Chapter 15 : Monoid, Semigroup.

*) recognising abstract patterns in code which have well-defined, lawful representations in mathematics.

*) word used to describe these abstractions is called algebra, by which we mean two or more operations & the set they operate over.

*) algebra in current context : means the study of mathematical symbols and the rules governing their manipulation.

↓

differentiated from arithmetic by its use of abstractions such as variables.

↓

we don't care much about values, rather about rules of how to manipulate this thing w/o reference to its particular value.

*) In Haskell, these algebras can be implemented using typeclasses. wherein the typeclasses define the set of operations. and when we talk about set, set can be thought of as the type the operations are for. The instance defines how each op will be performed for a given type or set.

→ Monoid : binary associative operation with an identity

operation → function

identity → value for which when combined with some other value, will give the other value

[] for list concatenation ← 0 for (+) ←

using (++)

↓

we can also use a fn. called mappend, from the Monoid typeclass.

↓

```
mappend [1, 2, 3, 4] [5, 6, 7]
```

↓

mempty (the identity value in the Monoid typeclass)

↓

```
mappend x mempty = x
mappend mempty x = x
```

*) In plain English, monoid is a fn. that takes two arguments and follows the two laws of associativity & identity.

↓

means that the args can be reordered to get the same result

Monoid is the typeclass that generalises these laws across types.

→ How Monoid is defined in Haskell: typeclasses give us a way to recognize, organize & use common functionalities & patterns across types that differ in some ways but also have things in common.

↓

Pattern of Monoid: types that have binary fns, which let us join things together in accordance with laws of associativity, and an identity value.

↓

Pattern behind summation, multiplication & list concatenation.

*) Defn. of Monoid :

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

→ the method through the two arg. op takes place

→ point fn.

↓

performing the redn over a list, with identity value used as base value

→ Examples of using Monoid

*) list : common type with Monoid instance

```
>> mappend [1,2,3] [4,5,6]
   [1,2,3,4,5,6]
>> mconcat [[1..3], [4..6]]
   [1,2,3,4,5,6]
>> foldr (++) [] [[1..3],[4..6]]
   [1,2,3,4,5,6]
>> foldr mappend mempty [[1..3], [4..6]]
```

*) instance Monoid [a] where

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

→ Why Integer doesn't have a Monoid : None of the numeric types do. This is because of the ambiguity as to which operation b/w summation & multiplication should be a monoidal operation over numbers, and there should exist only one unique

instance for a given typeclass - type pair.

↓

even though in mathematics, summation is defined as the monoid operation over numbers.

↓

To resolve the conflict between addn & multiplication in this case, we use Sum & Product newtypes to wrap numeric values, and signal which Monoid instance we want to use.

↓

The Sum & Product newtypes are built into the Data.Monoid module.

↓

```
>> mappend (Sum 1) (Sum 5)
Sum {getSum = 6}
>> mappend (Product 2.5) (Product 3.2)
Product
```

Technique: wrapping a datatype into different newtype to be able to utilize multiple instances of monoid & resolving ambiguity.

↓

enforce the unique instance by using newtype

↓

The way numeric values form monoid over addn & mult. lists form a monoid over a concatenation.

↓

Note: Even list has monoid over other operations.

→ why newtype?

i) guarantees no additional runtime overheads. in wrapping the original

combination of a structure and an operation on two of its instance, agreeing to associativity & identity properties

type.

ii) makes it clear that the created type is meant to be used as a wrapper for the underlying type. The newtype cannot eventually grow into a more complicated sum or product type, while a normal datatype can.

iii) to improve type safety: avoiding to mix many values of the same representation.

iv) to add a different typeclass instance behaviour other than the one marked with the underlying type

*) More sum & Product:

```
>> :info Sum
newtype Sum a = Sum {getSum :: a}
  :
instance Num a => Monoid (Sum a)
```

highlights the fact that these values can be used as monoid as long as these contain numeric values.

*) infix operator for mappend : (<>)

does the same thing & has the same type

*) (Sum 8) <> (Sum 9)

Sum {getsum = 17}

*) mappend need parenthesis to apply multiple times, but that is not the case with '<>'

```
>> Sum 1 <> Sum 2 <> Sum 3
```

Sum { getSum = 6 }

*) mconcat [sum8, sum9, sum10] → aggregates the result by applying with a starting element, and then taking one at a time, the <> operation.

