FIT2102
Programming Paradigms
Lecture 9

Folding and Traversing

Faculty of Information Technology

MONASH University

Learning Outcomes

- Compare folds in haskell to reduce in JavaScript
- Compare left and right folds
- Apply folds to compute aggregates over lists and other data
- Apply generalised folds to Foldable data types
- Apply folds to Foldables of Monoids
- Apply traverse to Traversables

JavaScript revisited
Reducing reduce redux

```
function sum(a: number[]): number {
    let total = 0
    for (let i = 0; i < a.length; i++) {
        total += a[i]
    }
    return total
}
```

This is horrible code!

- Two mutable variables
- `total` referenced in three places
- `i` referenced in four places
- We have to know how the container works
- Takes five lines of code to implement an operation we learned in Kindergarten!
- Can't easily generalise it to operations other than +

Reducing reduce redux

```
function sum(a: number[]): number {
    return a.reduce((v, total)=> v + total, 0);
}
```

Better… but:

- Still two variables (that could be mutated)
- `reduce` is only defined for arrays...

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

function reduce(f,a,l) {
    if (l) {
        return reduce(f, f(a,head(l)), tail(l))
    } else {
        return a;
    }
};

console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

function reduce(f,a,l) {
   return l ?
                 reduce(f, f(a,head(l)), tail(l))
                :
                 a;


}

console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

function reduce(f,a,l) {
    return l ? reduce(f, f(a,head(l)), tail(l)) : a;
}
console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

const reduce = (f,a,l) =>
   l ? reduce(f, f(a,head(l)), tail(l)) : a

console.log(reduce((x,y)=>x+y, 0, aList))
> 6
```

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

const reduce = f => a => l =>
   l ? reduce(f)(f(a)(head(l)))(tail(l)) : a

console.log(reduce((x,y)=>x+y)(0)(aList))
> 6
```

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

const reduce = f => a => l =>
   l ? reduce(f)(f(a)(head(l)))(tail(l)) : a

console.log(reduce(x=> y=> x+y)(0)(aList))
> 6
```

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

const fold = f=> a=> l=> l ? fold(f)(f(a)(head(l)))(tail(l)) : a

console.log(fold(x=> y=> x+y)(0)(aList))
> 6
```

Reducing reduce redux

```
const aList = cons(1)(cons(2)(cons(3)(null)))

const fold = f=> a=> l=> l ? fold(f)(f(a)(head(l)))(tail(l)) : a ,
      sum = fold(x=> y=> x+y)(0)
console.log(sum(aList))
> 6
```

Looks more like Haskell but:

- No help from the compiler
- Only for cons lists… what else is foldable?
- TypeScript annotations would help us make sure that we are putting sensible things in, but can’t guarantee purity
- So many arrows and brackets!

Haskell Folds

```haskell
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
sum :: Num a => [a] -> a
sum l = foldr (\i a -> i + a) 0 l
```

Item Accumulator

```haskell
sum [5,8,3,1,7,6,2]
⇨ foldr (\i a -> i + a) 0 [5,8,3,1,7,6,2]
```

foldr

```
[5,8,3,1,7,6,2]

      (\i[i:=2] a[a:=0] -> i + a)
```

foldr

```
[5,8,3,1,7,6,2]

    (\i[i:=6] a[a:=2] -> i + a)
```

foldr

```
[5,8,3,1,7,6,2]

  (\i[i:=7] a[a:=8] -> i + a)
```

foldr

```
[5,8,3,1,7,6,2]

(\i[i:=1] a[a:=15] -> i + a)
```

foldr

```
[5,8,3,1,7,6,2]

(\i[i:=3] a[a:=16] -> i + a)
```

foldr

```
[5,8,3,1,7,6,2]

(\i[i:=8] a[a:=19] -> i + a)
```

foldr

```
  [5,8,3,1,7,6,2]

(\i[i:=5] a[a:=27] -> i + a)
```

foldr

```
[5,8,3,1,7,6,2]

sum = 32
```

Right Folds

```
sum :: Num a => [a] -> a
sum l = foldr (\i a -> i + a) 0 l

sum [5,8,3,1,7,6,2]
⇨ foldr (\i a -> i + a) 0 [5,8,3,1,7,6,2]
```

