (f:  $\alpha \rightarrow \beta \rightarrow \beta$ ) (init:  $\beta$ ) (xs: List  $\alpha$ ):  $\beta$  :=
  List.map f xs ▷ reduce compose_monoid ◁ init
```

`foldl` is tricky:

```
(foldl #'- 0 (iota 4)) ;  $\Rightarrow ((-4 (-3 (-2 (-1 x)))) 0)$ .
```

since it's  $(f \text{ init } xs\_i)$  instead of  $(f \text{ } xs\_i \text{ init})$ . Meaning we'll pre-fill rvalue without evaluating the whole call.

```
def fold_left (f:  $\alpha \rightarrow \beta \rightarrow \alpha$ ) (init:  $\alpha$ ) (xs : List  $\beta$ ):  $\alpha$  :=
  xs.map ( $\lambda x \Rightarrow \lambda \text{init} \Rightarrow f \text{ init } x$ ) ▷ reduce compose_monoid ◁ i
```

- A practical implementation of `mapReduce` is to fuse `map` and `reduce` together. Much efficient than what we have now.
- We write them separately for sake of clarity.

# Performance: 🍰

A length of  $n$  list yields a composition of  $n$  closures.

A closure takes up several words of heap space.

Heap be like: 💀

To do this efficiently:

- factor out the folding function  $f$  in terms of

$$f\ z\ l = op\ z\ (g\ l)$$

- requires ingenuity