

A large, stylized 'X' logo composed of several overlapping, semi-transparent rectangular blocks in shades of purple and blue, creating a 3D effect. It is centered in the background of the slide.

Lists and Other Monoids

Tom Schrijvers

Leuven Haskell
User Group



Data
Genericity



Recursion
Schemes

GADTs

DSLs



Expression
Problem

Monads

Type
Families

Type
Classes

Lists
and
Other
Monoids

Effect
Handlers

Free
Theorems

...



Data
Genericity



Recursion
Schemes

GADTs

DSLs



Expression
Problem

Monads

Type
Families

Type
Classes

Lists
and
Other
Monoids

Effect
Handlers

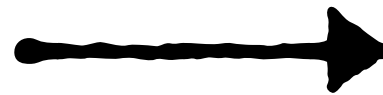
Free
Theorems

...

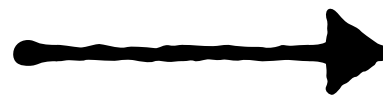
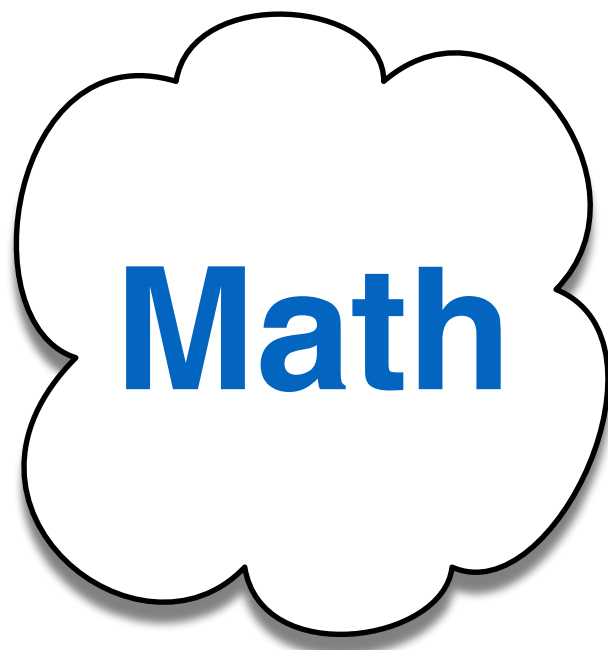
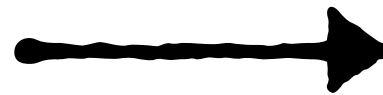
A large, stylized 'X' logo composed of several overlapping, semi-transparent, dark blue-grey rectangular blocks. The blocks are arranged to form the two intersecting lines of the 'X', with some blocks extending further to create a layered, three-dimensional effect.

Introduction to Monoids

Abstract Patterns



Abstract Patterns





Haskell's Math Inspiration



Haskell's Math Inspiration



Algebra

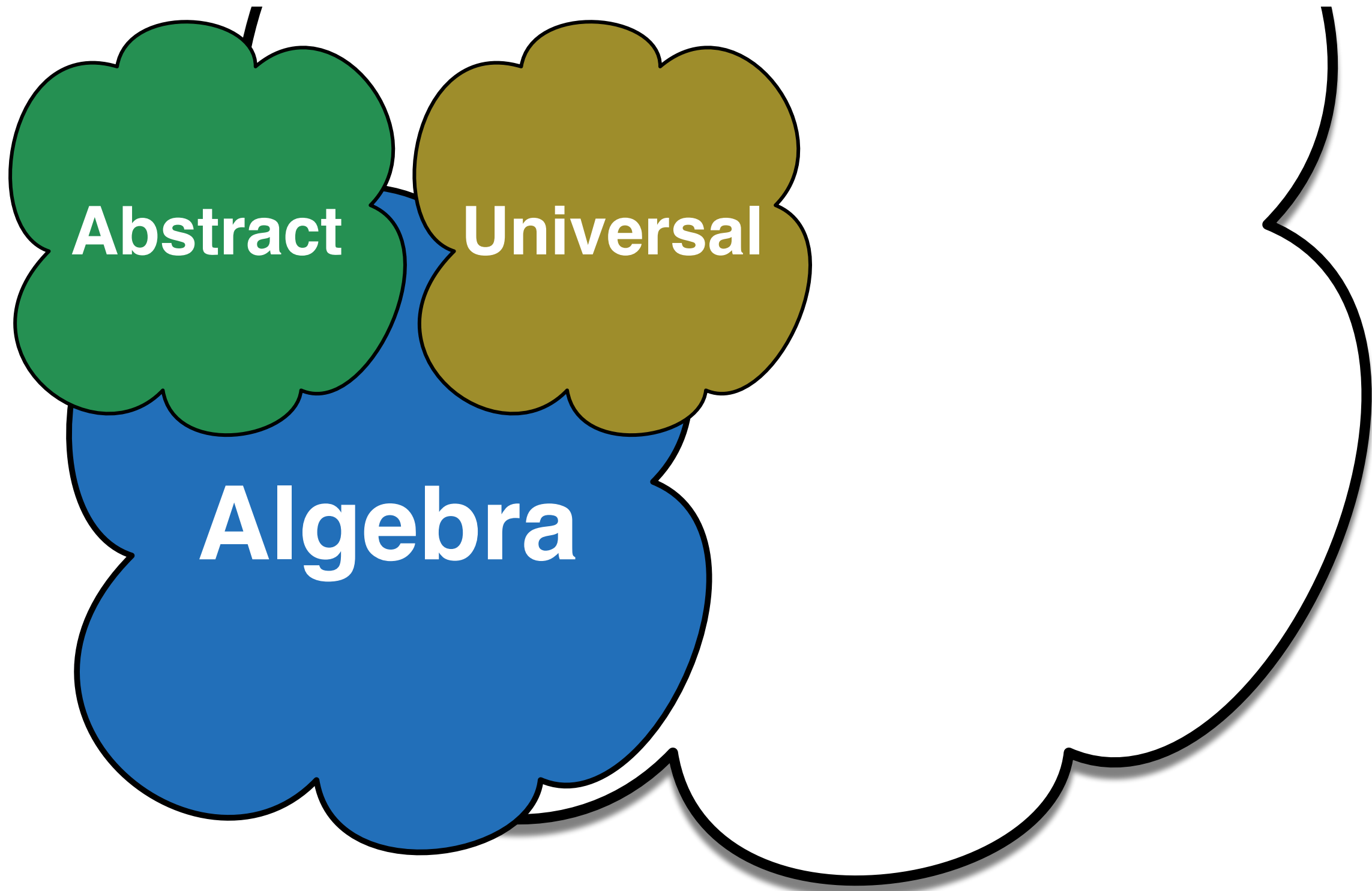
Haskell's Math Inspiration



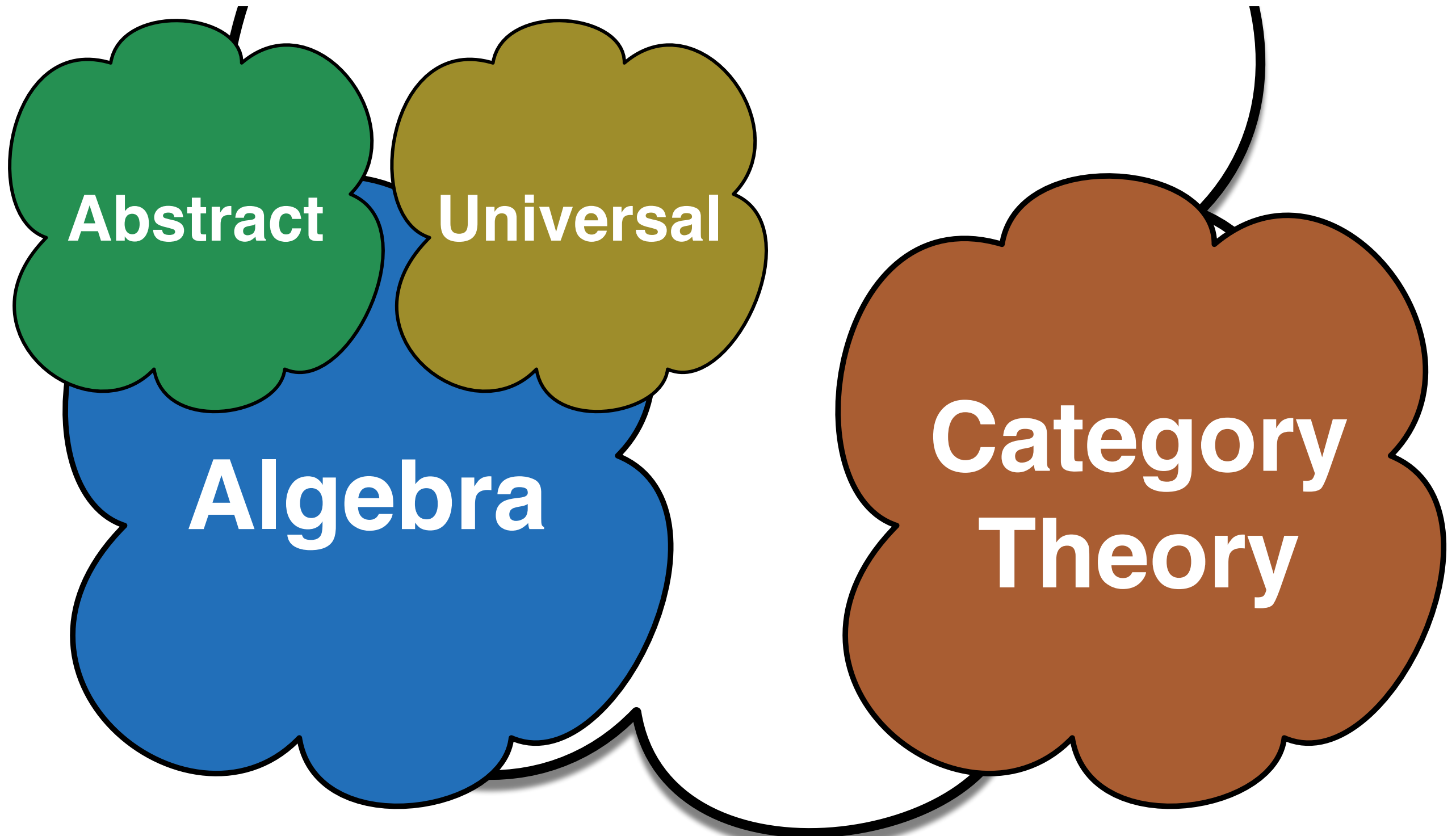
Abstract

Algebra

Haskell's Math Inspiration



Haskell's Math Inspiration



Algebra

**Category
Theory**

The diagram consists of three nodes connected by a continuous black line. On the left is a blue cloud-shaped node containing the word 'Algebra'. On the right is a brown cloud-shaped node containing the words 'Category Theory'. At the bottom left is a blue rounded rectangular node containing the text 'regular Haskell monoids'. A line connects the bottom of the 'Algebra' cloud to the 'regular Haskell monoids' rectangle. Another line connects the top of the 'Algebra' cloud to the top of the 'Category Theory' cloud. A third line connects the bottom of the 'Category Theory' cloud to the bottom of the 'regular Haskell monoids' rectangle, completing a loop.

Algebra

**Category
Theory**

**regular
Haskell monoids**

Algebra

**regular
Haskell monoids**

**Category
Theory**

**monoids in the
category of
endofunctors**

Algebra

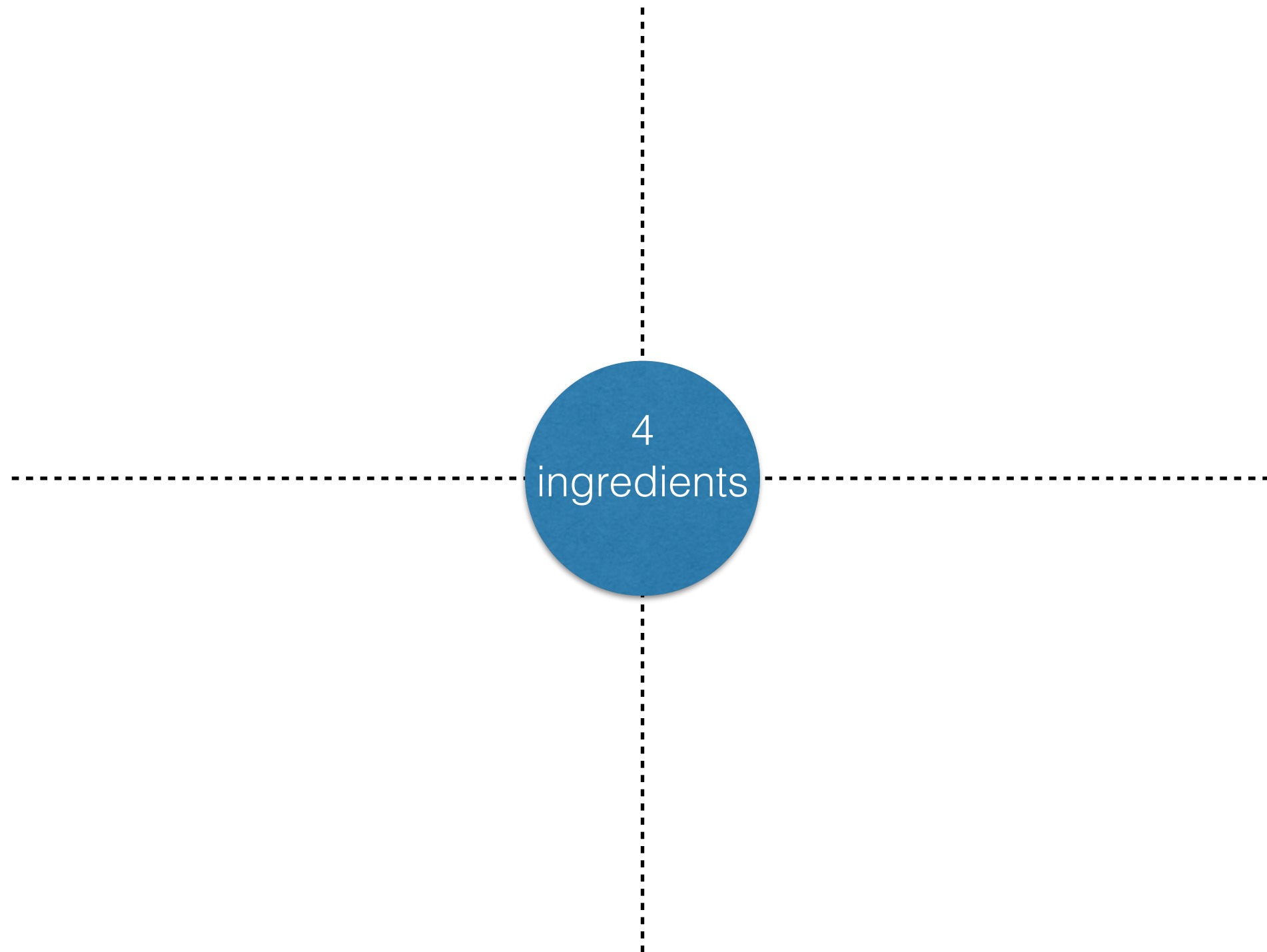
The diagram consists of four nodes arranged in a square. The top-left node is a blue cloud shape labeled 'Algebra'. The top-right node is a brown cloud shape labeled 'Category Theory'. The bottom-left node is a blue rounded rectangle labeled 'regular Haskell monoids'. The bottom-right node is a red rounded rectangle labeled 'Haskell monads and applicative functors'. A black line connects the bottom of the 'Algebra' cloud to the top of the 'regular Haskell monoids' box. Another black line connects the bottom of the 'Category Theory' cloud to the top of the 'Haskell monads and applicative functors' box. A curved black line connects the top of the 'regular Haskell monoids' box to the top of the 'Haskell monads and applicative functors' box, passing behind the space between the two top clouds.