Right Folds

```
sum :: Num a => [a] -> a
sum = foldr (\i a -> i + a) 0
```

⇦ eta reduction

```
sum [5,8,3,1,7,6,2]
⇨ foldr (\i a -> i + a) 0 [5,8,3,1,7,6,2]
```

Right Folds

```haskell
sum :: Num a => [a] -> a
sum = foldr (+) 0
```

⇦ operator sectioning

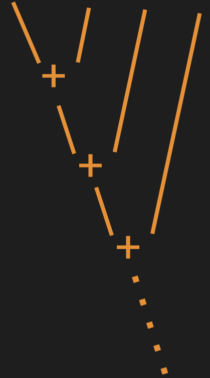
+ eta reduction

```haskell
sum [5,8,3,1,7,6,2]
```

⇨ `foldr (+) 0 [5,8,3,1,7,6,2]`

Left Folds

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
sum :: Num a => [a] -> a
sum = foldl (+) 0

sum [5,8,3,1,7,6,2]
⇨ foldl (+) 0 [5,8,3,1,7,6,2]
```



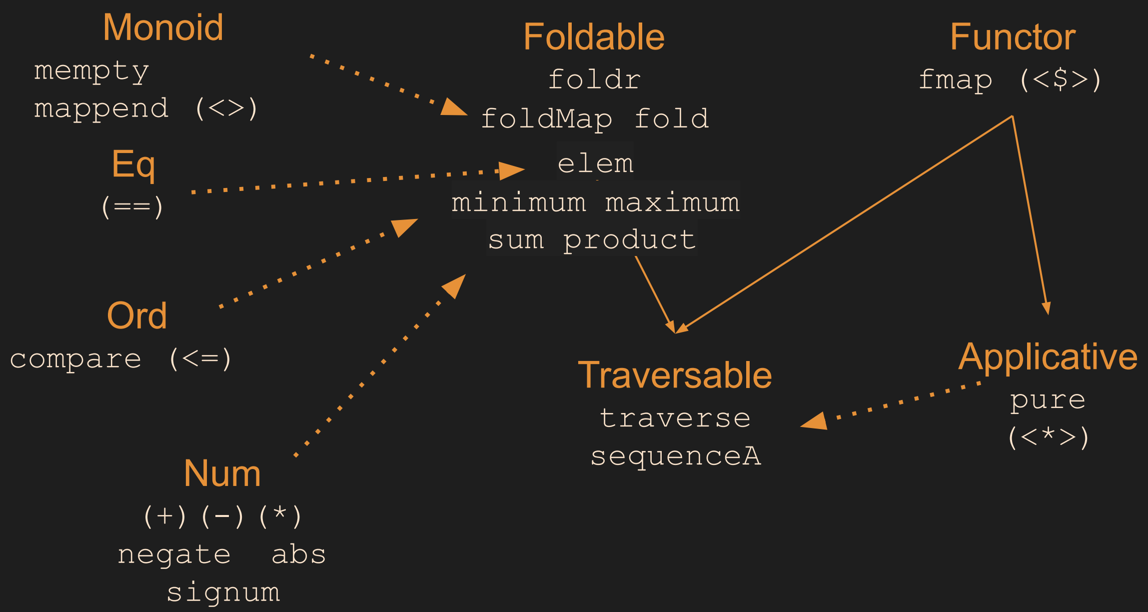
Fold left or fold right?

