Folding in Parallel

manually

notch1p

August 27, 2024

fold{l,r}

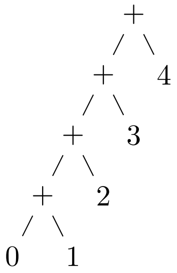
- foldl: $(\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$
- foldr: $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$

Examples:

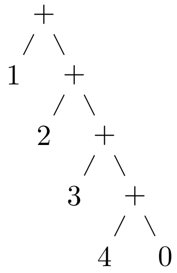
$$\begin{aligned} \textsf{foldl}\ (\cdot + \cdot)\ 0\ \iota.4 &= 10 \\ \textsf{foldr}\quad \cdots &= 10 \end{aligned}$$

how do they look?

$$\begin{aligned} &\mathsf{foldl}\ (\cdot + \cdot)\ 0\ \iota.4 && \iff (((0+1)+2)+3)+4 \\ &\mathsf{foldr}\quad \cdots && \iff 1+(2+(3+(4+0))) \end{aligned}$$



(a) foldl



(b) foldr

Sequential BAD

Compare:

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

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:

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

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

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

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:

```
def parreduce [Inhabited α] (m : Monoid α) (xs : ListSlice α) : α :=
    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: ∘ 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 ◁ init
```

- 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: 💀

folding, Efficiently

To do this efficiently:

- factor out the folding function f in terms of

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

- requires ingenuity

e.g. length of a list: `l.foldl (λ x _ ⇒ x + 1) 0`

With `mapReduce`, that is

```
l.map (Function.const Int 1) ▷ reduce ⟨0, (· + ·)⟩
```

where

- $\mathsf{op} = (+)$
- $g = (x : \mathsf{Int} \mapsto 1)$

Principle: Conjugate Transform

Guy Steele: the general principle/schema to transform a foldl is

$$\begin{aligned}\mathsf{foldl}\ (f : \alpha \to \beta \to \alpha)\ (z : \alpha)\ (l : \beta) = &\ \mathsf{map}\ (g : \beta \to \sigma)\ l \\ &\triangleright \mathsf{reduce}\ (m : \mathsf{Monoid}\ \sigma) \\ &\triangleright (h : \sigma \to \alpha)\end{aligned} \tag{1}$$

- g, h depends on f, z.
- σ shall be a "bigger" type that embeds α, β and there exists some associative operation and a unit element for it. In before we chose `compose_monoid` and $\alpha \to \alpha$ as type σ to obtain a generalized fold.

But how to find this σ, or broadly, how to find the corresponding monoid for f ?

example: subtract

$(+)$ is very nice. $(\mathbb{Z}, +)$ forms a abelian group. What about $(-)$:

- foldl $(-)\ 10\ \iota.4 = 10 - (1+2+3+4) = 10 -$ foldl $(+)\ 0\ \iota.4$
  thus foldl $(-)\ z\ l = z -$ reduce $\langle 0, (+)\rangle\ l$
- foldr...?

foldr $(-)\ z\ \iota.4 = 1 - (2 - (3 - (4 - z))) = 1 - 2 + 3 - 4 + z$

```
instance sub_monoid : Monoid (Int × Bool) where
    zero := (0, true)
    op := fun ⟨x₁, b₁⟩ ⟨x₂, b₂⟩ ⇒
        (if b₁ then x₁ + x₂ else x₁ - x₂, b₁ = b₂)

def int_foldr_sub (init: Int) (xs: List Int) : Int :=
    let fst :=
        xs.map (fun x: Int ⇒ (x, false))
            ▷ reduce sub_monoid ▷ Prod.fst
    if xs.length &&& 1 == 0 then init + fst else init - fst
```

example: Horner Rule

How do we parse ints:

```
s.foldl (fun acc c ⇒ acc * 10 + (c.toNat - '0'.toNat)) 0
```

that is, for a char sequence s, we have

$$\begin{aligned} \mathsf{parseInt}\ s &= \sum s_i \cdot r^i \text{ where } r = 10 \\ &= b_n \qquad \text{(Horner Rule)} \end{aligned}$$

where b is recursively defined:

$$\begin{aligned} b_0 &= 0 \cdot r && + s_0 \\ b_1 &= b_0 \cdot r && + s_1 \\ &\vdots \\ b_n &= b_{n-1} \cdot r && + s_n \end{aligned}$$

This recursive process is called horner rule.

We'll build a monoid for the (non-associative) $(a, c) \mapsto \text{a} \cdot 10 + c$ (suppose we've mapped the chars to its codepoint) Consider "071":

$$\mathsf{parseInt}\ 071 = (\underbrace{((0 \cdot 10 + 0) \cdot 10}_{a \cdot 10} + 7) \cdot 10 + 1$$

- $\mathsf{op} = x, y \mapsto x \cdot r' + y$ where r' could be 100, 1000, ...
  We need to track r':
- $\mathsf{op} = (x, b_1), (y, b_2) \mapsto (x \cdot b_2 + y, b_1 \cdot b_2)$. (easy to prove associative)
- has the unit $(0, 1)$ where $(x, b)\,\mathsf{op}(0, 1) = (0, 1)\,\mathsf{op}(x, b) = (x, b)$

Thus we obtain

```
instance horner_monoid: Monoid (Nat × Nat) :=
  ⟨(0,1), λ (x, r₁) (y, r₂) ⇒ (x * r₂ + y, r₁ * r₂)⟩
```

We denote left composition i.e. $f, g \mapsto (x \mapsto f\ x \triangleright g)$ as ⤳ for the sake of brevity:

```
def comp_left (f: α -> β) (g: β -> γ): α -> γ := (λ x ⇒ f x ▷ g)
infixl: 20 " ⤳ " ⇒ comp_left
```

And we get a parallel version of parseInt:
(much redundant cost here, but thats just a lean problem)

```
def parseInt_alt : String -> Nat :=
    String.toList
    ⤳ List.map (λ c ⇒ c.toNat - '0'.toNat)
    ⤳ List.map (λ x ⇒ (x, 10)) -- g
    ⤳ reduce horner_monoid
    ⤳ Prod.fst -- h
```

generalizing horner rule

What about a general version of `horner_monoid` i.e.

$$\forall f, \exists m\ (m : \mathsf{Monoid}, f : (\alpha \to \beta \to \alpha) \to f\ z\ x = m.\mathsf{op}\ (h\ z)\ x)$$

This is similiar to that in the last section as both involves composition.

```
instance hmonoid [Monoid α] : Monoid (α × (α -> α)) where
  zero := (Monoid.zero, id)
  op :=
    λ ⟨x₁, f₁⟩ ⟨x₂, f₂⟩ ⇒
      (Monoid.op (f₂ x₁) x₂, f₁ ⤳ f₂)
```

An efficient implementation will replace $\alpha \to \alpha$ with a value if possible. e.g. in `parseInt` f_1, f_2 is just $(\cdot \times 10)$. It can be represented by that 10 instead of a function; and the composition is represented by the product of which.

fin

Thank You

see Oleg Kiselyov's article,
Guy Steele's ICFP 2009 Talk