**Category
Theory**

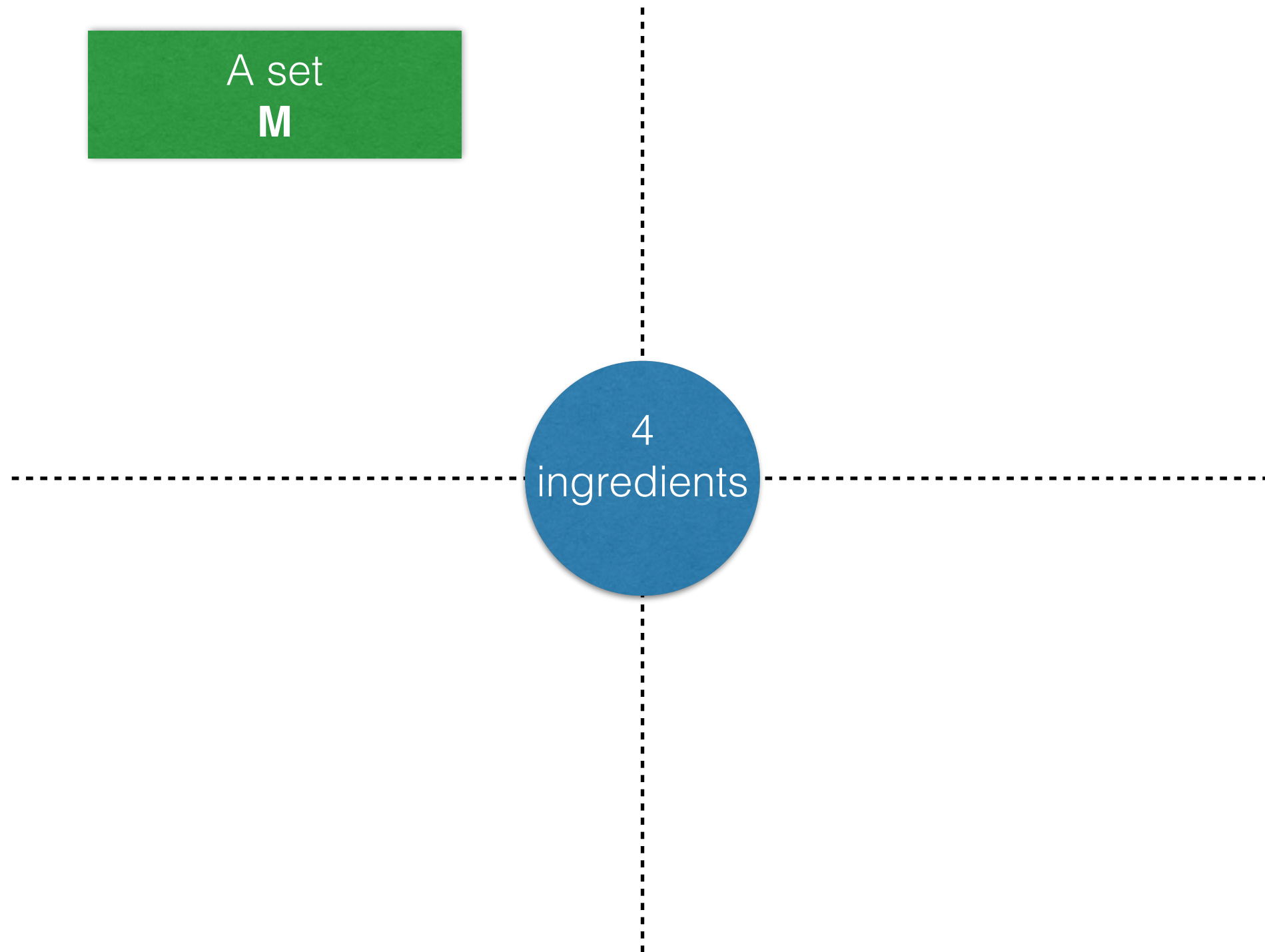
**regular
Haskell monoids**

**Haskell
monads and
applicative functors**

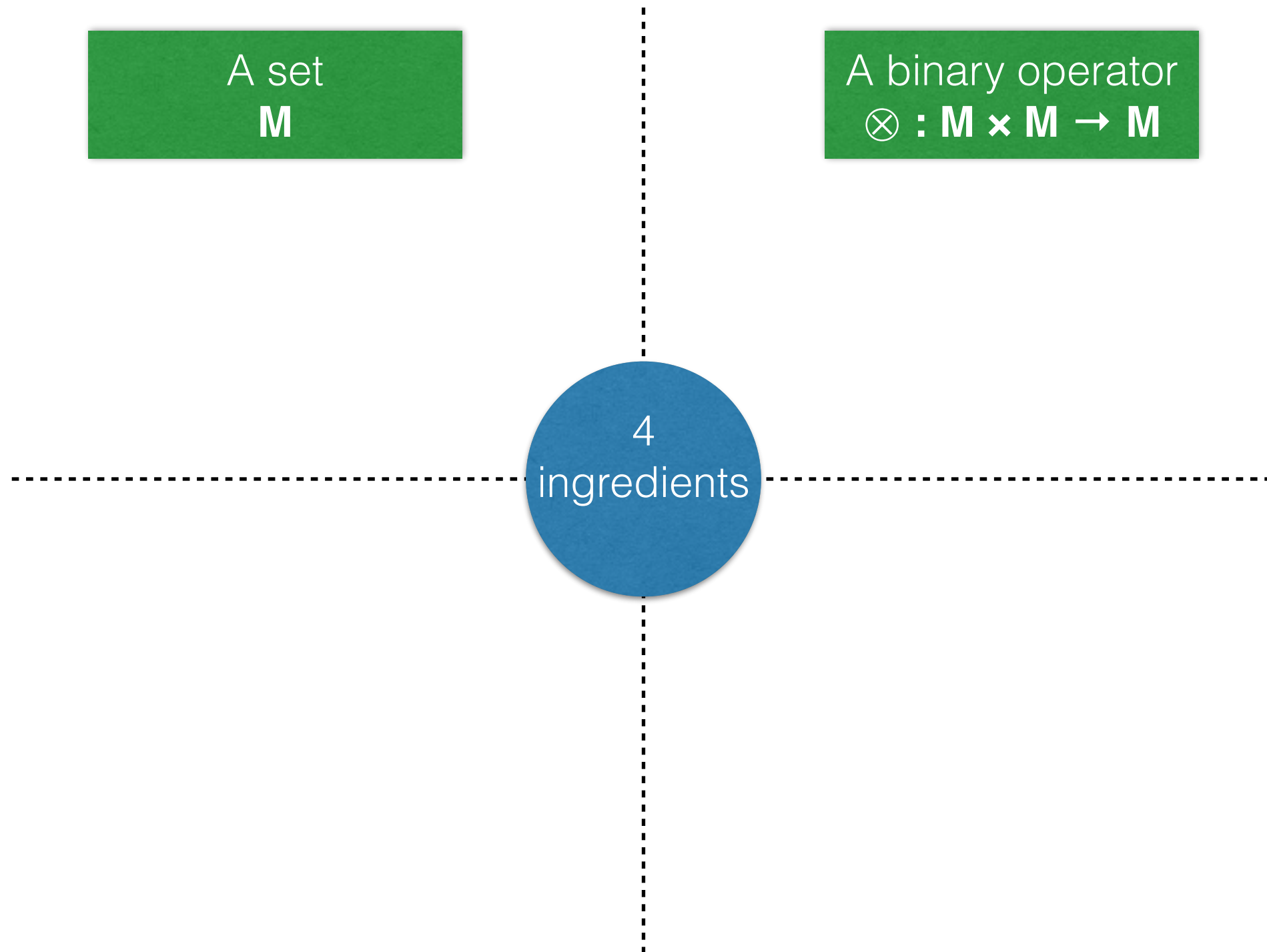
The Monoid Structure



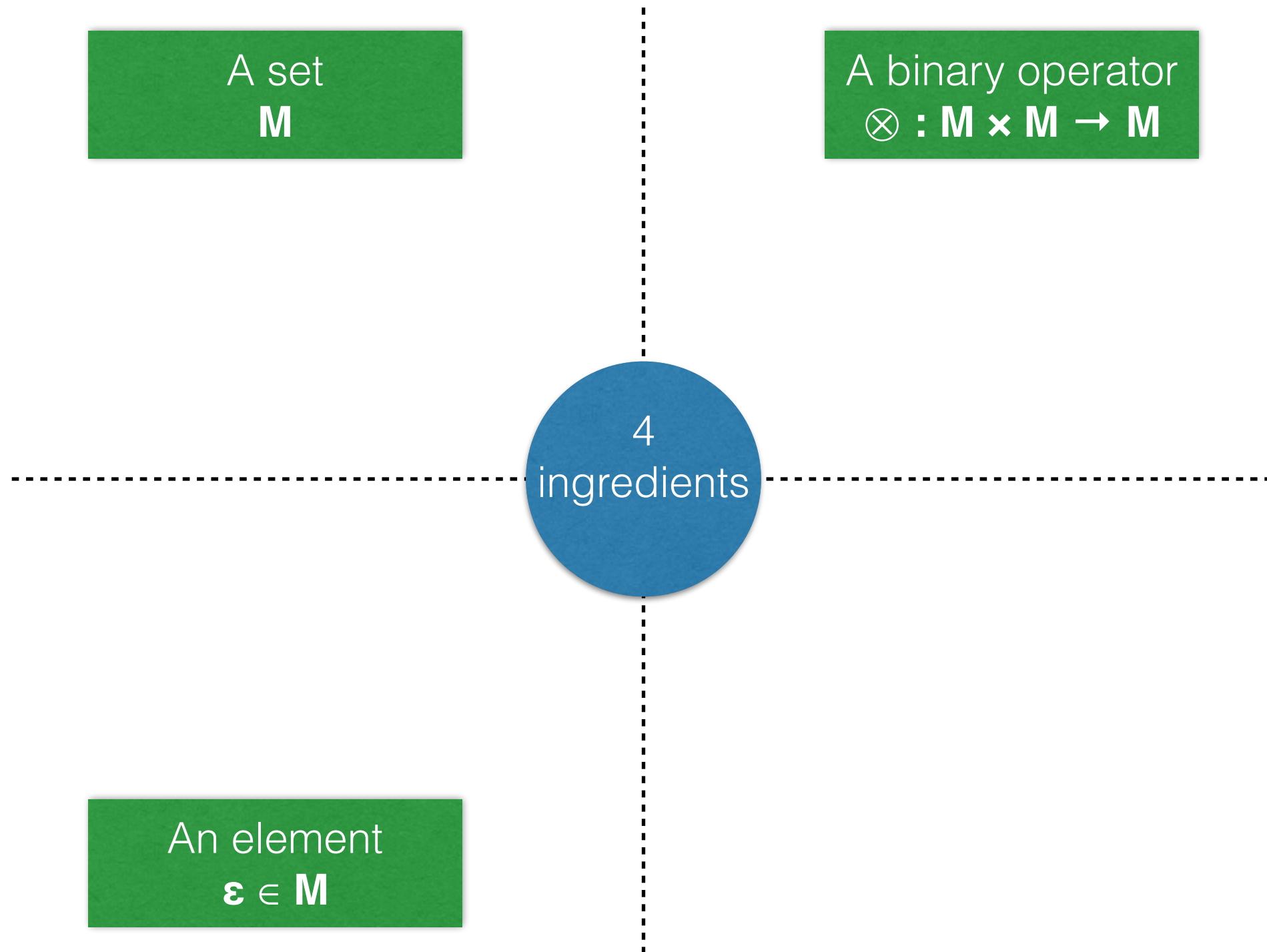
The Monoid Structure



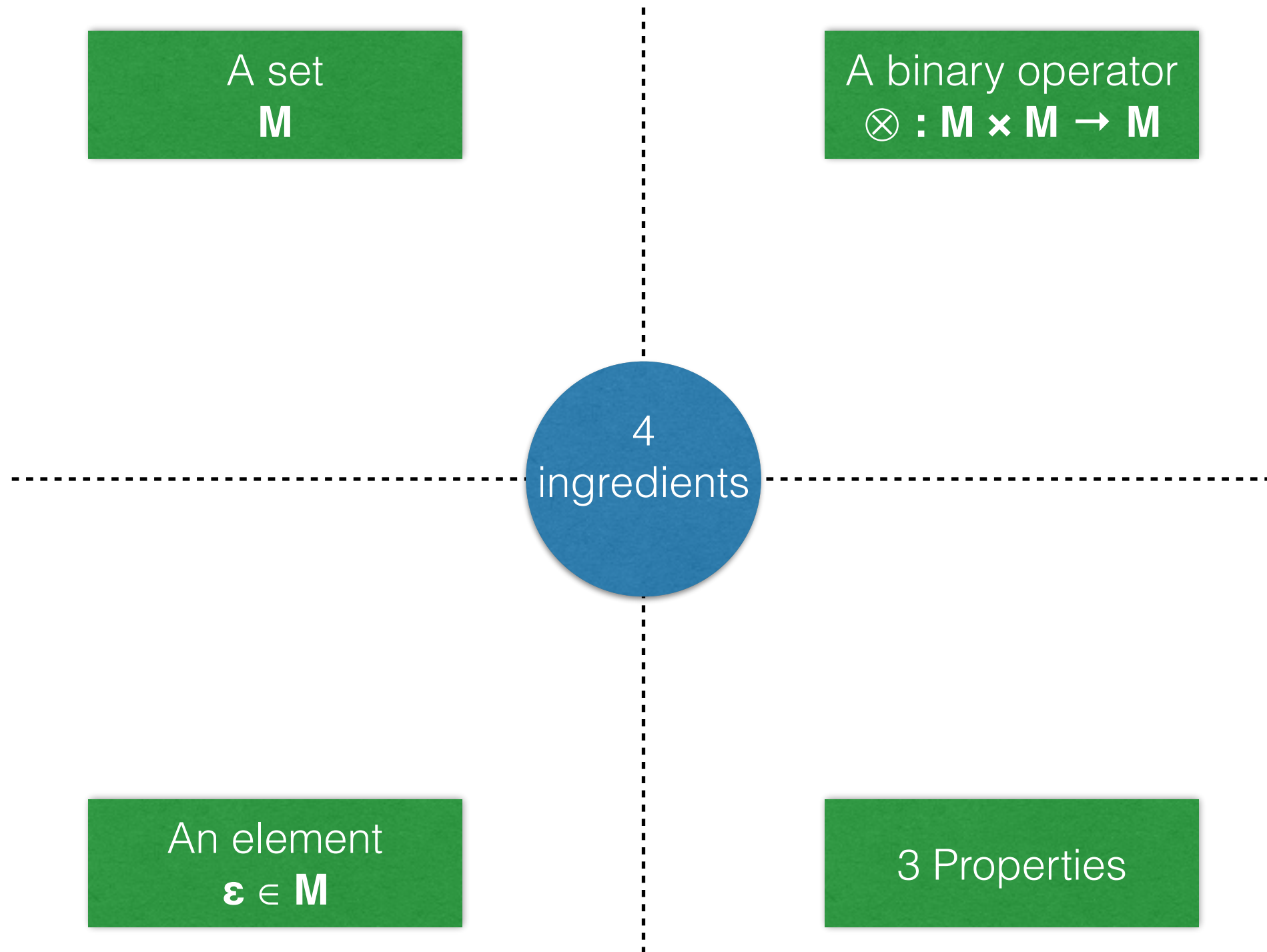
The Monoid Structure



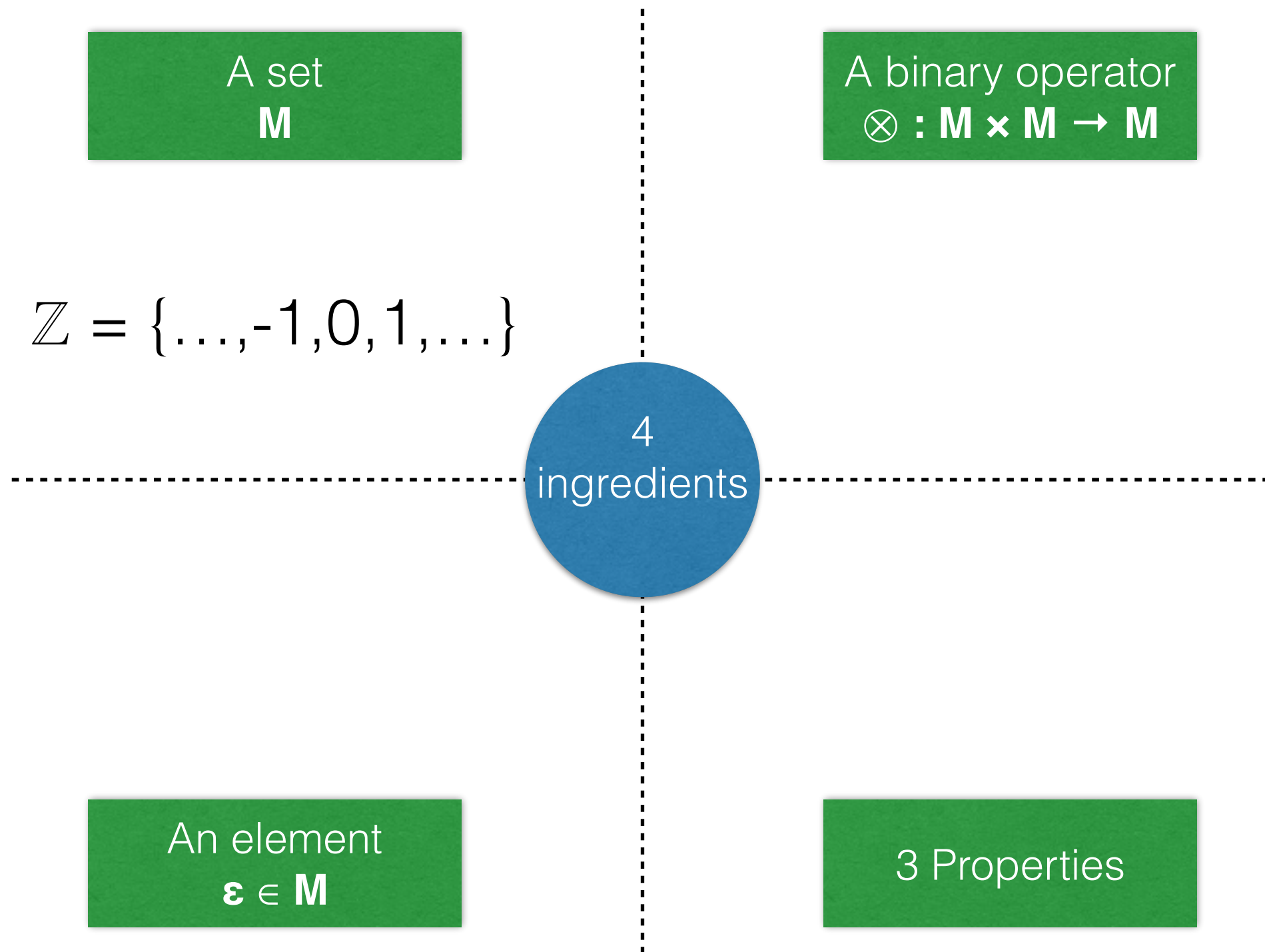
The Monoid Structure



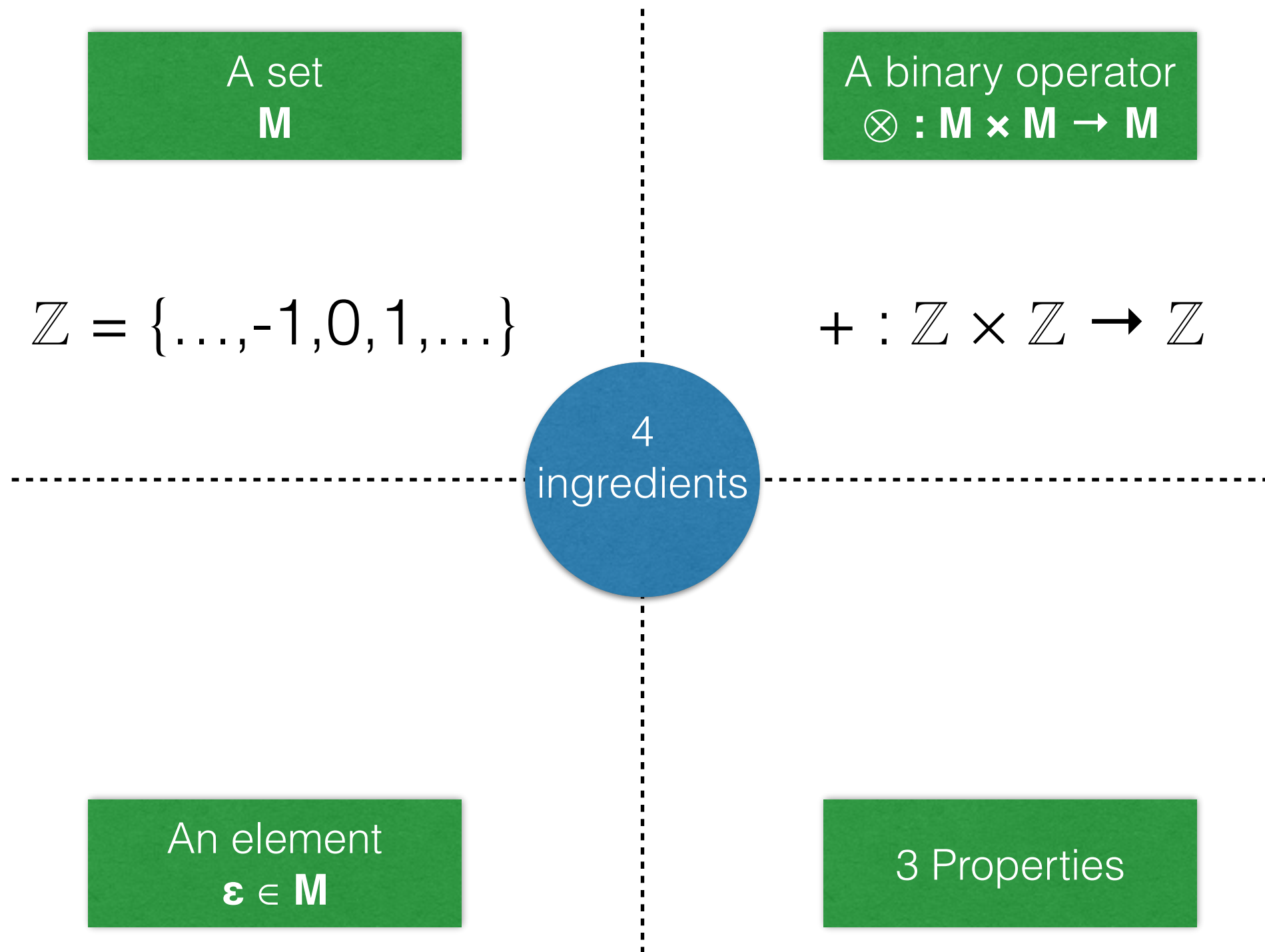
The Monoid Structure



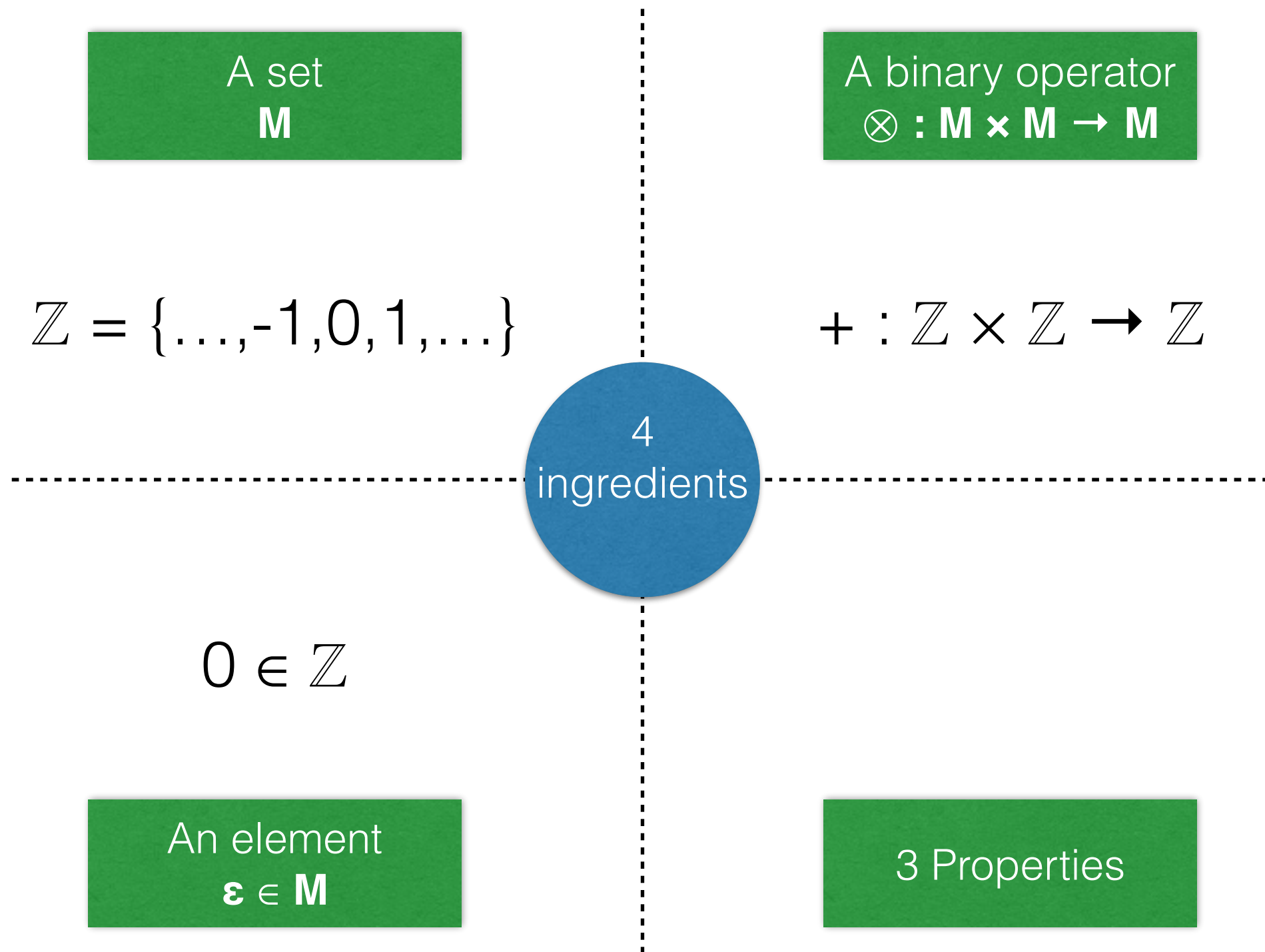
The Monoid Structure



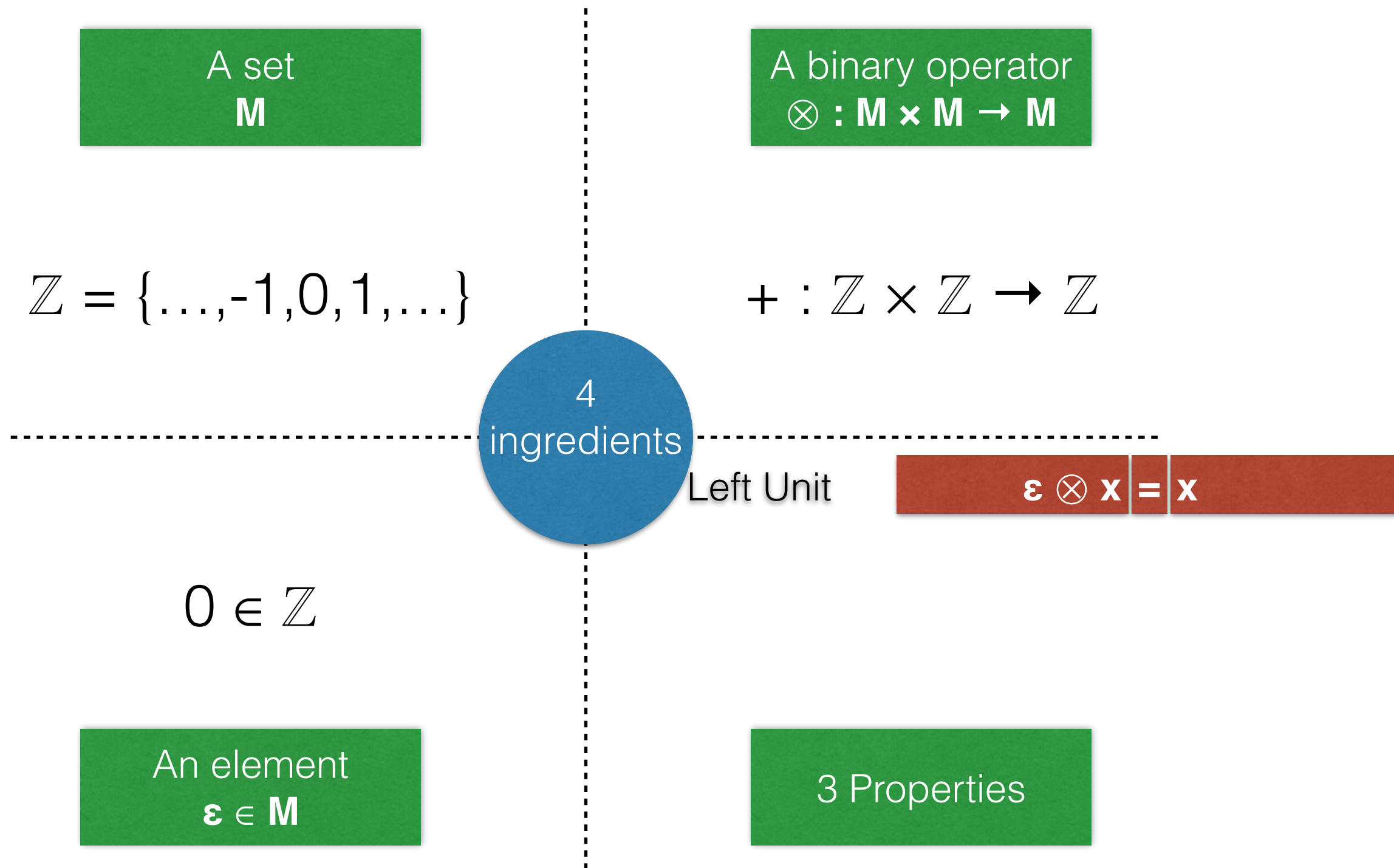
The Monoid Structure



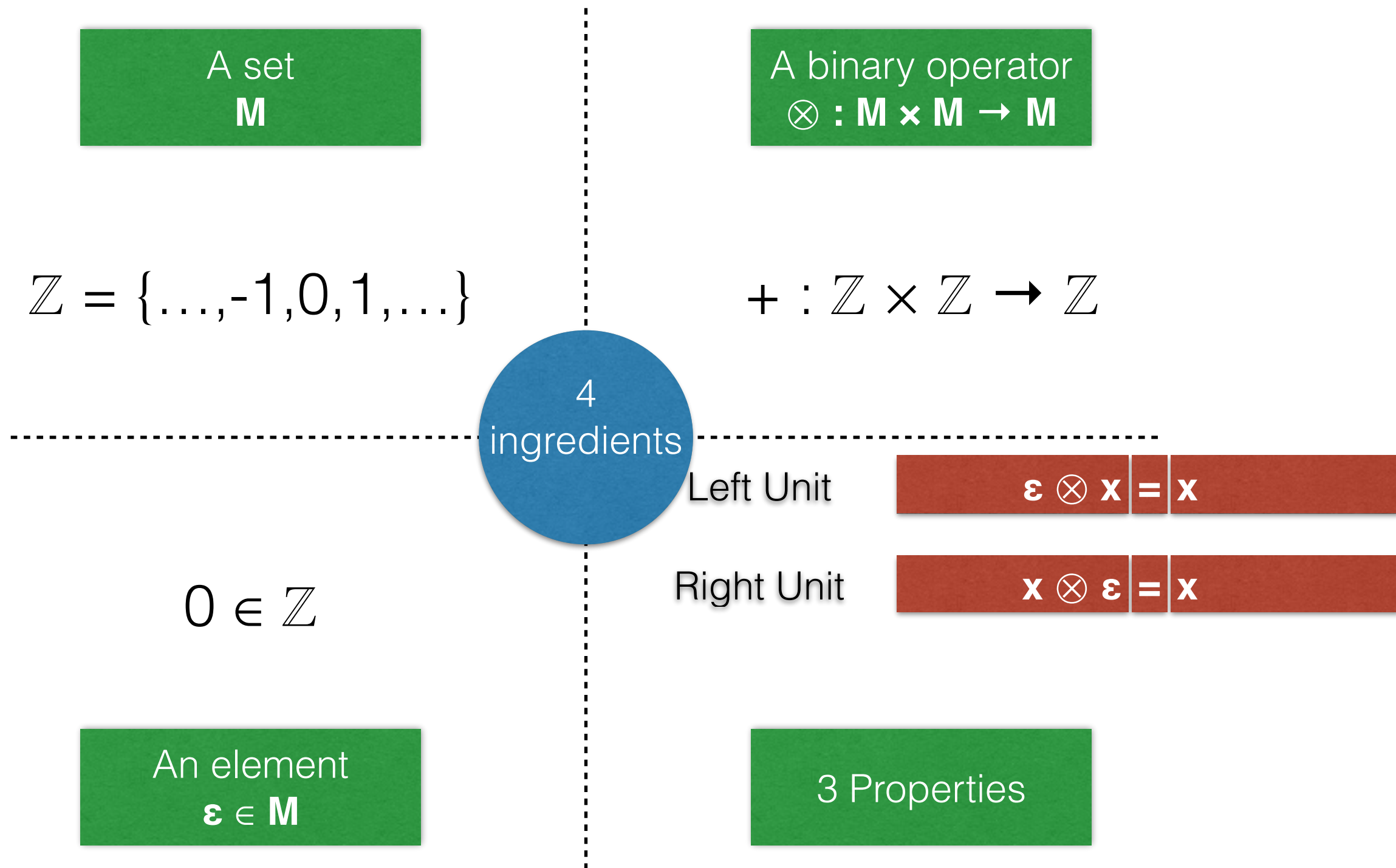
The Monoid Structure



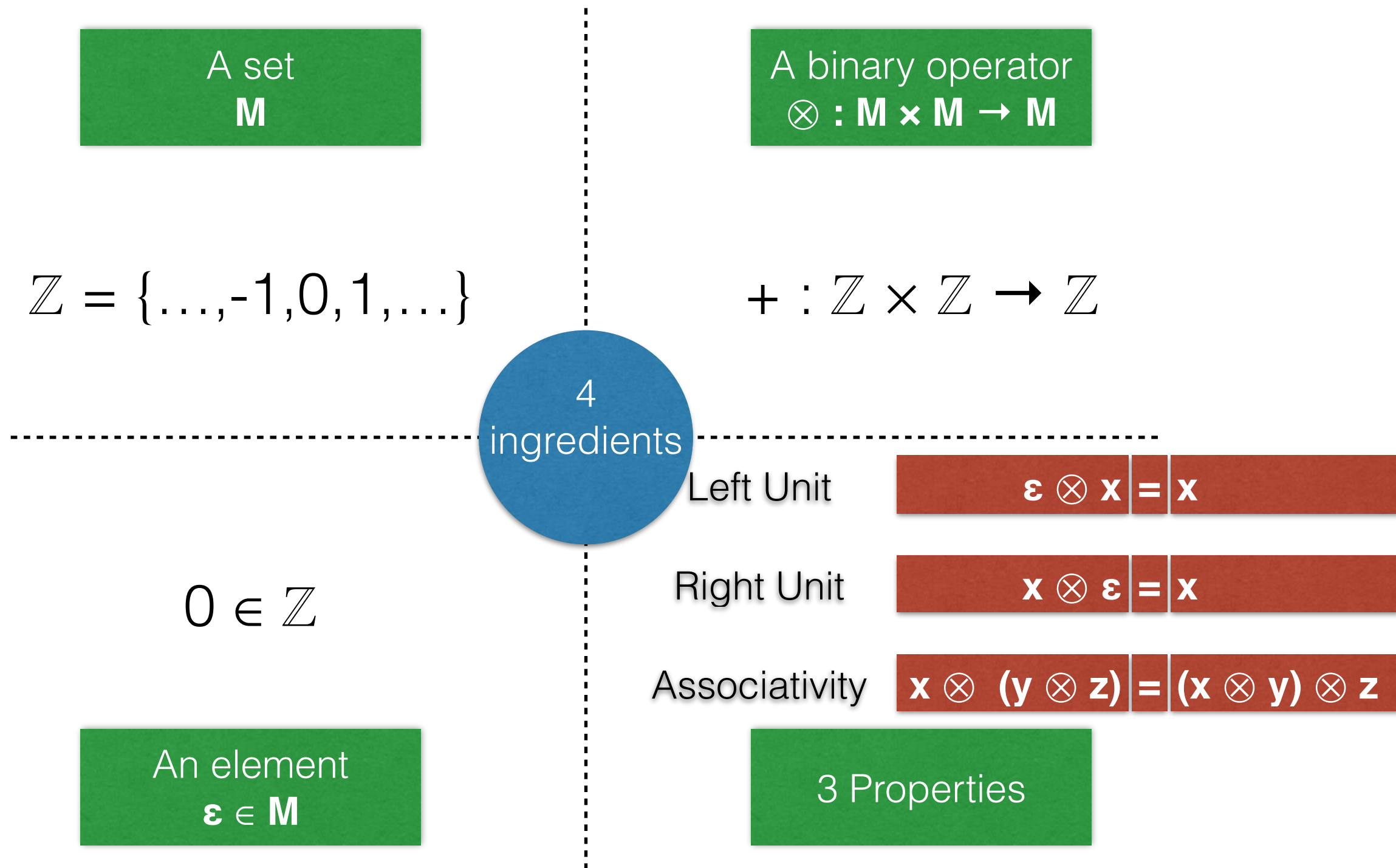
The Monoid Structure



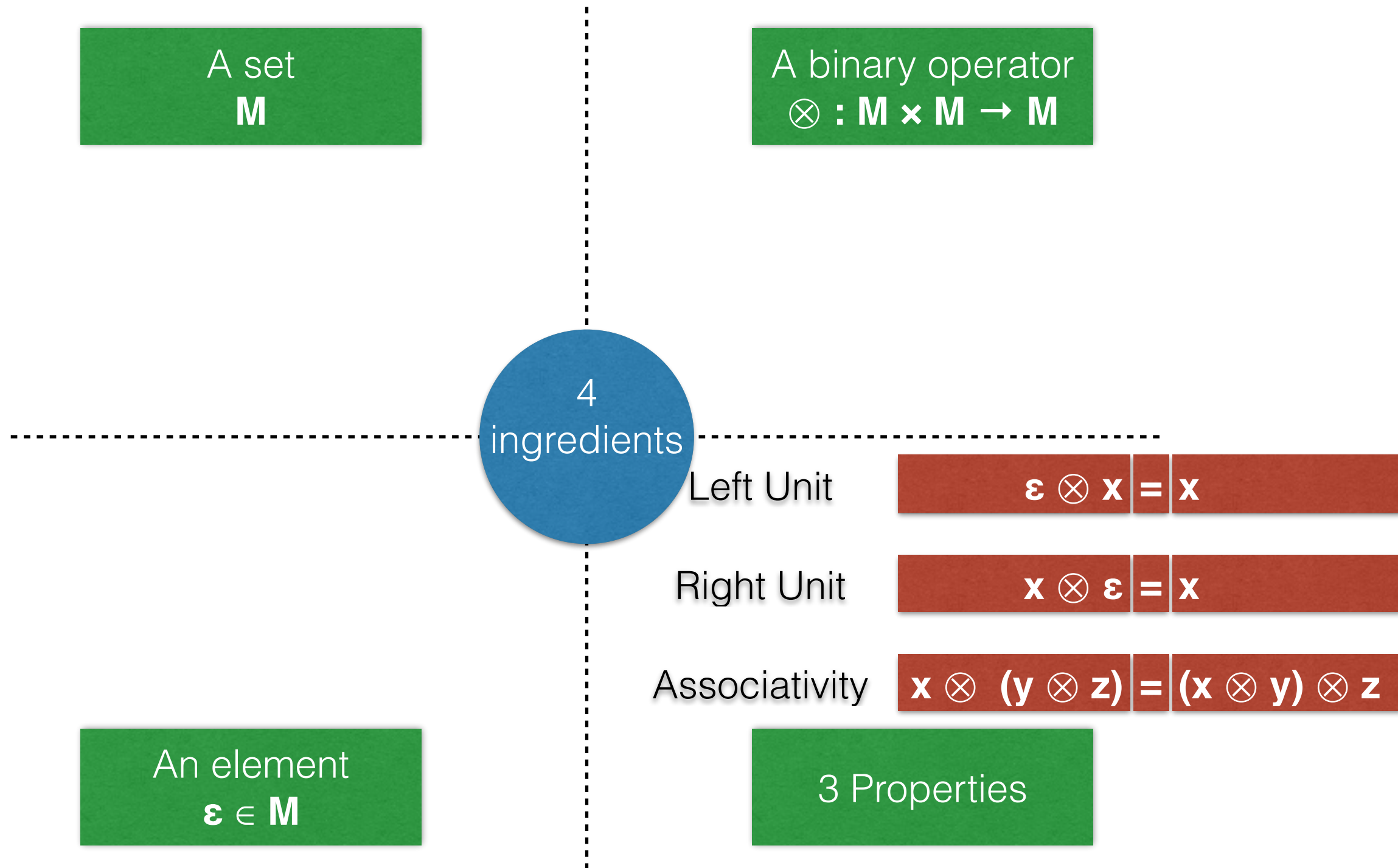
The Monoid Structure



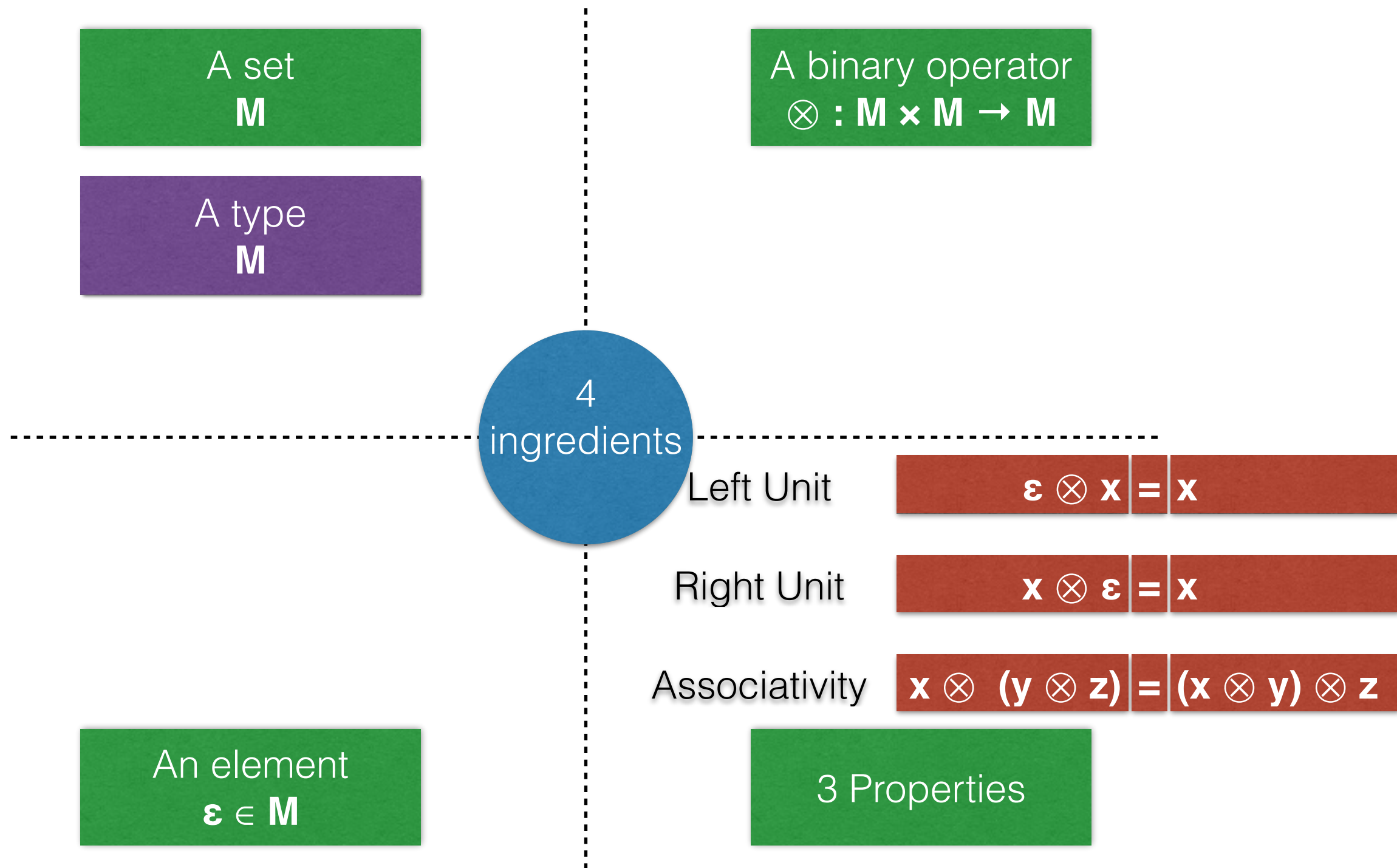
The Monoid Structure



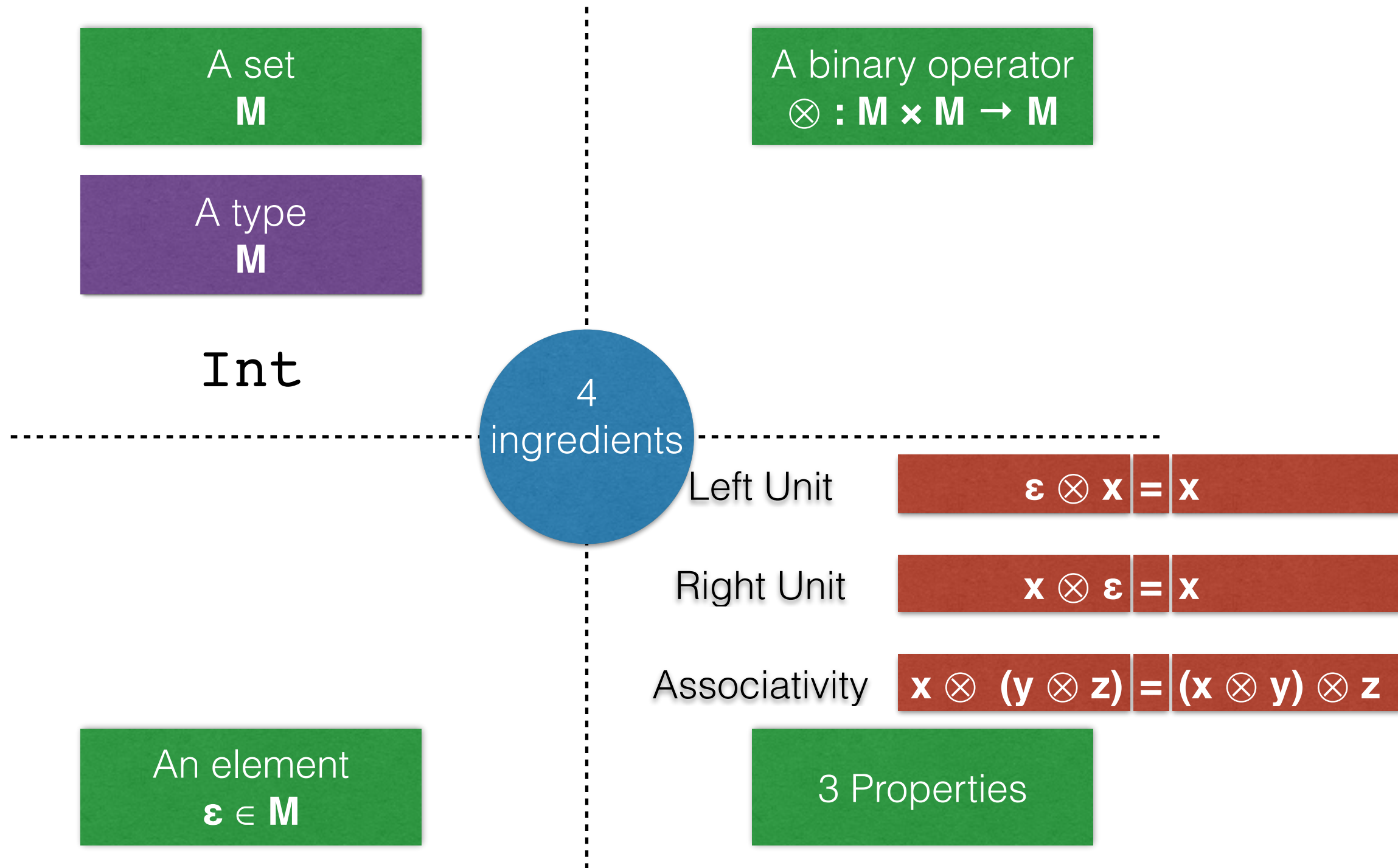
Haskell Monoids



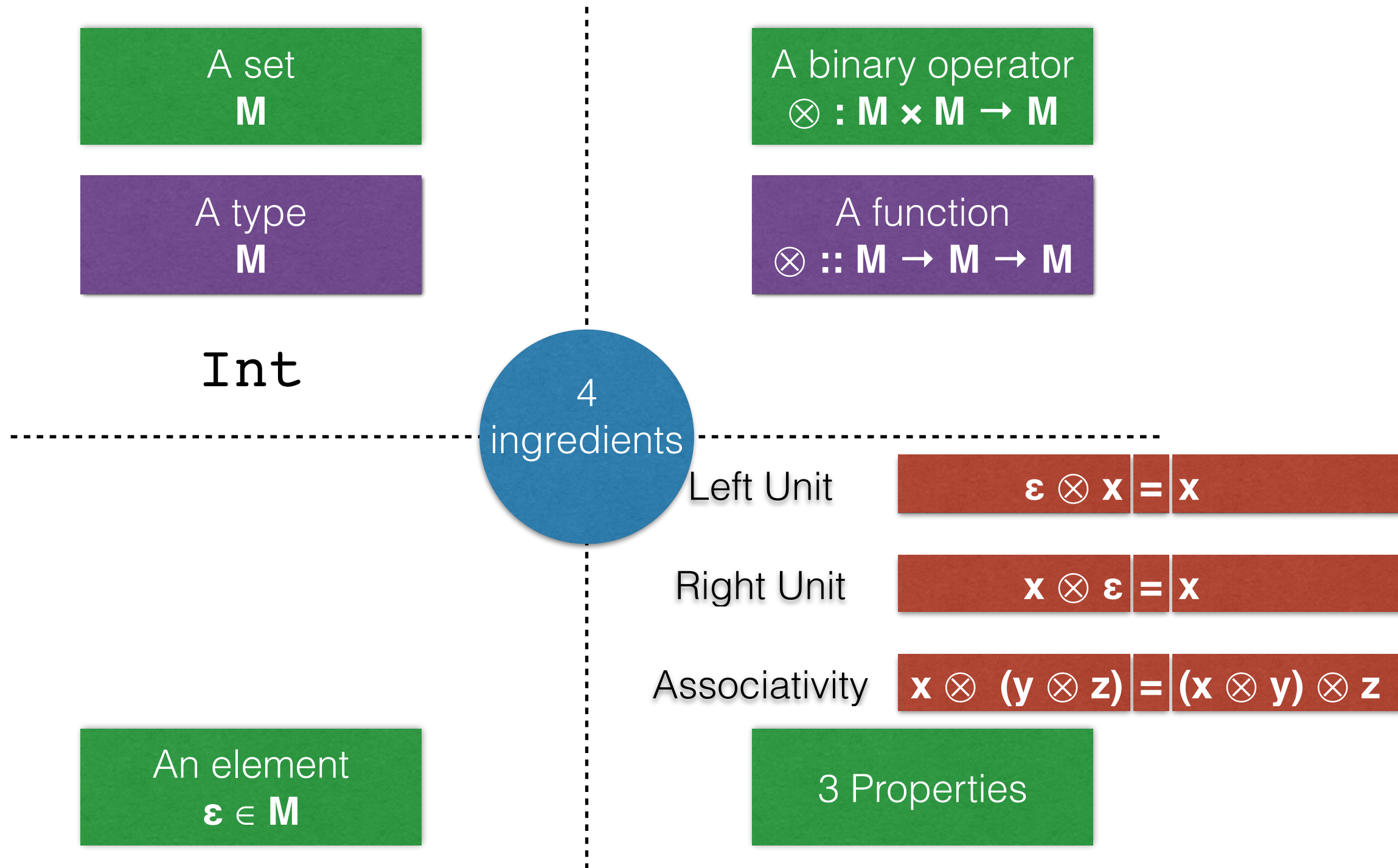
Haskell Monoids



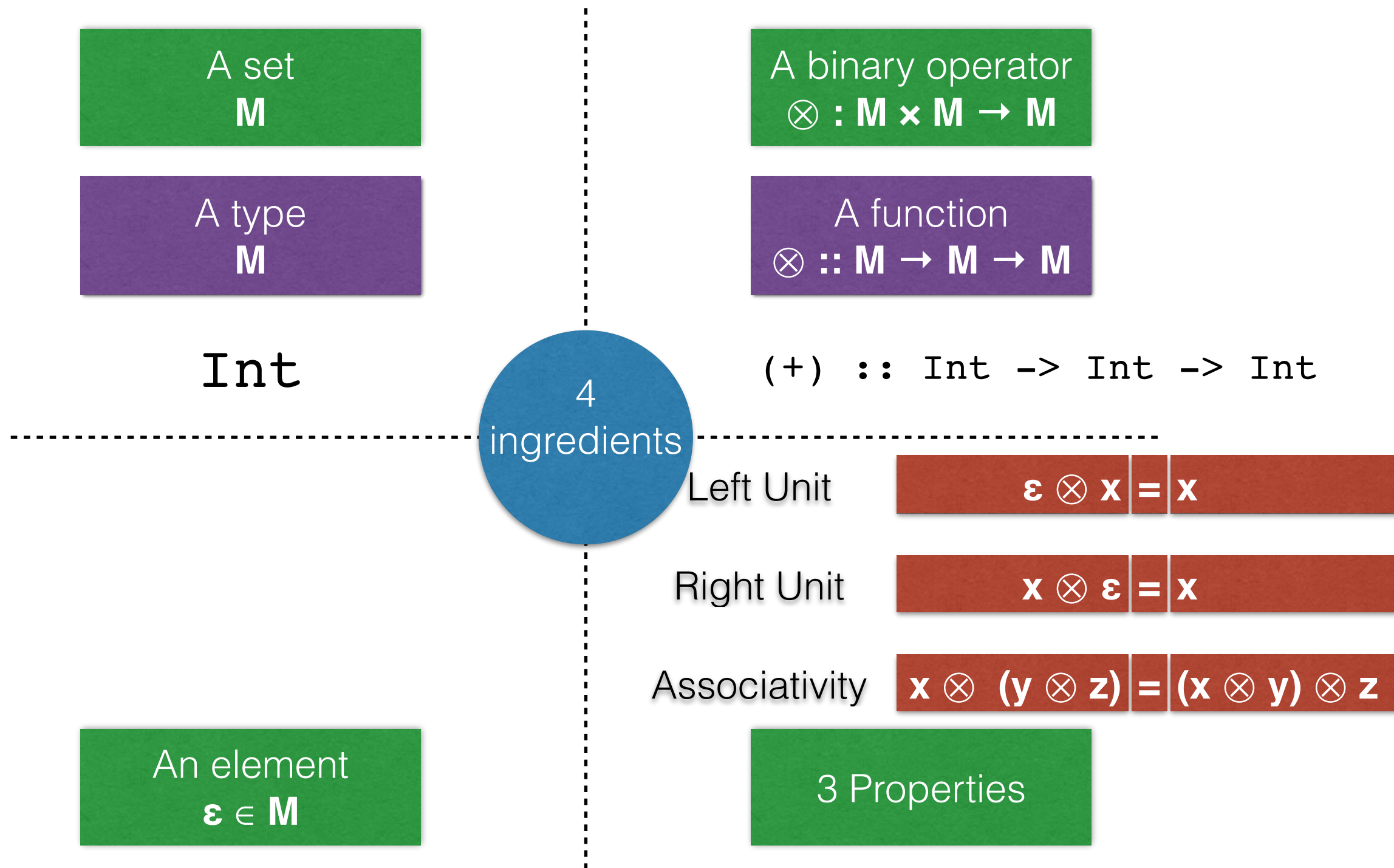
Haskell Monoids



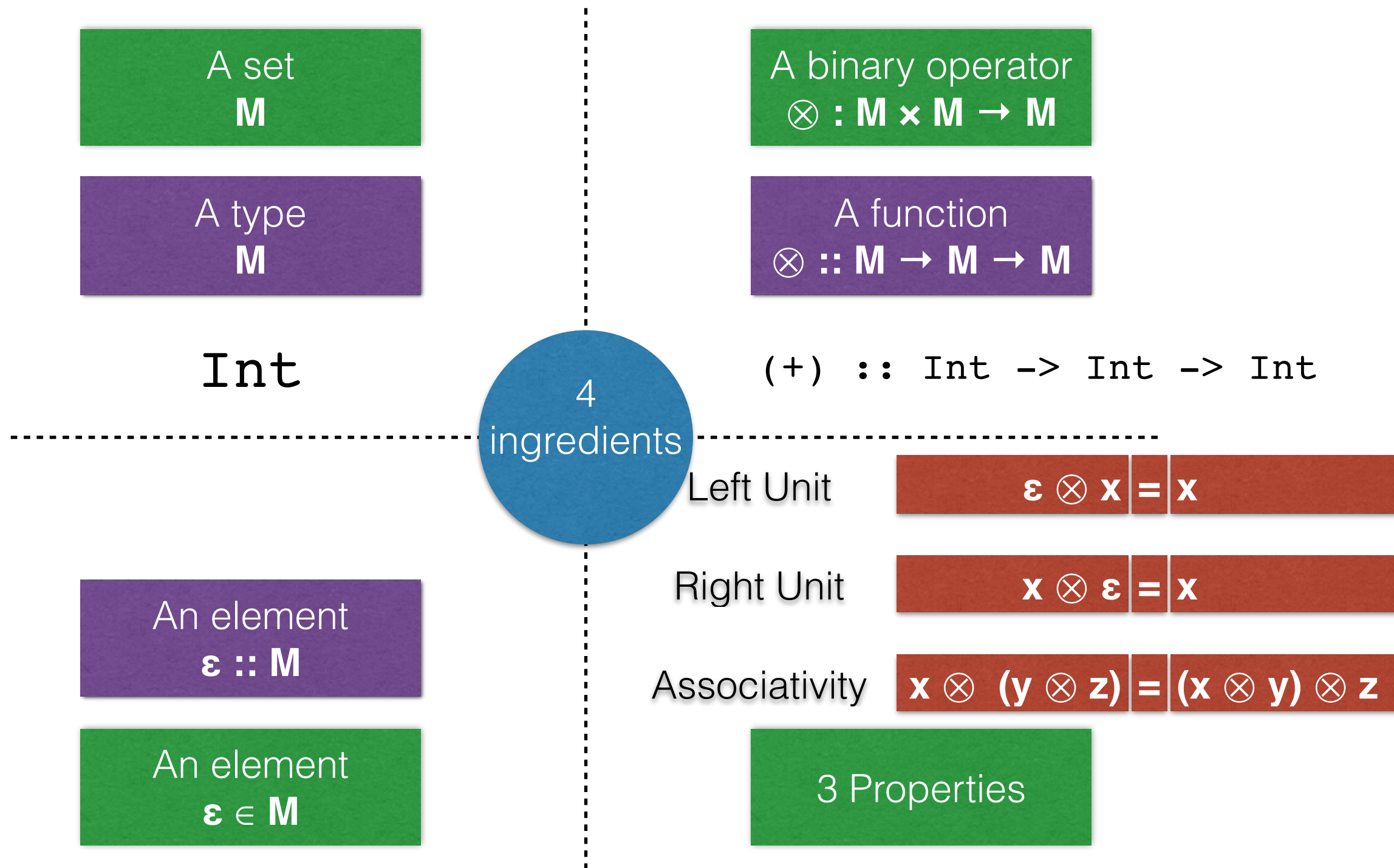
Haskell Monoids



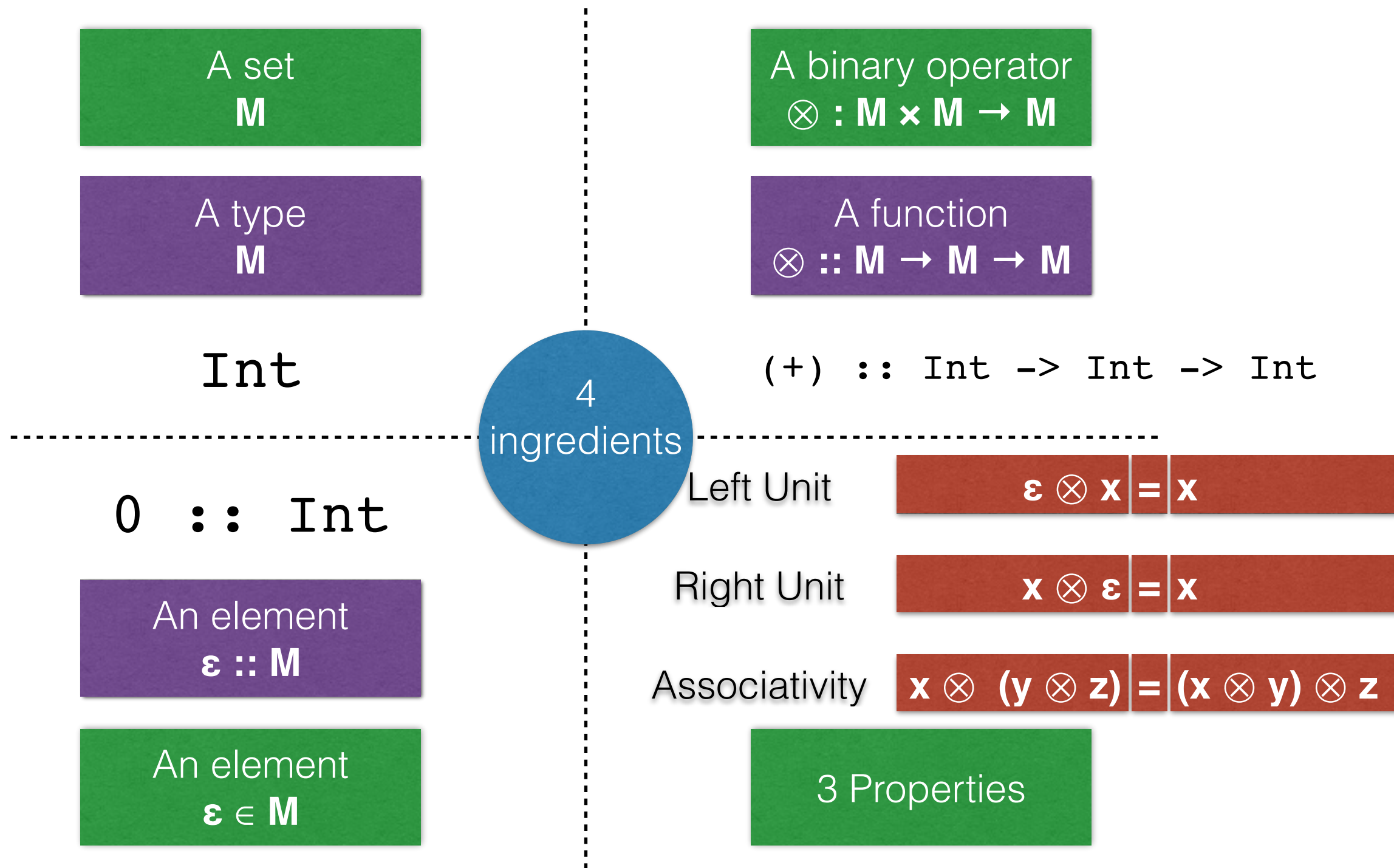
Haskell Monoids



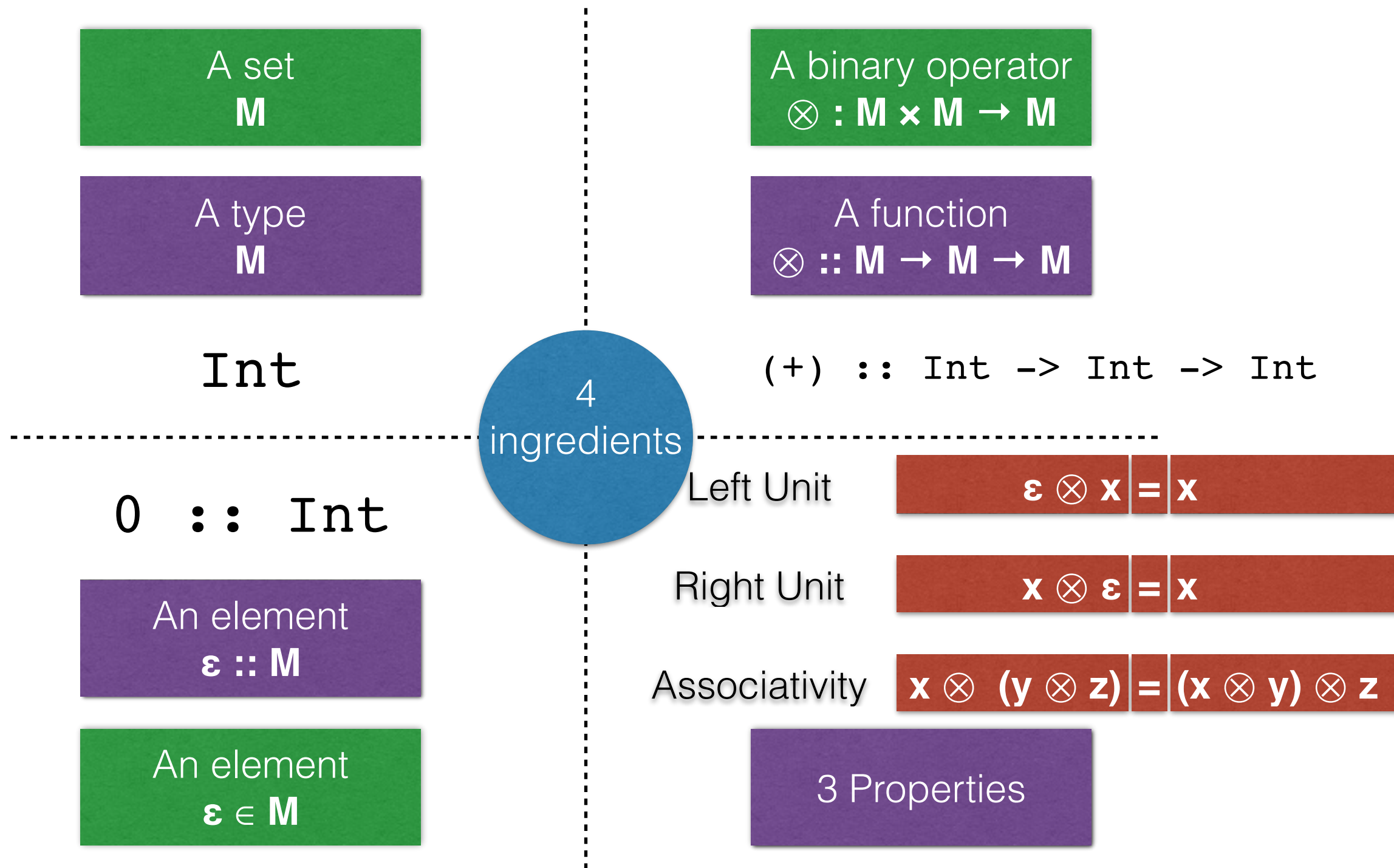
Haskell Monoids



Haskell Monoids



Haskell Monoids



Monoid Type Class

Monoid Type Class

A type
M

Monoid Type Class

A type
M

```
class Monoid m where
```

Monoid Type Class