```
const aList = cons(1)(cons(2)(cons(3)(null)))

const fold = f=> a=> l=> l ? fold(f)(f(a)(head(l)))(tail(l)) : a

console.log(fold(x=> y=> x+y)(0)(aList))
> 6
```

Exercise

To be announced...

Typeclasses for appends, maps, folds and traversals



Monoid - things that can be “appended” together `[1,2] <> [3..9]`

Foldable - things that can be folded to accumulate a value

`foldr (+) 0 [1..9]`

Functor - things that can be mapped over `(+1) <$> [1..9]`

Applicative - Functors that support function application within their contexts

`[(+1),(*2)] <*> [1..9]`

Traversable - Functors that can be traversed by a function with an Applicative effect

`traverse print [1..9]`

Monoid

```
ghci>:i Monoid
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a    -- has alias (<>)
  mconcat :: [a] -> a
  {-# MINIMAL mempty, mappend #-}
      -- Defined in `GHC.Base'

-- defining mconcat is optional, since it has the following default:
mconcat = foldr mappend mempty
```

Sum Monoid

```haskell
newtype Sum a = Sum { getSum :: a }
instance Num n => Monoid (Sum n) where
  mempty = Sum 0
  mappend (Sum x) (Sum y) = Sum (x + y)
```

```
Prelude> import Data.Monoid
Prelude Data.Monoid> mappend (Sum 3) (Sum 4)
Sum {getSum = 7}
Prelude Data.Monoid> mconcat [Sum 1, Sum 2, Sum 3, Sum 4]
Sum {getSum = 10}
```

Product Monoid

```
newtype Product a = Product { getProduct::a }
instance Num n => Monoid (Product n) where
  mempty = Product 1
  mappend (Product x) (Product y) = Product (x * y)

prelude Data.Monoid> getProduct $ mappend (Product 3) (Product 4)
12
Prelude Data.Monoid> mconcat [Product 2, Product 3, Product 4]
Product {getProduct = 24}
```

Foldable typeclass

```
ghci> :i Foldable
class Foldable (t :: * -> *) where
  ...
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  ...
  {-# MINIMAL foldMap | foldr #-}
        -- Defined in `Data.Foldable'

instance Foldable [] -- Defined in `Data.Foldable'
instance Foldable Maybe -- Defined in `Data.Foldable'
instance Foldable (Either a) -- Defined in `Data.Foldable'
instance Foldable ((,) a) -- Defined in `Data.Foldable'
```

Foldable fold and foldMap

```
class Foldable (t :: * -> *) where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  Data.Foldable.fold :: Monoid m => t m -> m
```

```
Prelude> import Data.Monoid
Prelude Data.Monoid> import Data.Foldable
Prelude Data.Monoid Data.Foldable> fold $ Sum <$> [1,2,3,4]
Sum {getSum = 10}
Prelude Data.Monoid Data.Foldable> foldMap Sum [1,2,3,4]
Sum {getSum = 10}
```

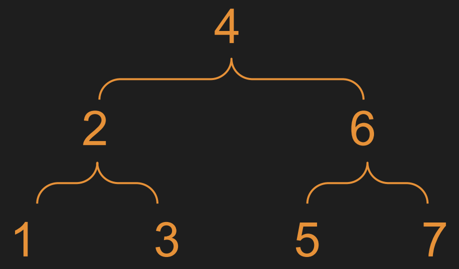
Folding other types

```
Prelude Data.Foldable> fold [[1,2],[3,4]]
[1,2,3,4]
```

List is also a Monoid:

```
  [] == mempty
  (++) == (<>) == mappend
```

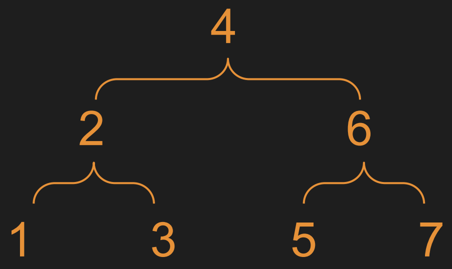
Folding custom data types