↓

i.e. repeated application of mappend.

→ Why bother? common, nice abstraction to work with

↓

further, having principled laws for it means, that we know how to combine monoidal operations safely. i.e. atleast one law-abiding Monoid instance for the type can be defined.

↓

Commonly used to structure & describe common modes of processing data. Sometimes to describe an API for incrementally processing a large dataset.

↓

Identity for a generic library for doing work in parallel. we can choose to describe our work in the form of a tree, with each unit of work being a leaf. From there, we can partition the tree into as many chunks as necessary to saturate the no. of processor cores or entire comps, we want to devote to the work.

↓

A problem that can arise here is that if we have a pair-wise operation, and we need to combine odd no. of leaves, how do we even out the count

(using monoid, we can apply the operation to both odd & even no. of arguments, in a recursive manner)

↓

One straightforward method could be to simply provide mempty to the odd leaves, more like byes in a tournament, to even out for the next layer of aggregation.

↓

A variant of monoid that provides more guarantees is Abelion or commutative monoid. Commutativity can be particularly helpful when doing concurrent or distributed processing of data because it means the intermediate results being computed in a different order won't change the eventual answer

↓

Monoid are associated with folding strongly.

↓

fold or mappend mempty ([2, 4, 6] :: [Product Int])

→ Laws: laws provide us guarantees that gives us predictable composition of programs. Laws lead to development of trustworthy programs.

Monoid must abide the following laws : algebras as typeclasses

a) mappend mempty x = x

   mappend x mempty = x

b) mappend x (mappend y z) = mappend (mappend x y) z

c) mconcat = foldr mappend mempty

*) The important part is that these guarantees stay even when we have no idea about what monoid we are working with.

→ Different instance, same representation : possible to have more than one valid monoid, which is a behaviour different from other typeclasses.

↓

More than one valid monoids calls for the convenient use of newtypes. to build a wrapper arce the type.

↓

In many cases, monoidal operation is about finding a summary value for the set. Appending is perhaps best thought of not as a way of combining values in the way addn or list concatenation does, but as a way to condense any set of values to summary value.

↓

for example, Boolean : (ff) and (||) → newtypes to distinguish

&& : All

|| : Any

↓

>> All True <> All True

All { getAll = True }

>> All True <> All True <> All False

All { getAll = False }

>> Any False <> Any False <> Any True

Any { getAny = True }

*) Maybe has more than two possible Monoids. Two of them:

i) First : returns the first (left) non Nothing value

ii) Last : returns the last (right) non Nothing value

↓

Like Boolean disjunction, but with explicit preference for the leftmost or rightmost success in a series of Maybe values.

With Boolean, all we know is True & False, and it doesn't really matter where the values occurred. With Maybe however, we need to make a decision as to which Just value we'll return if there are multiple successes.

↓

>> First (Just 1) `mappend` First (Just 2)

First { getFirst = 1 }

>> Last (Just 1) `mappend` Last (Just 2)

Last { getLast = 2 }

```
>> Last (Nothing) <> Last (Nothing)
Last {getLast = Nothing}
```

→ Reusing algebras by asking for algebras

*) The other Monoid instance for Maybe:

Combining rather than choosing a values contained with the Maybe a type.

typeclass behaviour being inherited

*) instance Monoid b ⇒ Monoid (a → b) — inheriting the behaviour

instance (Monoid a, Monoid b) ⇒ Monoid (a, b)

instance (Monoid a, Monoid b, Monoid c) ⇒ Monoid (a, b, c)

↓

These Monoids are giving us a new Monoid for a larger type by reusing the Monoid instances of types that represent components of the larger types.

↓

i.e. for the first one, the o/p of the function will have monoidal behaviour, hence a fn. appn., wherein a monoidal result is expected, can be mappended. (a → b)

↓

for the second it means that if the constituent types have monoidal behaviour, so will the bigger type, and the behaviour will be picked from individual monoidal behaviours.

↓

```
let x = (Sum 2, Sum 3)
let y = (Sum 4, Sum 2)
```

mappend x y

↓

This obligation to ask for a Monoid for an encapsulated type (such as a in Maybe a) exists even when not all possible values of the larger type contain the value of the type argument.

↓

for example Nothing does not contain the a we're trying to get a Monoid for.

↓

for a monoid that will have a mappend operation for the a values, we need to Monoid for whatever type a is. (condition to define an instance)

↓

Monoids like First & Last wrap the Maybe a but donot require a Monoid for the a value itself because they don't mappend the a values or provide a mempty of them.