```
class Monoid m where
```

A type
M

An element
 ε :: M

Monoid Type Class

```
class Monoid m where  
  mempty  :: m
```

A type
M

An element
 ε :: M

Monoid Type Class

```
class Monoid m where  
  mempty  :: m
```

A type
M

An element
 ε :: M

A function
 \otimes :: M → M → M

Monoid Type Class

```
class Monoid m where  
  mempty    :: m  
  mappend   :: m -> m -> m
```

A type
 M

An element
 $\varepsilon :: M$

A function
 $\otimes :: M \rightarrow M \rightarrow M$

Monoid Type Class

```
class Monoid m where  
  mempty    :: m  
  mappend   :: m -> m -> m
```

A type
 M

An element
 $\varepsilon :: M$

A function
 $\otimes :: M \rightarrow M \rightarrow M$

3 Properties

Monoid Type Class

```
class Monoid m where
```

```
  mempty  :: m
```

```
  mappend :: m -> m -> m
```

A type
 M

An element
 $\varepsilon :: M$

A function
 $\otimes :: M \rightarrow M \rightarrow M$

3 Properties

not in the
Haskell
language



Monoid Type Class

```
class Monoid m where  
  mempty    :: m  
  mappend   :: m -> m -> m
```

A type
 M

An element
 $\varepsilon :: M$

A function
 $\otimes :: M \rightarrow M \rightarrow M$

3 Properties

$(\langle \rangle) = \text{mappend}$

convenient binary
operator

Monoid Instance

```
class Monoid m where
```

```
    mempty    :: m
```

```
    mappend   :: m -> m -> m
```

```
instance Monoid Int where