```haskell
data Tree a = Empty
            | Leaf a
            | Node (Tree a) a (Tree a)
  deriving (Show)

t = Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Node (Leaf 5) 6 (Leaf 7))

instance Foldable Tree where
   foldMap :: Monoid m => (a -> m) -> Tree a -> m
   foldMap _ Empty = mempty
   foldMap f (Leaf x) = f x
   foldMap f (Node l x r) = foldMap f l <> f x <> foldMap f r
```

Folding custom data types

```haskell
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
t = Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Node (Leaf 5) 6 (Leaf 7))
instance Foldable Tree where
   foldMap :: Monoid m => (a -> m) -> Tree a -> m
   foldMap _ Empty = mempty
   foldMap f (Leaf x) = f x
   foldMap f (Node l x r) = foldMap f l <> f x <> foldMap f r
ghci> getSum $ foldMap Sum t
28
ghci> foldMap (:[]) t
[1,2,3,4,5,6,7]
ghci> foldr (:) [] t
[1,2,3,4,5,6,7]
```



Traversable Typeclass

```
Prelude> :i Traversable
class (Functor t, Foldable t) => Traversable (t :: * -> *) where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)

Prelude> sequenceA [Just 3, Just 2, Just 4]
Just [3,2,4]
Prelude> sequenceA [Just 3, Just 2, Nothing]
Nothing
```

More fun with sequenceA

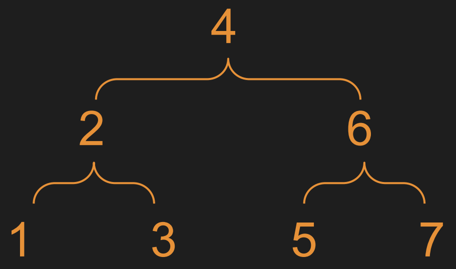
```haskell
sequenceA :: Applicative f => t (f a) -> f (t a)
```

```
Prelude> :t sequenceA [(+3),(*2),(+6)]
sequenceA [(+3),(*2),(+6)] :: Num a => a -> [a]
Prelude> sequenceA [(+3),(*2),(+6)] 2
[5,4,8]
Prelude> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]

*BinTree> treeOfMaybes = Just <$> t
*BinTree> sequenceA treeOfMaybes
Just (Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Node (Leaf 5) 6 (Leaf 7)))
```

Traverse

```
nothingIfNegative i
    | i < 0 = Nothing
    | otherwise = Just i

ghci> traverse nothingIfNegative [1,2,3,4,5]
Just [1,2,3,4,5]
ghci> traverse nothingIfNegative [1,2,3,-4,5]
Nothing
```

Instancing Traversable




```haskell
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
t = Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Node (Leaf 5) 6 (Leaf 7))
instance Foldable Tree where
    foldMap :: Monoid m => (a -> m) -> Tree a -> m
    foldMap _ Empty = mempty
    foldMap f (Leaf x) = f x
    foldMap f (Node l x r) = foldMap f l <> f x <> foldMap f r

instance Functor Tree where
    fmap :: (a -> b) -> Tree a -> Tree b
    fmap _ Empty = Empty
    fmap f (Leaf x) = Leaf $ f x
    fmap f (Node l v r) = Node (fmap f l) (f v) (fmap f r)
```

Instancing Traversable

```
instance Traversable Tree where
   traverse :: Applicative f => (a -> f b) -> Tree a -> f (Tree b)
   traverse _ Empty = pure Empty
   traverse f (Leaf a) = Leaf <$> f a
   traverse f (Node l x r) = Node <$> traverse f l <*> f x <*> traverse f r

ghci> traverse print t
1
2
3
4
5
6
7
```

Conclusions

- Abstracting common code patterns into reusable, generic functions means we don’t have to write tedious, error-prone code ourselves
- Foldable: Folds are such a useful pattern that Haskell gives them their own typeclass
- Traversable gives a way to traverse a data structure, mapping a function inside a structure while accumulating the applicative contexts along the way
- We’ll see traverse again when we look at our last typeclass for this unit: Monad