↓

A phantom type does not need to inherit a Monoidal type to define an instance

↓

for example data Booly a = False'
| True'
deriving (Eq, Show)

instance Monoid (Booly a) where
mappend False' _ = False'

```
mappend _ False' = False'
mappend True' True' = True'
```

→ Exercise : optional Monoid

```
data Optional a = Nada
                | Only a
                deriving ()
```

since classes have evolved, so to inherit a Monoid, we need to define a Semigroup class instance to define monoidal behaviour of the empty element in monoid instance. Plus '<>' use is prevalent.

```
instance Monoid a => Monoid (Optional a) where
  mempty = Nada

instance Semigroup a => Semigroup (Optional a) where
  (<>) (Only x) (Only y) = Only (x <> y)
  (<>) (Only x) _ = Only x
  (<>) _ (Only x) = Only x
  (<>) _ _ = Nada
```

→ Associativity : says that we can associate, or group, the arguments of one operation differently & the result will be the same.

↓

Not as strong as commutative property.

↓

Addition, multiplication : both commutative & associative

(++) : only associative

↓

Commutativity might be useful when there is a need to reorder evaluation of data for efficiency purposes without needing to worry about the result changing.

→ Identity : turns an operation into an identity fn. no identities without operations. Having a sensible identity value (mempty). is another requirement of a monoid.

→ The problem of orphan instances :

↓

when an instance is defined for a datatype and a typeclass, but not in the same module as either the declaration of the typeclass or data type.

↓

If we don't own the typeclass or datatype, newtype it. i.e. orphan instance behaviour should be avoided at all costs.

↓

for example, make a project directory and change into that directory. Then we make 2 files, one module in each : Listy.hs & ListyInstances.hs

↓

```haskell
module Listy where

newtype Listy a =
  Listy [a]
  deriving (Eq, Show)
```

```haskell
module ListyInstances where

import Data.Monoid
import Listy

instance Monoid (Listy a) where
  mempty = Listy []
  mappend (Listy l) (Listy l') =
    Listy $ mappend l l'
```

Listy.hs
ListyInstances.hs
} directory tree

*) Now to build ListyInstances such that it can see Listy, we must use the -I flag to include the current directory & make modules within it discoverable.
'.' indicates the current directory

↓

ghc -I. -- make ListyInstances.hs.

↓

only o/p is an object file, the result of compiling a module file that can be reused as library by Haskell code., because we didn't define a main suitable for producing an executable.

↓

Only doing this to avoid the hassle of initializing (via Stack new or similar) a project. for anything more complicated or long-lived than this, we use a dependency and build mgmt tool like Cabal (available with Stack).

↓

Now, if we were to copy the Monoid instance from ListyInstances into Listy and build again., we'll get an error about duplicate instances.

↓

Orphan instances are still a problem even if duplicate instances aren't both imported into a module, because it means that our typeclass will start behaving differently depending on what modules are imported, which breaks the basic, fundamental assumptions and niceties of typeclasses.

*) Solns. to address type classes:

i) Type defined but not the typeclass: Put the instance in the same module as the type defn., so that the type defn cannot be imported w/o its instances.

ii) typeclass defined, but not the type: instance in the same module as the typeclass module, so that the typeclass cannot be imported w/o the instances.

iii) neither typeclass nor the type: Define a newtype for the type & follow 1.

*) A type must have a unique (singular) implementation of a typeclass in scope, and avoiding orphan instances.

is how we prevent conflicting instances.

→ Semigroup : Monoid - identity. core operation remains binary and associative. ∴ defn. of semigroup is :

```
class semigroup a where
    (<>) :: a → a → a
```

and one law : $a <> (b <> c) = (a <> b) <> c$

*) Nonempty : datatype which can't have a Monoid instance but does have a Semigroup instance is the NonEmpty list type. As the name suggests, it is a list that cannot be empty.

↓

```
data NonEmpty a = a :| [a]
```

→ infix data constructor that takes two (type) args

↓

product type

derived

↓

```
>> :t 1 :| [2,3]
```

to operate NonEmpty → need to import Data.List.NonEmpty

↓

use N as type alias, to access fns.

```
1 :| [2,3] :: Num a => NonEmpty a
>> let xs = 1 :| [2,3]
>> let ys = 4 :| [5,6]
>> xs <> ys
1 :| [2,3,4,5,6]
>> N.head xs
1
>> N.length xs
3
```

→ Strength can be weakness : strength of an algebra refers to the no. of operations, an algebra provides which in turn expands what we can do with any given instance of that algebra w/o needing to know specifically what type we are working with.