```

```
    mempty    = 0
```

```
    mappend   = (+)
```

Monoid for Bool?

```
class Monoid m where
```

```
    mempty    :: m
```

```
    mappend   :: m -> m -> m
```

```
instance Monoid Bool where
```

```
    mempty    = ???
```

```
    mappend   = ???
```

Two Possible Instances

Two Possible Instances

```
instance Monoid Bool where  
  mempty    = True  
  mappend   = ( && )
```


Two Possible Instances

```
instance Monoid Bool where  
  mempty    = True  
  mappend   = (&&)
```

```
instance Monoid Bool where  
  mempty    = False  
  mappend   = (||)
```

Two Possible Instances

```
instance Monoid Bool where
```

```
    mempty    = True
```

```
    mappend  = (&&)
```

Can't have two
instances for the
same type!!!

```
instance Monoid Bool where
```

```
    mempty    = False
```

```
    mappend  = (||)
```

What's in a name? that which we call a *Bool*
By any other name would smell as sweet;



Newtype to the Rescue!

What's in a name? that which we call a *Bool*
By any other name would smell as sweet;



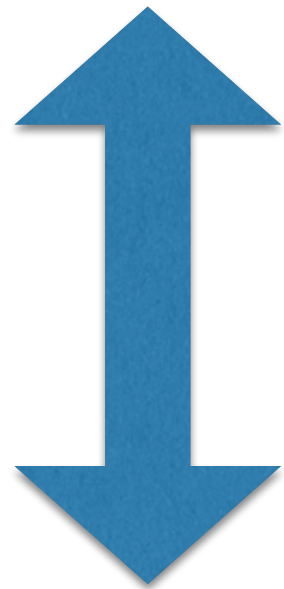
Newtype to the Rescue!

```
newtype All = All { getAll :: Bool }
```

```
newtype Any = Any { getAny :: Bool }
```

Newtype to the Rescue!

```
newtype A11 = A11 { getA11 :: Bool }
```



Both isomorphic to
Bool

```
newtype Any = Any { getAny :: Bool }
```

Newtype to the Rescue!

