Semigroups

one typeclass with one method!

What we will cover

- what is a semigroup
- some basic semigroups
- how to combine them horizontally
- vertical combinators
- more complex semigroups
- summary

Semigroups - why do we care?

Semigroups are in my opinion one of the most underutilised constructs in haskell

They are very composable and type safe and work with a large set of types

They enable you to compose both vertically and horizontally

They are easy to use and very simple conceptually but enable very complex workflows

I'll cover a subset of what you can do with semigroups and try to avoid mentioning

What is a semigroup?

```
class Semigroup a where
  (<>) :: a -> a -> a
```

a function <> that takes two values of the same type and combines them
only requirement is that the semigroup must be associative

here are some examples of common functions that can be made into semigroups :

appending lists
"abc" ++ "def" = "abcdef"

addition
4 + 5 = 9

multiplication
4 * 5 = 20

Simple Semigroups

we have only one function (<>) for semigroups so we need to distinguish between addition and multiplication

for addition we use Sum

```
newtype Sum a = Sum a

instance Num a => Semigroup (Sum a) where
  Sum x <> Sum y = Sum (x + y)

Sum 1 <> Sum 2 <> Sum 4 <> Sum 7 = Sum 14
```

multiplication uses Product

```
newtype Product a = Product a

instance Num a => Semigroup (Product a) where
  Product x <> Product y = Product  (x * y)

Product 1 <> Product 2 <> Product 4 <> Product 7 =  Product 56
```

Associativity law

s <> (t <> u) = (s <> t) <> u

meaning it doesnt matter where you put the parentheses

appending lists
"hel" ++ ("low" ++ "orld")
("hel" ++ "low") ++ "orld"
"helloworld"

multiplication
12 * (6 * 2) = (12 * 6) * 2 = 144

division is not associative
12 / (6 / 2) = 4
(12 / 6) / 2 = 1

Min / Max

```
newtype Min a = Min a

instance Ord a => Semigroup (Min a) where
  Min x <> Min y = Min (min x y)

Min 44 <> Min 3 <> Min 7 <> Min 9 = Min 3

run (\a -> Min a) [44,3,7,9] = Min 3


Max ....
```

Horizontal Composition

```
Sum 1 <> Sum 2 <> Sum 4 <> Sum 7 = Sum 14

run (\a -> Sum a) [1,2,4,7] = Sum 14

Product 1 <> Product 2 <> Product 4 <> Product 7 =  Product 56

run (\a -> Product a) [1,2,4,7] = Product 56

(Sum 1,Product 1) <> (Sum 2, Product 2) <> (Sum 4, Product 4) <> (Sum 7, Product 7) = (Sum 14, Product 56)

run (\a -> (Sum a, Product a)) [1,2,4,7] = (Sum 14, Product 56)

run (\a -> (Sum a, Product a, Min a, Max a)) [1,2,4,7] = (Sum 14, Product 56, Min 1, Max 7)
```

Vertical Composition

```
newtype S a = S a

instance Semigroup a => Semigroup (S a) where
   S a <> S a1 = S (a <> a1)   -- normal composition

S "a" <> S "b" <> S "c"
S "abc"

S [1,2] <> S [5,6,7] <> S [9]
S [1,2,5,6,7,9]
```

Dual

```
instance Semigroup a => Semigroup (S a) where
   S a <> S a1 = S (a1 <> a)   -- Dual

Dual "a" <> Dual "b" <> Dual "c"
"cba"

Dual [1,2] <> Dual [5,6,7] <> Dual [9]
Dual [9,5,6,7,1,2]
```

First

```
instance Semigroup a => Semigroup (S a) where
   S a <> S a1 = S a                -- First

First "a" <> First "b" <> First "c"
First "a"

First [1,2] <> First [5,6,7] <> First [9]
First [1,2]
```

Last

```
instance Semigroup a => Semigroup (S a) where
   S a <> S a1 = S a1                -- Last

Last "a" <> Last "b" <> Last "c"
Last "c"

Last [1,2] <> Last [5,6,7] <> Last [9]
Last [9]
```

Dual + First + Last

Dual (First "a") <> Dual (First "b") <> Dual (First "c")
Dual (First "c")

Dual (Last "a") <> Dual (Last "b") <> Dual (Last "c")
Dual (Last "a")

Average

```
data Avg = Avg (Sum Rational) (Sum Int)
mkAvg a = Avg (Sum a) (Sum 1)

run mkAvg [10,12,13] = Avg (Sum 10) (Sum 1) <>  Avg (Sum 12) (Sum 1) <> Avg (Sum 13) (Sum 1)
    = Avg (Sum 35) (Sum 3)

unAvg :: Avg a -> Rational
unAvg (Avg (Sum a) (Sum b)) = a / fromIntegral b

runE mkAvg [10,12,13] = 35 % 3
```

Common format: mk* to lift a value into the semigroup then combine using (<>) then un* to extract the value from the semigroup.

