Functional Programming

Overview: Monoid, Foldable, Traversable

Ravi Chugh

UChicago CS 223
Fall 2023

```
foldr (++)        ""         ["223","00"]                :: [Char]

foldr (+)         0          [2,2,3,0,0]                 :: Int
foldr (*)         1          [2,2,3,0,0]                 :: Int

foldr (||)        False      [True, True, False]         :: Bool
foldr (&&)        True       [True, True, False]         :: Bool

foldr firstJust Nothing [Nothing, Just 2, Just 23] :: Maybe Int
   ⇒ Just 2

foldr lastJust  Nothing [Nothing, Just 2, Just 23] :: Maybe Int
   ⇒ Just 23

foldr plusJust  Nothing [Nothing, Just 2, Just 23] :: Maybe Int
   ⇒ Just 25

foldr multJust  Nothing [Nothing, Just 2, Just 23] :: Maybe Int
   ⇒ Just 46
```

```
foldr  (++)       ""       ["223","00"]                    ::  [Char]

foldr  (+)        0        [2,2,3,0,0]                      ::  Int
foldr  (*)        1        [2,2,3,0,0]                      ::  Int

foldr  (||)       False    [True, True, False]             ::  Bool
foldr  (&&)       True     [True, True, False]             ::  Bool

foldr  firstJust  Nothing  [Nothing, Just 2, Just 23]  ::  Maybe Int

foldr  lastJust   Nothing  [Nothing, Just 2, Just 23]  ::  Maybe Int

foldr  plusJust   Nothing  [Nothing, Just 2, Just 23]  ::  Maybe Int

foldr  multJust   Nothing  [Nothing, Just 2, Just 23]  ::  Maybe Int
```

Semigroup t => Monoid t

- One instance per type, so wrapper types

```
foldr (+) 0 ([2,2,3,0,0]   :: []         Int)
foldr (+) 0 (("CMSC", 223) :: (,) String Int)
foldr (+) 0 ((Right 223)   :: Either a   Int)
```

```
foldr (+) 0 ([2,2,3,0,0]   :: []         Int)
foldr (+) 0 (("CMSC", 223) :: (,) String Int)
foldr (+) 0 ((Right 223)   :: Either a   Int)
```

Foldable t

```
foldr   ::                      (a -> b -> b) -> b -> t a -> b
foldl   ::                      (b -> a -> b) -> b -> t a -> b
fold    :: Monoid m =>                              t m -> m
foldMap :: Monoid m =>     (a -> m)       -> t a -> m

elem    :: Eq a     => a -> t  a  -> Bool
( concat  ::                    t [a] -> [a]                    )

...
```

```
sequenceListOfMaybes :: [Maybe a]         -> Maybe [a]

sequenceListOfMaybes :: [] (Maybe a) -> Maybe ([] a)
sequenceListOfTrees  :: [] (Tree  a) -> Tree  ([] a)
sequenceTreeOfMaybes :: Tree (Maybe a) -> Maybe (Tree a)
```

t_outer (t_inner a)   t_inner (t_outer a)

Traversable t_outer

```
sequenceA :: Applicative t_inner =>
             t_outer (t_inner a) -> t_inner (t_outer a)

traverse  :: Applicative t_inner =>
             (a -> t_inner b) -> t_outer a -> t_inner (t_outer b)
```

- traverse is "effectful fmap"
- Traversable simultaneously generalizes Functor and Foldable