```
newtype All = All { getAll :: Bool }
```

```
instance Monoid All where
```

```
    mempty    = All True
```

```
    All x `mappend` All y = All (x && y)
```

```
newtype Any = Any { getAny :: Bool }
```


Newtype to the Rescue!

```
newtype All = All { getAll :: Bool }
```

```
instance Monoid All where
```

```
    mempty = All True
```

```
    All x `mappend` All y = All (x && y)
```

```
newtype Any = Any { getAny :: Bool }
```

```
instance Monoid Any where
```

```
    mempty = Any False
```

```
    Any x `mappend` Any y = Any (x || y)
```

Newtype to the Rescue!

```
newtype All = All { getAll :: Bool }
```

```
instance Monoid All where
```

```
    mempty = All True
```

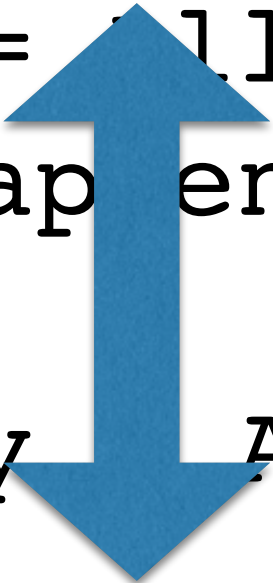
```
    All x `mappend` All y = All (x && y)
```

```
newtype Any = Any { getAny :: Bool }
```

```
instance Monoid Any where
```

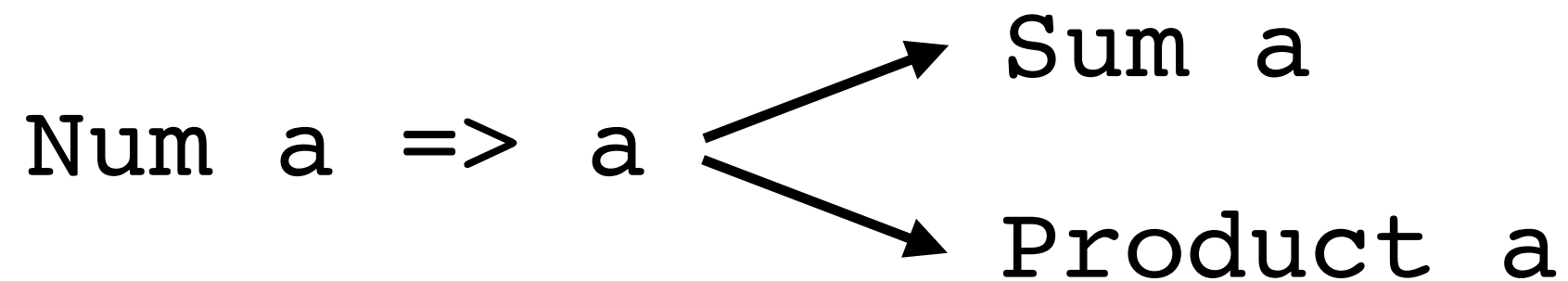
```
    mempty = Any False
```

```
    Any x `mappend` Any y = Any (x || y)
```

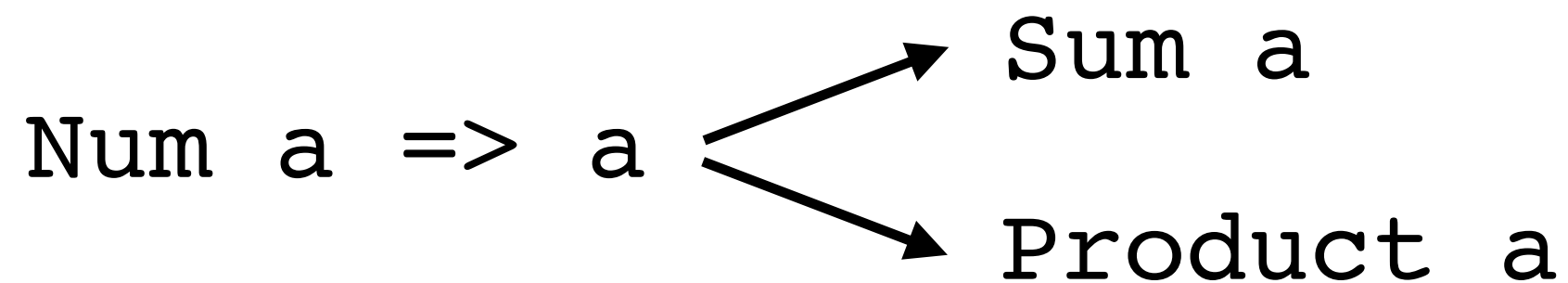


Two non-conflicting
instances

Same Problem for Num



Same Problem for Num



Homework

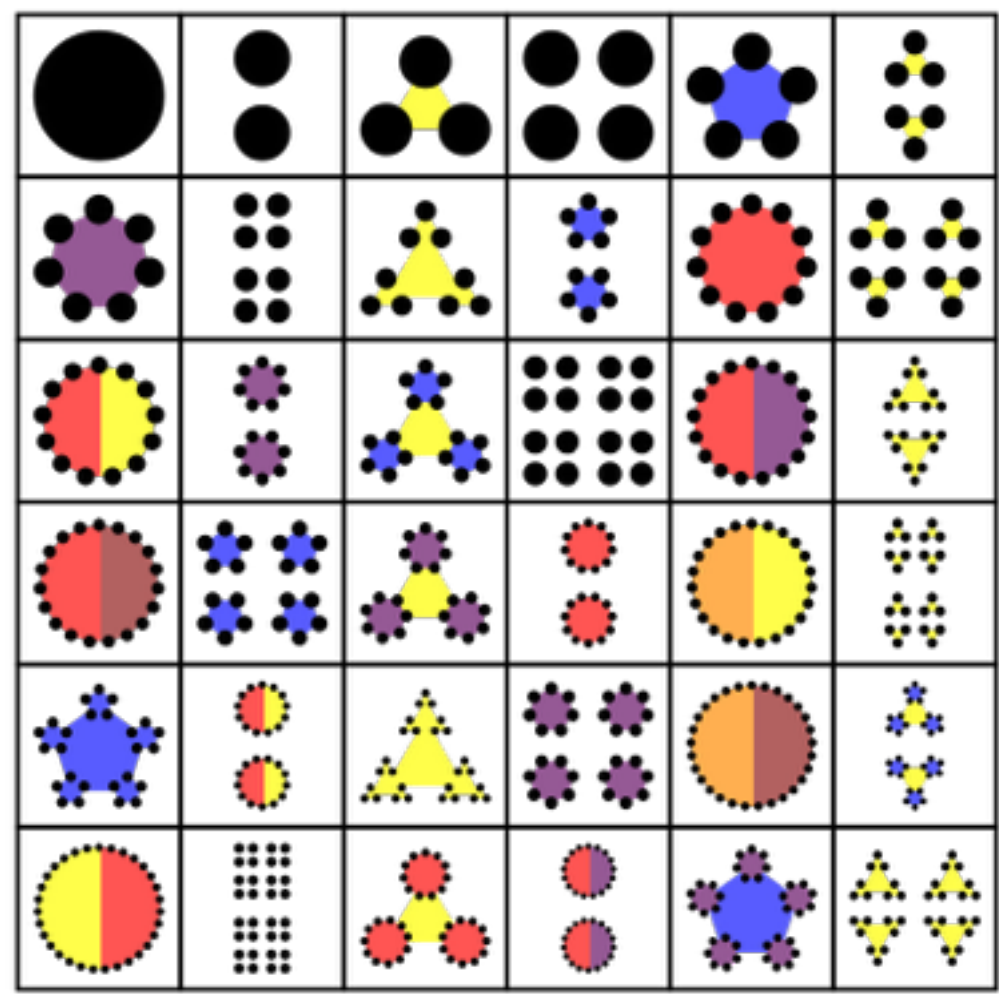
Invent 10 more monoid structures for Int

A large, stylized 'X' logo composed of two overlapping chevron shapes, rendered in a dark purple color. It is centered in the background of the slide.