Variance and Standard Deviation

```
data Variance = Variance Int    --  Number of elements in the sample
                         Rational -- Current sum of elements of sample
                         Rational -- Current sum of squares of deviations from current mean

mkVariance :: Rational -> Variance
mkVariance a = Variance 1 a 0

unVariance :: Variance -> Rational
unVariance (Variance a _ c) = c / fromIntegral a

instance Semigroup Variance where
  Variance n1 ta sa <> Variance n2 tb sb   = Variance (n1+n2) (ta+tb) sumsq
    where  sqr x = x * x
           na = fromIntegral n1
           nb = fromIntegral n2
           nom = sqr (ta * nb - tb * na)
           sumsq | n1 == 0   = sb
                 | n2 == 0   = sa
                 | otherwise = sa + sb + nom / ((na + nb) * na * nb)
```

langs : sample data

lang **score** **year** **name**
langType
---- ------ ----- ------

Haskell 15 1990 Simon Peyton-Jones Func
Rust 8 2010 Graydon Hoare
Imperative
Idris 19 2018 Edwin Brady
Proof
Ocaml 13 1996 Xavier Leroy
Func
Scala 12 2004 Martin Odersky
Func
Agda 19 1999 Ulf Norell
Proof
Coq 25 1984 Gerard Pierre
Huet Proof
Clojure 13 2007 Rich Hickey
Func
Purescript 15 2013 Phil Freeman

Group

```
newtype Group k v = Group (Map k v)

mkGroup :: (Ord k, Semigroup v) => k -> v -> Group k v
mkGroup k v = Group (M.singleton k v)

instance (Ord k, Semigroup v) => Semigroup (Group k v) where
  Group a <> Group b = Group (M.unionWith (<>) a b)

runE (\a -> mkGroup (score a) [(lang a,langType a)]) langs
(8,[("Rust",Imperative),("APL",Imperative)])
(11,[("F#",Func)])
(12,[("Scala",Func)])
(13,[("Ocaml",Func),("Clojure",Func)])
(15,[("Haskell",Func),("Purescript",Func)])
(19,[("Idris",Proof),("Agda",Proof)])
(25,[("Coq",Proof)])
```

Partition 1

```
newtype Partition v = Partition (Group Bool v)

newtype Group k v = Group (Map k v)

mkPartition :: Semigroup v => (a -> Bool) -> (a -> v) -> a -> Partition v

runE (mkPartition even (:[])) [1..10]
These [1,3,5,7,9] [2,4,6,8,10]

data These a b = This a | These a b | That b

runE (mkPartition (>20) (:[])) [1..10]
This [1,2,3,4,5,6,7,8,9,10]

runE (mkPartition (<20) (:[])) [1..10]
That [1,2,3,4,5,6,7,8,9,10]
```

Partition 2

```
runE (\a -> (mkPartition even ((:[])) a,(First "tot=",Sum a), (First "cnt=",Sum 1),mkAvg a)) [1..10]
(These [1,3,5,7,9] [2,4,6,8,10],("tot=",55),("cnt=",10),11 % 2)

runE (mkPartition (>4) (\a ->[a])) [1..10]
These [1,2,3,4] [5,6,7,8,9,10]

runE (mkPartition (>4) (\a -> (Sum a, [a]))) [1..10]
These (10,[1,2,3,4]) (45,[5,6,7,8,9,10])

runE (mkPartition (>4) (\a ->(Sum 1, Sum a,[a]))) [1..10]
These (4,(10,[1,2,3,4])) (6,(45,[5,6,7,8,9,10]))

runE (mkPartition (>4) (\a -> (First a, Last a, [a]))) [1..10]
These (1,(4,[1,2,3,4])) (5,(10,[5,6,7,8,9,10]))
```

Ascending / Descending sequences

```
>runE mkAscStrict [1,4,5,7,9]
Right ()

>runE mkAscStrict [1,4,5,7,7,9]
Left ([7],[1,4,5,7,9])

>runE mkAscMono [1,4,5,7,7,9]
Right ()

>allOrdering [1,2,3,4,5]
["ascStrict","ascMono"]

>allOrdering [5,5,5]
["ascMono","allEq","descMono"]

>allOrdering [5,4,3,3,2,1]
["descMono"]

>allOrdering [5,4,3,3,2,1,5]
[]
```

Span

```
runE (\a -> mkSpanMax ([a]) a) [1,4,9,5,7,9]
(Just [1,4,5,7],[9,9])

runE (\a -> mkSpanMax ([a],Sum 1) a) [1,4,9,5,7,9]
(Just ([1,4,5,7],4),([9,9],2))

runE (\a -> mkSpanMax ([a],mkAvg a) a) [1,4,9,5,7,9]
(Just ([1,4,5,7],17 % 4),([9,9],9 % 1))

(Just a,b) = runE (\a -> mkSpanMin [a] (Arg (score a) a)) langs
wprint b

(Just a,b) = runE (\a -> mkSpanMin (mkSortAscs [desc score, desc year] a) (score a))
wprint b
```