↓

The reason we cannot & donot want to make all our algebras as big as possible is that there are datatypes which are useful representationally, but donot have the ability to satisfy everything in a larger algebra that could work fine if we removed an operation or law.

↓

This introduces the need for Semigroup. for example, needing to remove identity req. from nonempty to define an operation & have the associative safety.

↓

Monoid is stronger than a semigroup, since it has a strict superset of the operations & laws that Semigroup provides.

↓

In the manner as explained for the inverse relationship between operations permitted over a type & the no. of types that can be satisfied i.e. no. of types permitted ↓, no. of possible operations ↑

can also be seen for no. of operations and laws an algebra demands and the no. of datatypes that can provide a law abiding instance of that algebra

↓

When monoid is too strong or more than we need, we can use semigroup. 'magma' is something weaker than semigroup but isn't necessarily required in day-to-day Haskell.

→ Better living through Quickcheck: Quickcheck happens to be an excellent way to get a sense of whether or not the laws are likely to be obeyed by an instance.

*) validating associativity

```
\f a b c → f a (f b c) == f (f a b) c
```

abstract property of associativity for a given fn. f

↓

```
\(<>) a b c → a <> (b <> c) == (a <> b) <> c
```

↓

```
asc :: Eq a
    ⇒ (a → a → a)   → fn. accepting two parameteres
    → a → a → a     → three args to test associativity among
    → Bool          → result of test
asc (<>) a b c =
    a <> (b <> c) == (a <> b) <> c
```

↓

quickest way to test the fn. for a property test

↓

```
import Data.Monoid
import Test.QuickCheck

monoidAssoc :: (Eq m, Monoid m)
            => m -> m -> m -> Bool
monoidAssoc a b c =
  (a <> (b <> c)) == ((a <> b) <> c)
```

↓

Need to declare the types for the fn. in order to run the tests, so that Quickcheck knows what types of data to generate.

↓

```
>> type S = String
>> type B = Bool
>> quickCheck (monoidAssoc :: S -> S -> S -> B)
```

↓

Quickcheck uses the Arbitrary typeclass to provide the randomly generated i/ps for testing the fn.

↓

we may not want to rely on an Arbitrary instance existing for the type of our i/ps. It may be that we need a generator for a type that doesn't belong to us; so we'd rather not make an orphan instance.

↓

or it could be that the type already has an Arbitrary instance, but we want to run the tests with a different random distribution of values, or to make sure that certain special edge cases are being checked in addn. to random values.

↓

Verbose Check can be used to check which values were passed for testing.

*) Testing left & right identity

```
monoidLeftIdentity :: (Eq m, Monoid m)
                   => m
                   -> Bool
monoidLeftIdentity a = mempty <> a = a
```

↓

```
quickCheck (monoidLeftIdentity :: String -> Bool)
```

similarly can be done about right identity

*) Testing Quickcheck's patience : checking on an invalid monoid

↓

why a Bool monoid can't have false as the identity, always returning the value false, and still be a Monoid.

↓

```
import Control.Monad
import Data.Monoid
import Test.QuickCheck

data Bull = Fools
          | Twoo
          deriving (Eq, Show)

instance Arbitrary Bull where
  arbitrary =
    frequency [(1, return Fools)
              , (1, return Twoo)]

instance Monoid Bull where
  mempty = Fools
  mappend _ _ = Fools

type BullAppend =
  Bull -> Bull -> Bull -> Bool
```

→ Associativity type check.

```
main :: IO ()
main = do
  let ma = monoidAssoc
      mli = monoidLeftIdentity
      mri = monoidRightIdentity
  quickCheck (ma :: BullAppend)
  quickCheck (mli :: Bull -> Bool)
  quickCheck (mri :: Bull -> Bool)
```

→ this passes

→ fails

→ fails

} since on applying identity, we are not getting x, rather

getting fooled.

more like why mempty for product is not 0. It spits out 0 for all ops. in multiplication.

*) Using quickcheck can be a great way to cheapily & easily sanity check the validity of our instance against the laws

---

→ Chapter exercises.

*) Semigroup exercises