Applications



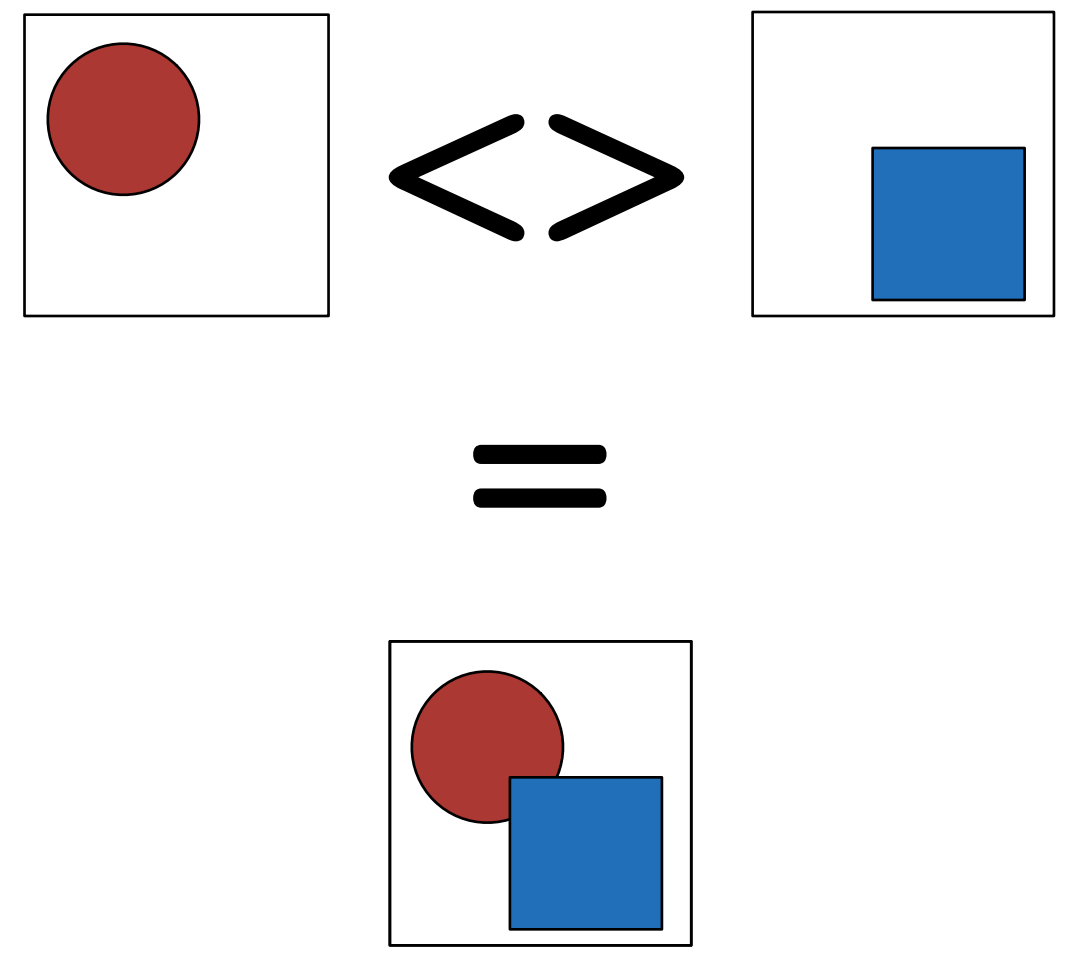
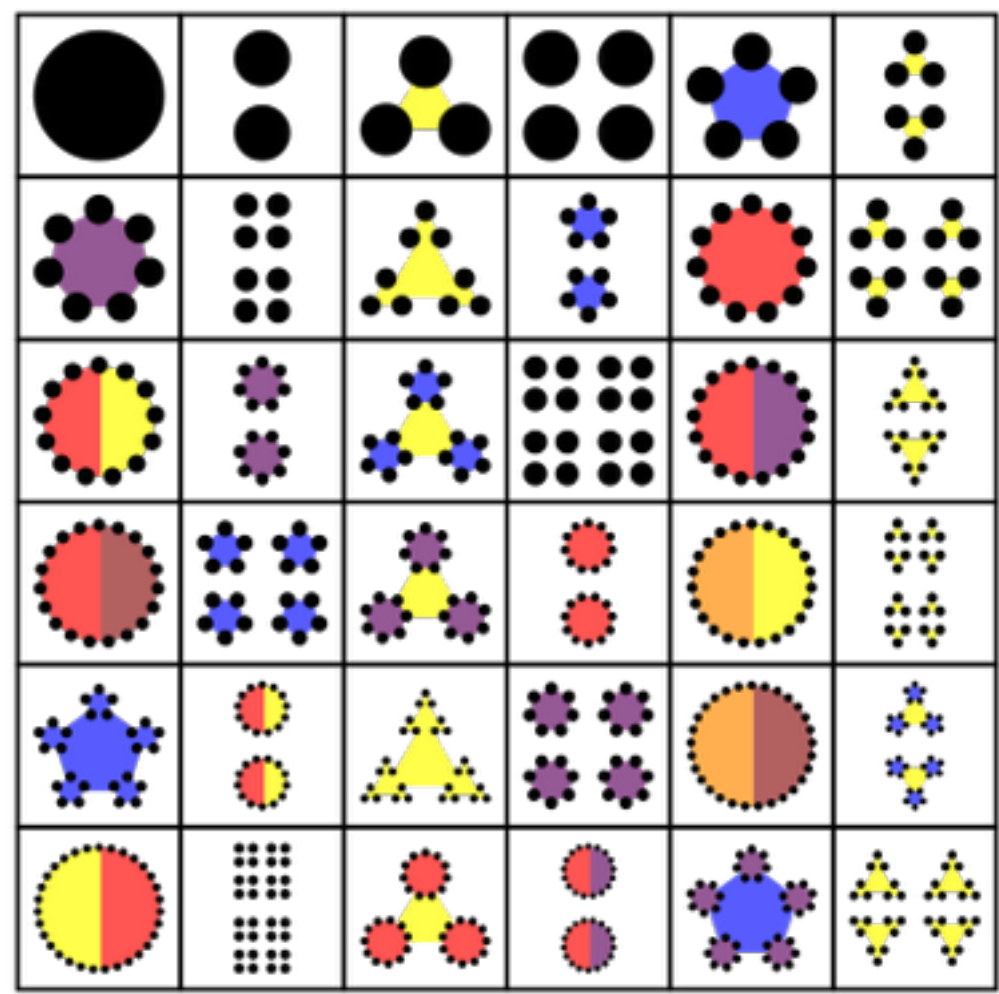
The diagrams Package



Brent Yorgey



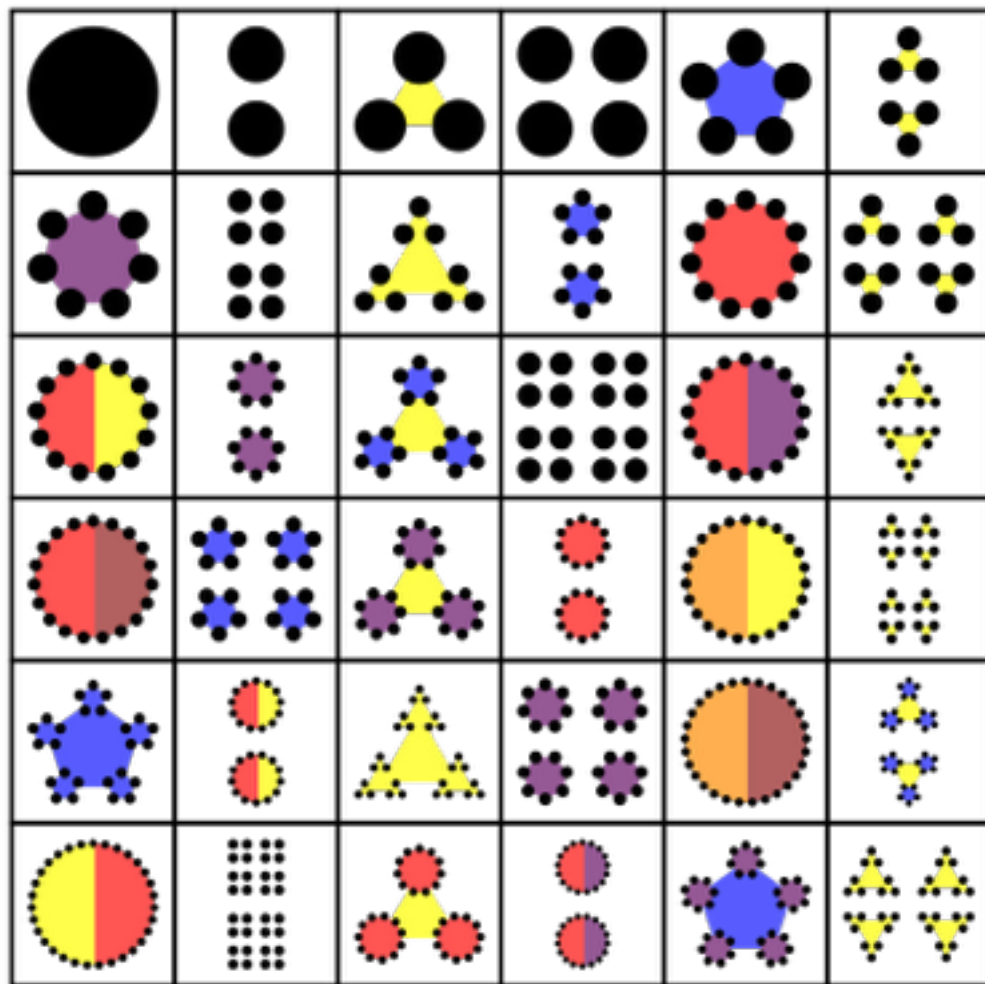
The diagrams Package



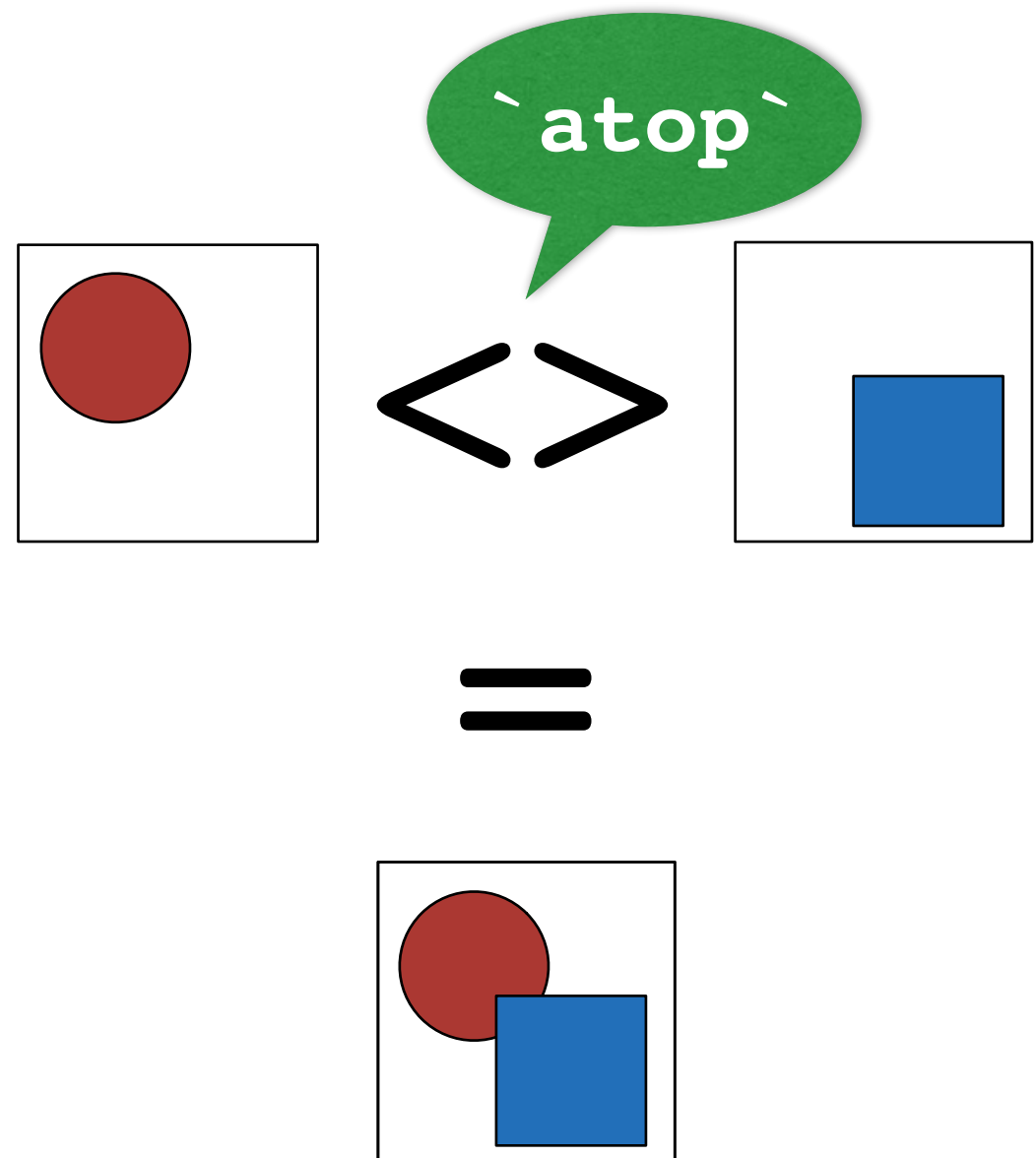
Brent Yorgey

1

The diagrams Package



Brent Yorgey



2

Compositional Settings

e.g., Command-Line Options:

```
--enable-foo --disable-foo --enable-foo --baz=help
```



Duncan Coutts

2

Compositional Settings

e.g., Command-Line Options:

--enable-foo --disable-foo --enable-foo --baz=help



Duncan Coutts

default setting <> setting update <> ... <> setting update

Compositional Settings

```
data ConfigFlags =  
  ConfigFlags {  
    foo :: Flag Bool,  
    bar :: Flag PackageDB,  
    baz :: [String]  
  }
```

Compositional Settings

```
data ConfigFlags =  
  ConfigFlags {  
    foo :: Flag Bool,  
    bar :: Flag PackageDB,  
    baz :: [String]  
  }
```

```
data Flag a = Flag a | Default  
  
instance Monoid (Flag a) where  
  mempty = Default  
  _ `mappend` f@(Flag _) = f  
  f `mappend` Default    = f
```

right biased

Compositional Settings

```
instance Monoid ConfigFlags where
```

```
  mempty =
```

```
    ConfigFlags
```

```
      { foo = mempty
```

```
      , bar = mempty
```

```
      , baz = mempty
```

```
      }
```

```
c1 `mappend` c2  =
```

```
  ConfigFlags
```

```
    { foo = foo c1 <> foo c2
```

```
    , bar = bar c1 <> bar c2
```

```
    , baz = baz c1 <> baz c2
```

```
    }
```

Data Aggregation

```
class Foldable t where  
  foldMap :: Monoid m  
           => (a -> m)  
           -> (t a -> m)
```


Data Aggregation

polymorphic collection

```
class Foldable t where  
  foldMap :: Monoid m  
           => (a -> m)  
           -> (t a -> m)
```

Data Aggregation

polymorphic collection

```
class Foldable t where  
  foldMap :: Monoid m  
           => (a -> m)  
           -> (t a -> m)
```

```
toList    :: Foldable t => t a -> [a]  
and       :: Foldable t => t Bool -> Bool  
or        :: Foldable t => t Bool -> Bool  
any       :: Foldable t => (a -> Bool) -> t a -> Bool  
all       :: Foldable t => (a -> Bool) -> t a -> Bool  
sum       :: (Foldable t, Num a) => t a -> a  
product   :: (Foldable t, Num a) => t a -> a  
maximum   :: (Foldable t, Ord a) => t a -> a  
minimum   :: (Foldable t, Ord a) => t a -> a  
elem      :: (Foldable t, Eq a) => a -> t a -> Bool
```

...

Data Aggregation

```
data Tree a
  = Empty
  | Fork (Tree a) a (Tree a)

instance Foldable Tree where
  foldMap gen Empty
    = mempty
  foldMap gen (Fork l x r)
    = foldMap gen l <> gen x <> foldMap gen r
```

Data Aggregation

```
data Tree a
  = Empty
  | Fork (Tree a) a (Tree a)

instance Foldable Tree where
  foldMap gen Empty
    = mempty
  foldMap gen (Fork l x r)
    = foldMap gen l <> gen x <> foldMap gen r
```

```
> sum (Fork (Fork Empty 5 Empty) 3 Empty)
8
> maximum (Fork (Fork Empty 5 Empty) 3 Empty)
5
```

Foldable/Traversable Proposal

```
toList    :: Foldable t => t a -> [a]
and        :: Foldable t => t Bool -> Bool
or         :: Foldable t => t Bool -> Bool
any        :: Foldable t => (a -> Bool) -> t a -> Bool
all        :: Foldable t => (a -> Bool) -> t a -> Bool
sum        :: (Foldable t, Num a) => t a -> a
product    :: (Foldable t, Num a) => t a -> a
maximum    :: (Foldable t, Ord a) => t a -> a
minimum    :: (Foldable t, Ord a) => t a -> a
elem       :: (Foldable t, Eq a) => a -> t a -> Bool
```

...

Foldable/Traversable Proposal

aka
Burning Bridges
Proposal

```
toList    :: Foldable t => t a -> [a]
and        :: Foldable t => t Bool -> Bool
or         :: Foldable t => t Bool -> Bool
any        :: Foldable t => (a -> Bool) -> t a -> Bool
all        :: Foldable t => (a -> Bool) -> t a -> Bool
sum        :: (Foldable t, Num a) => t a -> a
product    :: (Foldable t, Num a) => t a -> a
maximum    :: (Foldable t, Ord a) => t a -> a
minimum    :: (Foldable t, Ord a) => t a -> a
elem       :: (Foldable t, Eq a) => a -> t a -> Bool
```

...

Foldable/Traversable Proposal

aka
Burning Bridges
Proposal

In Prelude as of
GHC 7.10

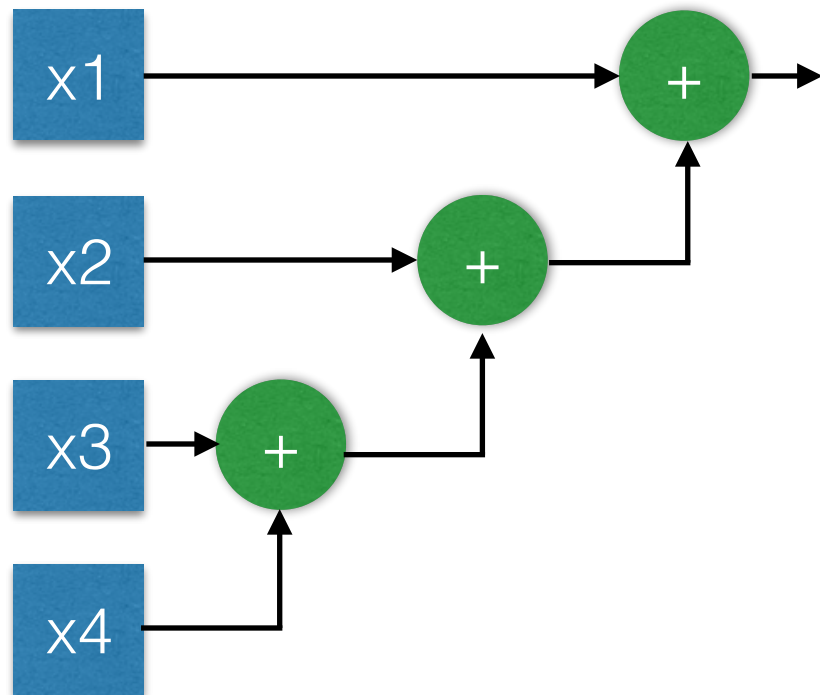
```
toList    :: Foldable t => t a -> [a]
and        :: Foldable t => t Bool -> Bool
or         :: Foldable t => t Bool -> Bool
any        :: Foldable t => (a -> Bool) -> t a -> Bool
all        :: Foldable t => (a -> Bool) -> t a -> Bool
sum        :: (Foldable t, Num a) => t a -> a
product    :: (Foldable t, Num a) => t a -> a
maximum    :: (Foldable t, Ord a) => t a -> a
minimum    :: (Foldable t, Ord a) => t a -> a
elem       :: (Foldable t, Eq a) => a -> t a -> Bool
```

...