MultiSet

a multiset is like a set but allows multiple elements

```
unMultiSet' "xybacazxz"
[('a',2),('b',1),('c',1),('x',2),('y',1),('z',2)]

newtype MultiSet k = MultiSet (Map k (Sum Int))

instance Ord k => Semigroup (MultiSet k) where
  MultiSet m <> MultiSet n = MultiSet (M.unionWith (+) m n)

unMultiSet' ("abc" <> "def")
[('a',1),('b',1),('c',1),('d',1),('e',1),('f',1)]

unMultiSet' ("axbc" <> "deaxxf")
[('a',2),('b',1),('c',1),('d',1),('e',1),('f',1),('x',3)]
```

Intersection

intersection of lists: uses MultiSets to track intersections

runE (mkISectListI min) ["yaab","aaaaabbx","dbefaa"]
("aaabdefxy","aab")

left hand side has the values that do not intersect
right hand side has the values that intersect

IO programs are semigroups

```
instance Semigroup a => Semigroup (IO a) where
  ioa <> iob = do
                         a <- ioa
                        b< - iob
                        return (a <> b)

pgm1 :: IO String
pgm1 = do
   putStrLn "in pgm1"
   return "hello"

pgm2 :: IO String
pgm2 = do
   putStrLn "in pgm2"
   return " world"

pgm1 <> pgm2
in pgm1
in pgm2
```

Parallel + Race using async library 1

```
newtype Parallel a = Parallel (A.Concurrently a) deriving Semigroup

mkParallel  :: Semigroup a => IO a -> Parallel a
mkParallel = Parallel . A.Concurrently

newtype Race a = Race (A.Concurrently a)

instance Semigroup (Race a) where
  Race a <> Race b = Race (a <|> b)

mkRace :: Semigroup a => IO a -> Race a
mkRace = Race . A.Concurrently
```

Parallel + Race using async library 2

```
pgm4d :: Int -> IO [(String, Int)]
pgm4d n = do
  th <- myThreadId
  let msg = "thread " ++ show th ++ " val=" ++ show n ++ " deciseconds"
  handle (\e-> putStrLn ("error: " ++ msg ++ " " ++ show e) >> throwM e) $ do
    putStrLn ("info: " ++ msg ++ " starting")
    threadDelay (100000*n)
    putStrLn ("info: " ++ msg ++ " ended")
    return [(msg,n)]

runE (mkParallel . pgm4d) [10,40,1,5]

runE (mkRace . pgm4d) [10,40,1,5]

BT.bisequenceA $ runE ((\x -> (mkRace x, mkParallel x)) . pgm4d) [10,40,1,5]

BT.bisequenceA $ runE ((This . mkRace <> That . mkParallel) . pgm4d) [10,40,1,5]
```

Parallel + Race expansion

```
BT.bisequenceA $ runE ((This . mkRace <> That . mkParallel) . pgm4d) [10,40,1,5]

BT.bisequenceA $ runE (\a -> This (mkRace (pgm4d a)) <> That (mkParallel (pgm4d a))) [10,40,1,5]

-- this expands out to this: 2 levels of semigroup nesting

BT.bisequenceA $ unw $
     (This (mkRace (pgm4d 10)) <> That (mkParallel (pgm4d 10)))
<> (This (mkRace (pgm4d 40)) <> That (mkParallel (pgm4d 40)))
<> (This (mkRace (pgm4d 1))   <> That (mkParallel (pgm4d 1)))
<> (This (mkRace (pgm4d 5))   <> That (mkParallel (pgm4d 5)))

prThese it
this ("thread ThreadId 824 val=1 deciseconds",1)

that ("thread ThreadId 826 val=10 deciseconds",10)
that ("thread ThreadId 828 val=40 deciseconds",40)
that ("thread ThreadId 830 val=1 deciseconds",1)
that ("thread ThreadId 831 val=5 deciseconds",5)
```

Summary

- semigroups traverse the list only once!
- semigroups provide vertical and horizontal composability
- type safety guarantee that you can't mess up
- ease of use but enable very complex workflows
- there are a lot of things I didn't cover that add more dimensions to semigroups
  - eg Ordering / Arg / Down / Sorting / Reader
- questions?

eof

Sort

```
class Ord a where
  compare :: a -> a -> Ordering

data Ordering = LT | EQ | GT
instance Semigroup Ordering where
  EQ <> r = r
  LT <> _ = LT
  GT <> _ = GT

comparing :: Ord a => (b -> a) -> b -> b -> Ordering

-- sort by score
wprint $ runE (mkSort (comparing score)) langs

-- sort by score descending
wprint $ runE (mkSort (comparing (Down . score))) langs
```

Down

```
newtype Down a = Down a

instance Ord a => Ord (Down a) where
  compare (Down a) (Down a1) = compare a1 a


-- sort by score descending and within same scores descending by year
wprint $ runE (mkSort (comparing (Down . score) <> comparing (Down . year))) langs
```

crazy using Option + Arg + Dual

```
   Dual (Option Nothing)
<> Dual (Option (Just (Min (Down (Arg 10 "the first ten")))))
<> Dual (Option (Just (Min (Down (Arg 10 "the other ten")))))
<> Dual (Option (Just (Min (Down (Arg 4 "the first four")))))
<> Dual (Option (Just (Min (Down (Arg 4 "the other four")))))
<> Dual (Option (Just (Min (Down (Arg 7 "seven")))))
<> Dual (Option Nothing)
```