Folding in Parallel

manually

notch1p

August 23, 2024

fold{l,r}

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

Examples:

$$\begin{aligned} \textsf{foldl}\ (\cdot + \cdot)\ 0\ \iota.4 &= 10 \\ \textsf{foldr}\quad \ldots &= 10 \end{aligned}$$

how do they look?

$$\text{foldl}\ (\cdot + \cdot)\ 0\ \iota.4 \iff (((0+1)+2)+3)+4$$

$$\text{foldr}\quad \ldots \iff 1+(2+(3+(4+0)))$$

```
        +
       / \
      +   4
     / \
    +   3
   / \
  +   2
 / \
0   1
```

(a) foldl

```
  +
 / \
1   +
   / \
  2   +
     / \
    3   +
       / \
      4   0
```

(b) foldr

Sequential BAD

Compare:

1. $(((0+1)+2)+3)+4$ **sequential** $O(\log n)$
2. $(0+1)+(2+3+4)$ **parallel** $\Omega(\log n)$, $O(\log n)$

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:

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

When does the equation hold?

monoid

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

$$(a \,\mathsf{op}\, b) \,\mathsf{op}\, c = a \,\mathsf{op} (b \,\mathsf{op}\, c), \qquad \text{Associativity}$$
$$a \,\mathsf{op}\, i = i \,\mathsf{op}\, a = a. \qquad \text{Identity}$$

- Monoid: A (carrier) set with an associative binary operation op and a unit element.

reduce

In other words,

```
class Monoid (α: Type) where
zero: α
op: α -> α -> α
```

e.g. for +,

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

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

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

Then summing over $\iota.4$ would be

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

+ 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 α] (xs: List α): α :=
match xs with
| [] ⇒ Monoid.zero
| [x] ⇒ x
| x::xs ⇒ Monoid.op x (reduce xs)
```

How about parallel? Split list to smaller list:

```
class ListSlice (α : Type) where
l: List α
start: Nat
finish: Nat
```

parallel reduce

Parallel:

```
partial def parreduce [Inhabited α] (m : Monoid α) (xs : ListSli
match xs.finish + 1 - xs.start with
| 0 ⇒ m.zero
| 1 ⇒ xs.l.get! xs.start
| 2 ⇒ m.op (xs.l.get! xs.start) (xs.l.get! (xs.start + 1))
| 3 ⇒
m.op
(m.op (xs.l.get! xs.start) (xs.l.get! (xs.start + 1)))
(xs.l.get! (xs.start + 2))
| n + 4 ⇒
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,

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

- how about constructing monoid from function composition...

Obviously,

$$(f \circ g) \circ h = f \circ (g \circ h)$$
$$\mathrm{id} \circ f = f \circ \mathrm{id} = f$$

Thus we obtain

```
instance compose_monoid : Monoid (α -> α) := ⟨id, λ f g x ⇒ f (g x)⟩
```

Key idea: $\circ$ 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: α -> β -> β) (init: β) (xs: List α): β :=
List.map f xs ▷ reduce compose_monoid ◁ init
```

foldl is tricky:

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

since it's `(f init xs_i)` instead of `(f xs_i init)`. Meaning we'll pre-fill rvalue without evaluating the whole call.

```
def fold_left (f: α -> β -> α) (init: α) (xs : List β): α :=
xs.map (λ x ⇒ λ init ⇒ f 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