A large, stylized, light purple 'X' graphic is centered in the background. It is composed of two overlapping 'X' shapes, one slightly offset from the other, creating a sense of depth. The 'X' is made of thick, diagonal strokes.

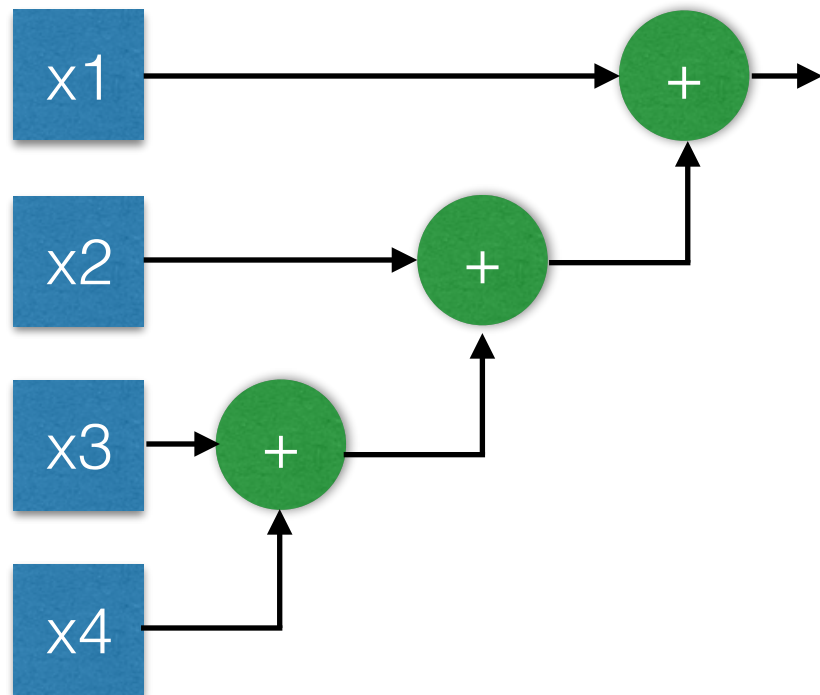
Divide & Conquer

Linear Processing



Linear Processing

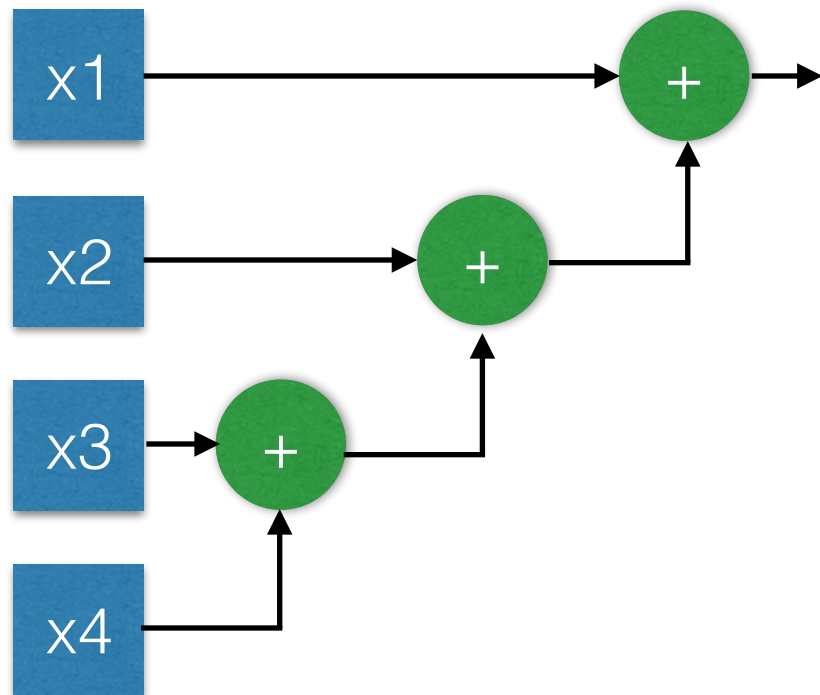
$x1 \text{ } \langle \rangle \text{ } (x2 \text{ } \langle \rangle \text{ } (x3 \text{ } \langle \rangle \text{ } x4))$



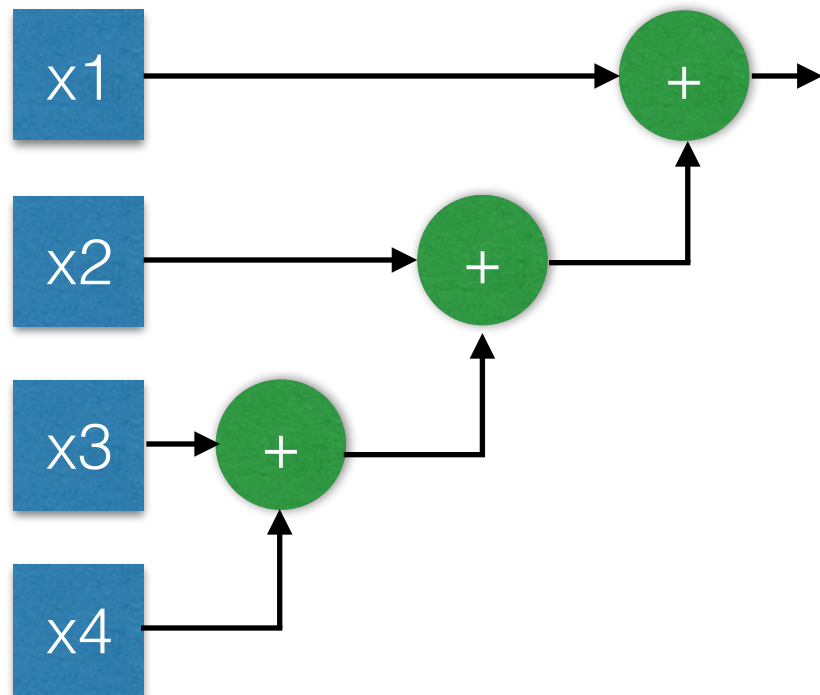
Linear Processing

$x1 \langle \rangle (x2 \langle \rangle (x3 \langle \rangle x4))$

$x1 \langle \rangle (x2 \langle \rangle x34)$



Linear Processing

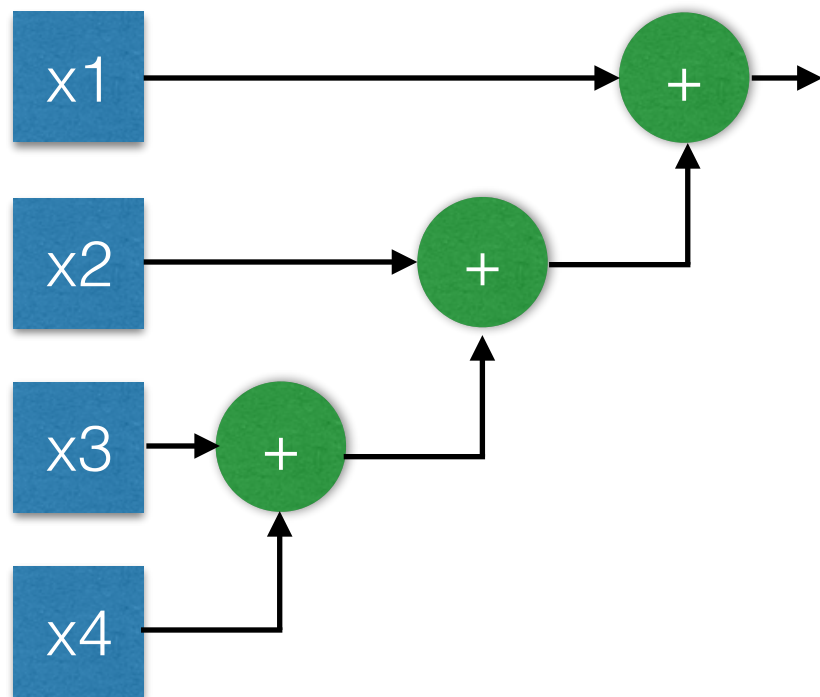


$x1 \langle \rangle (x2 \langle \rangle (x3 \langle \rangle x4))$

$x1 \langle \rangle (x2 \langle \rangle x34)$

$x1 \langle \rangle (x234)$

Linear Processing



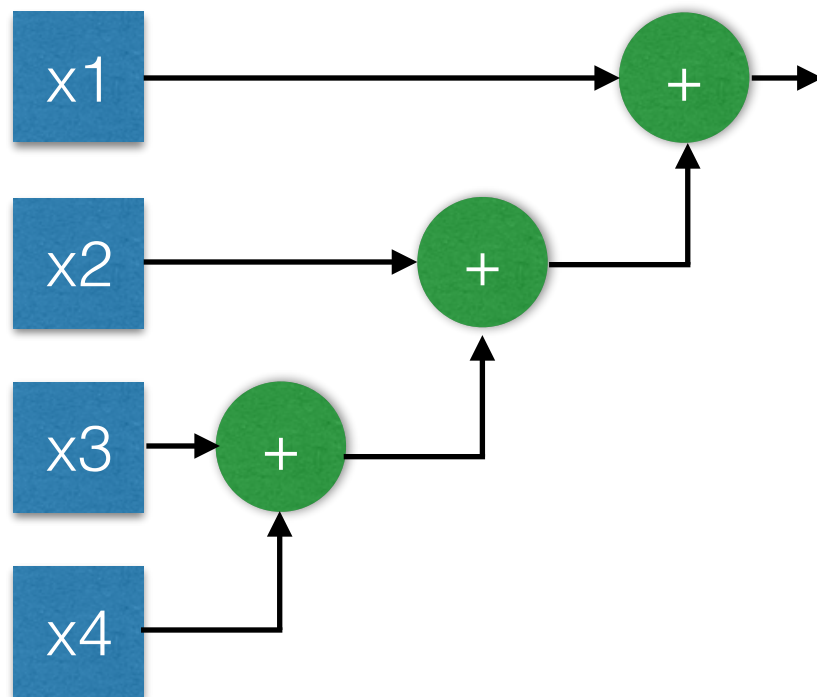
$x1 \langle \rangle (x2 \langle \rangle (x3 \langle \rangle x4))$

$x1 \langle \rangle (x2 \langle \rangle x34)$

$x1 \langle \rangle (x234)$

$x1234$

Linear Processing



n-1 gate
delays

$x1 \lt \! > (x2 \lt \! > (x3 \lt \! > x4))$

$x1 \lt \! > (x2 \lt \! > x34)$

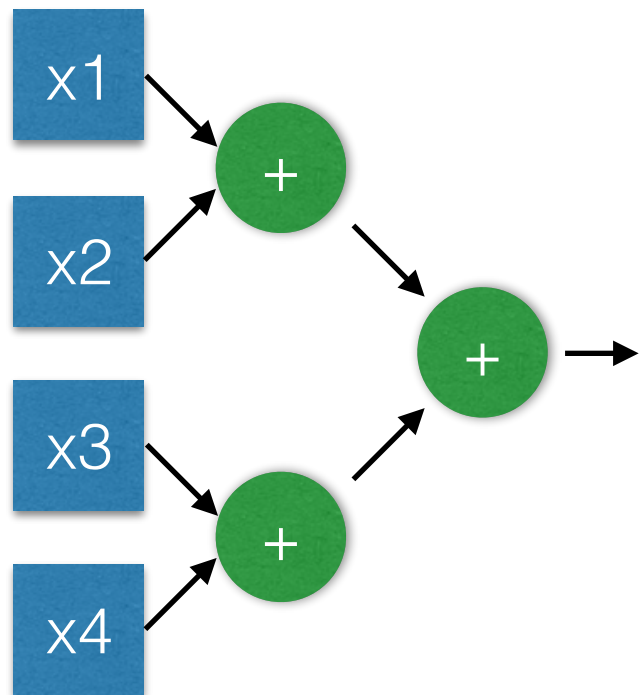
$x1 \lt \! > (x234)$

$x1234$

Linear Strategy

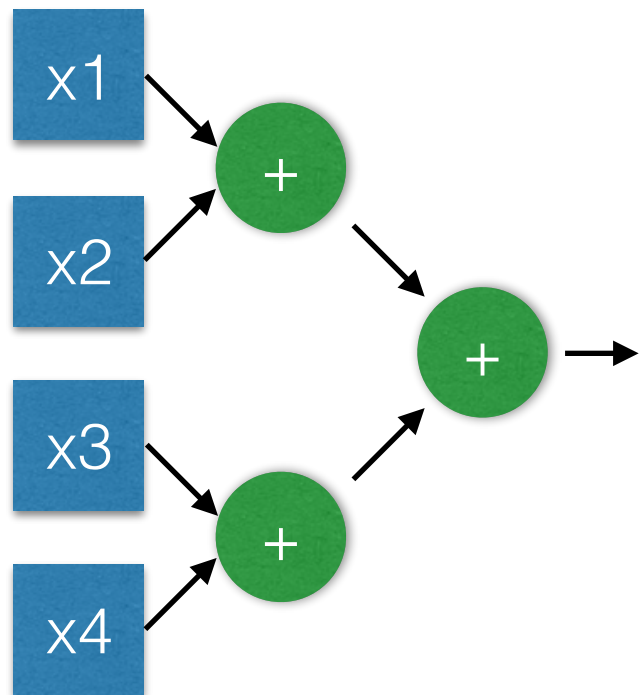
```
mconcat :: Monoid m => [m] -> m  
mconcat = foldr mappend mempty
```


Parallel Processing



Parallel Processing

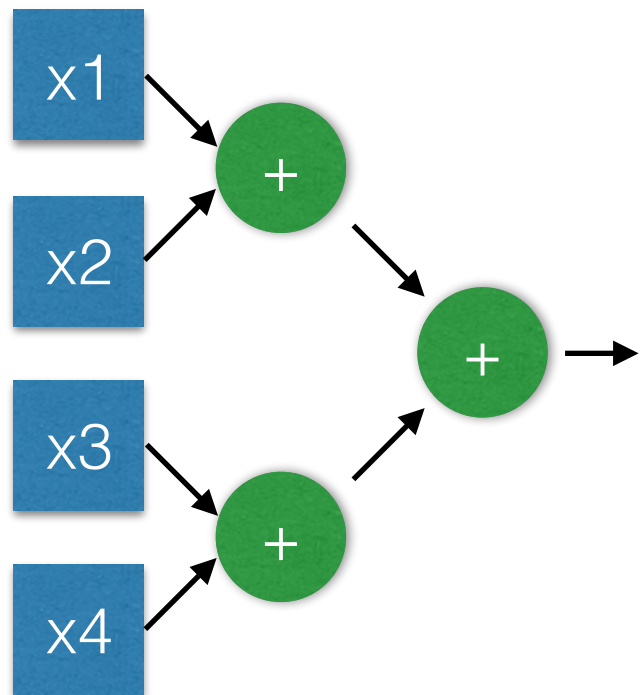
(x1 <> x2) <> (x3 <> x4)



Parallel Processing

(x1 <> x2) <> (x3 <> x4)

x12 <> x34

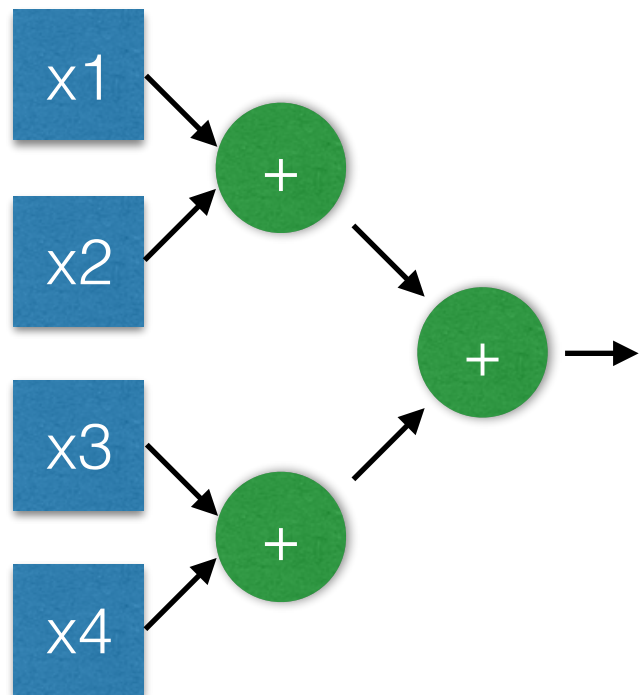


Parallel Processing

(x1 <> x2) <> (x3 <> x4)

x12 <> x34

x1234

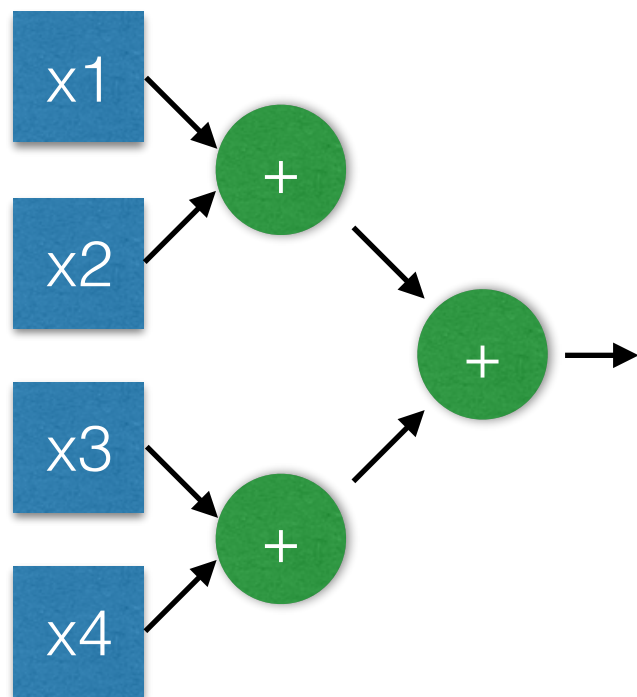


Parallel Processing

$$(x1 \text{ } \langle \rangle \text{ } x2) \text{ } \langle \rangle \text{ } (x3 \text{ } \langle \rangle \text{ } x4)$$

$$x12 \text{ } \langle \rangle \text{ } x34$$

$$x1234$$



log n gate
delays

Parallel Strategy

```
pconcat :: Monoid m => [m] -> m
pconcat [] = mempty
pconcat [x] = x
pconcat xs = (ys `par` zs) `pseq` (ys <> zs)
  where
    len = length xs
    (ys', zs') = splitAt (len `div` 2) xs
    ys = pconcat ys'
    zs = pconcat zs'
```



The List Monoid

The List Monoid

```
class Monoid m where  
    mempty    :: m  
    mappend   :: m -> m -> m  
  
instance Monoid [a] where  
    mempty    = []  
    mappend   = (++)
```


A large, stylized 'X' logo composed of two overlapping, semi-transparent, dark blue-grey geometric shapes. The shapes are composed of several parallel lines, giving it a layered, architectural appearance. It is centered in the background of the slide.

Equational Reasoning

Left Unit Proof

`mempty `mappend` ys`

`=`

**Proof Style:
Equational
Reasoning**

`ys`

[
[] ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
]

Left Unit Proof

`mempty `mappend` ys`
`= {- def. of mempty -}`

`=`

**Proof Style:
Equational
Reasoning**

`ys`

[`[] ++ ys = ys`
[`(x:xs) ++ ys = x : xs ++ ys`]

Left Unit Proof

```
empty `mappend` ys
= {- def. of empty -}
  [] `mappend` ys
=
```

**Proof Style:
Equational
Reasoning**

`ys`

```
[ [] ++ ys = ys
  (x:xs) ++ ys = x : xs ++ ys ]
```

Left Unit Proof

```
mempty `mappend` ys
= {- def. of mempty -}
  [] `mappend` ys
= {- def. of mappend -}
```

**Proof Style:
Equational
Reasoning**

`ys`

```
[ [] ++ ys = ys
  (x:xs) ++ ys = x : xs ++ ys ]
```

Left Unit Proof

```
mempty `mappend` ys
= {- def. of mempty -}
  [] `mappend` ys
= {- def. of mappend -}
  [] ++ ys
```

**Proof Style:
Equational
Reasoning**

`ys`

```
[ [] ++ ys = ys
  (x:xs) ++ ys = x : xs ++ ys ]
```

Left Unit Proof

```
empty `mappend` ys
= {- def. of empty -}
  [] `mappend` ys
= {- def. of mappend -}
  [] ++ ys
= {- def. of (++) -}
  ys
```

**Proof Style:
Equational
Reasoning**

```
[ [] ++ ys = ys
  (x:xs) ++ ys = x : xs ++ ys ]
```

Right Unit Proof

$$\begin{aligned} & 1 \ ++ \ [] \\ = & \\ & 1 \end{aligned}$$

$$\begin{array}{l} \boxed{\begin{array}{l} [] \quad \quad ++ \ ys \quad = \ ys \\ (x : xs) ++ ys = x : xs ++ ys \end{array}} \end{array}$$

Right Unit Proof

$$1 \mathrel{++} [] = 1$$

Proof Style:
Structural
Induction
+
Equational
Reasoning

$$\begin{array}{l} [] \mathrel{++} ys = ys \\ (x:xs) \mathrel{++} ys = x : xs \mathrel{++} ys \end{array}$$

Base Case: $1 = []$

$$= [] ++ []$$

Proof Style:
Structural
Induction
+
Equational
Reasoning

$$\begin{array}{l} [] ++ ys = ys \\ (x:xs) ++ ys = x : xs ++ ys \end{array}$$

Base Case: $1 = []$

$[] ++ []$
 $= \{- \text{def. of } (++) -\}$
 $[]$

Proof Style:
Structural
Induction
+
Equational
Reasoning

$[] ++ ys = ys$
 $(x : xs) ++ ys = x : xs ++ ys$

Inductive Case: $1 = x : xs$

$$(x : xs) ++ []$$

=

$x : xs$

Proof Style:
Structural
Induction
+
Equational
Reasoning

$$\begin{array}{l} [] ++ ys = ys \\ (x : xs) ++ ys = x : xs ++ ys \end{array}$$

Inductive Case: $1 = x : xs$

$$(x : xs) ++ []$$

=

$x : xs$

Proof Style:
Structural
Induction
+
Equational
Reasoning

Induction Hypothesis

$$xs ++ [] = xs$$

$$[] ++ ys = ys$$

$$(x : xs) ++ ys = x : xs ++ ys$$

Inductive Case: $1 = x : xs$

$$(x : xs) ++ [] \\ = \{- \text{def. of } (++) -\}$$

$x : xs$

Proof Style:
Structural
Induction
+
Equational
Reasoning

Induction Hypothesis

$$xs ++ [] = xs$$

$$[] ++ ys = ys$$

$$(x : xs) ++ ys = x : xs ++ ys$$

Inductive Case: $1 = x : xs$

$(x : xs) ++ []$
 $= \{- \text{def. of } (++) -\}$
 $x : xs ++ []$

$x : xs$

Proof Style:
Structural
Induction
+
Equational
Reasoning

Induction Hypothesis

$xs ++ [] = xs$

$[] ++ ys = ys$

$(x : xs) ++ ys = x : xs ++ ys$

Inductive Case: $1 = x : xs$

$(x : xs) ++ []$
= { - def. of $(++)$ - }
 $x : xs ++ []$
= { - ind. hypot. - }
 $x : xs$

Proof Style:
Structural
Induction
+
Equational
Reasoning

Induction Hypothesis

$xs ++ [] = xs$

$[] ++ ys = ys$

$(x : xs) ++ ys = x : xs ++ ys$

Associativity Proof

$$\begin{aligned} &xs \ ++ \ (ys \ ++ \ zs) \\ = & \\ &(xs \ ++ \ ys) \ ++ \ zs \end{aligned}$$

Proof Style:
Structural
Induction
+
Equational
Reasoning

$$\begin{array}{l} [\quad [] \quad ++ \ ys \quad = \quad ys \\ [\quad (x : xs) \quad ++ \ ys \quad = \quad x : xs \ ++ \ ys \end{array}$$

Associativity Proof

$$\begin{aligned} &xs \ ++ \ (ys \ ++ \ zs) \\ = & \\ &(xs \ ++ \ ys) \ ++ \ zs \end{aligned}$$

Homework



Proof Style:
Structural
Induction
+
Equational
Reasoning

$$\begin{array}{l} [\quad [] \quad ++ \ ys \quad = \quad ys \\ [\quad (x : xs) \quad ++ \ ys \quad = \quad x : xs \ ++ \ ys \end{array}$$



The Free Monoid

Monoid (Homo)morphism

a function between monoids

$$f :: M1 \rightarrow M2$$

such that:

$$f \text{ mempty} = \text{mempty}$$

and:

$$f (x <> y) = f x <> f y$$

Monoid (Homo)morphism

a function between monoids

`length :: [a] -> Int`

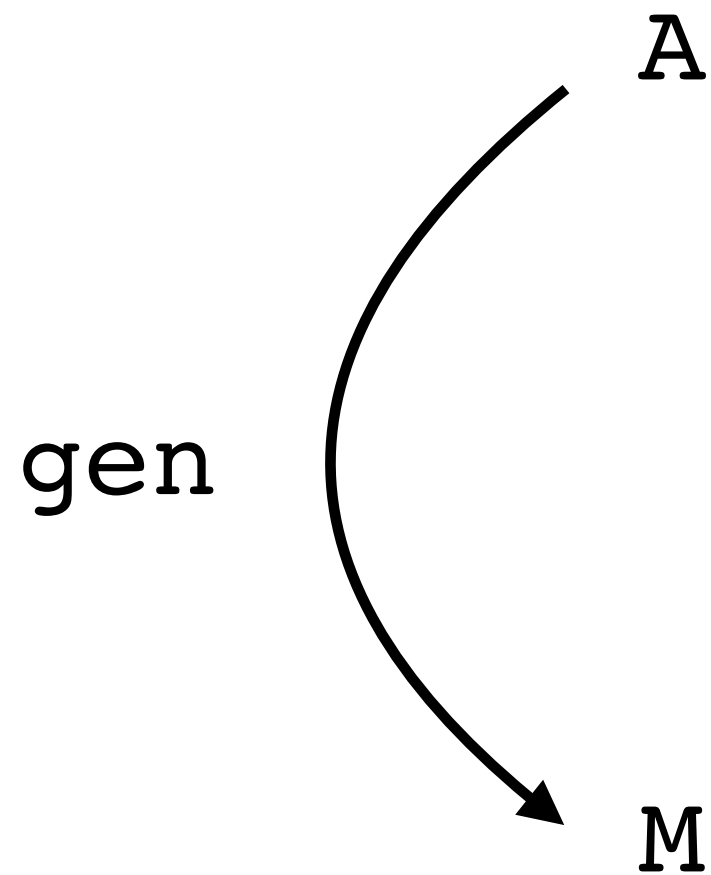
such that:

`length [] = 0`

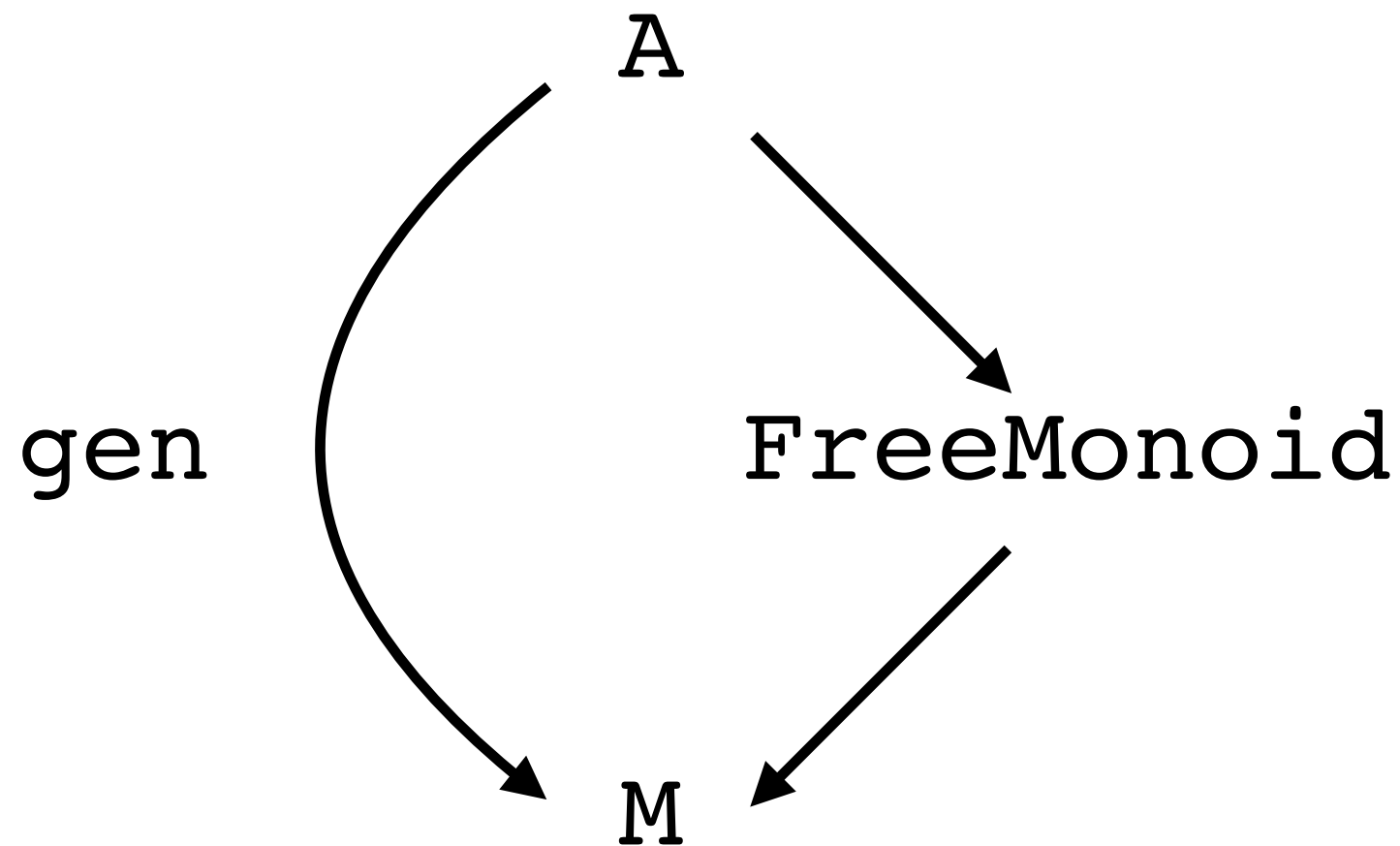
and:

`length (x ++ y) = length x + length y`

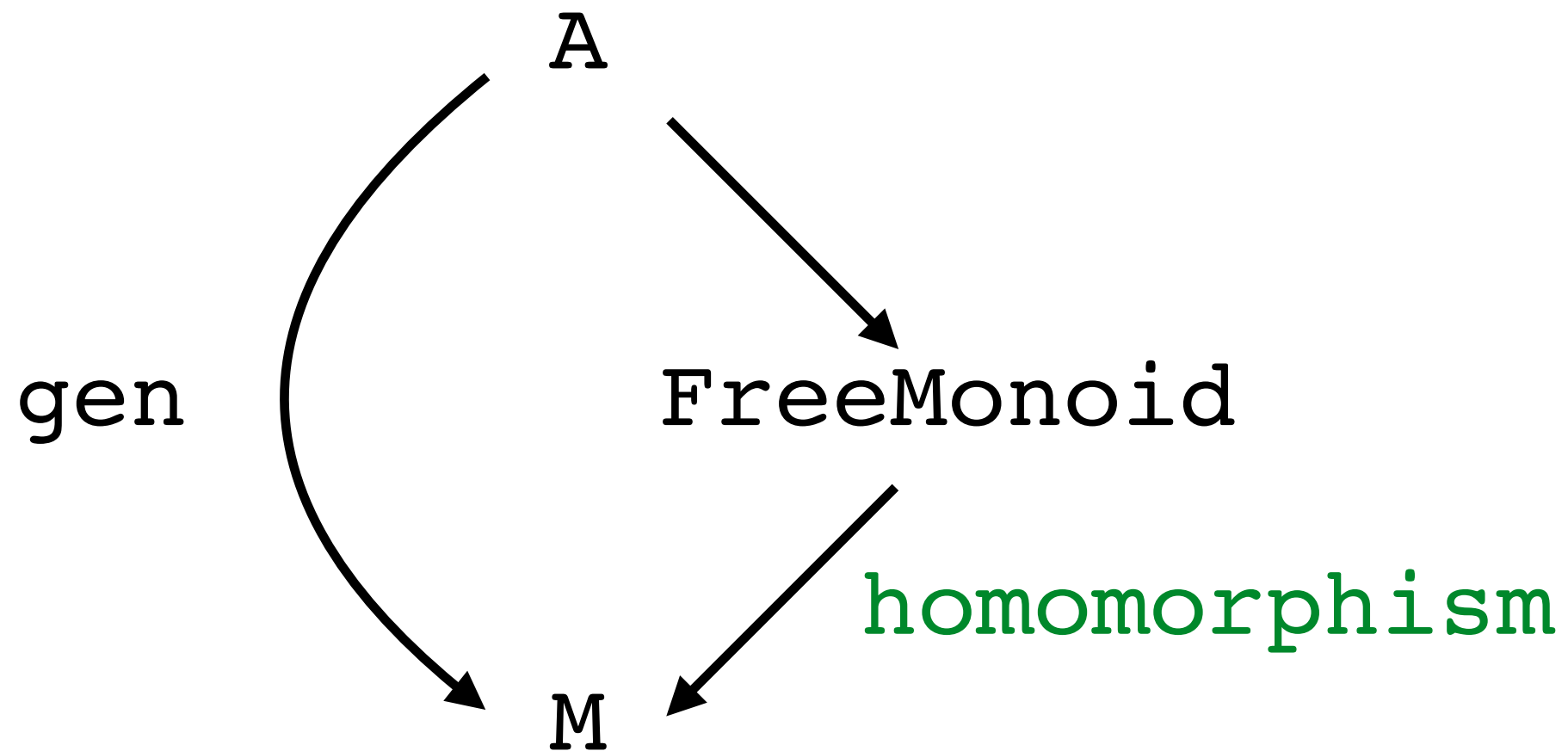
Free Monoid



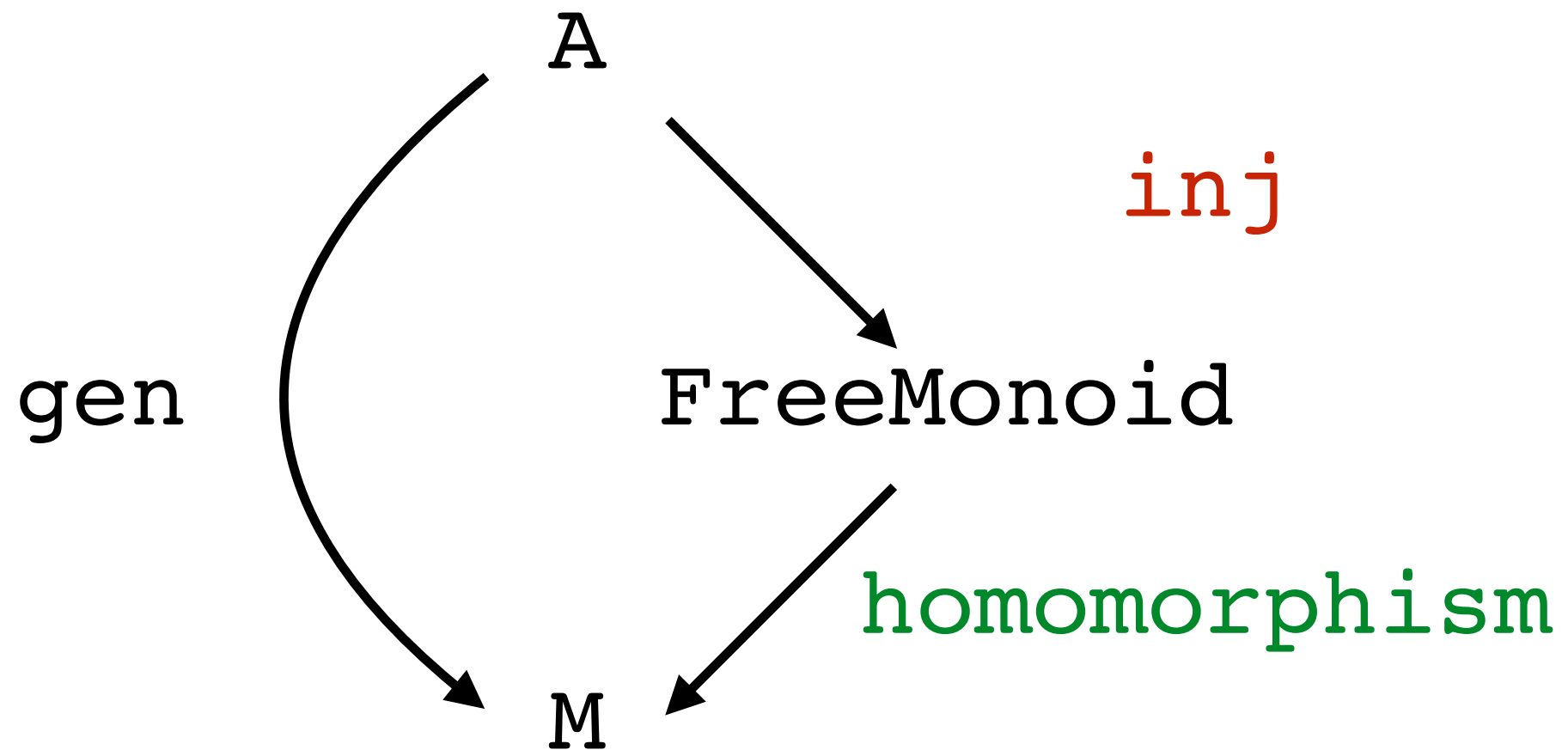
Free Monoid



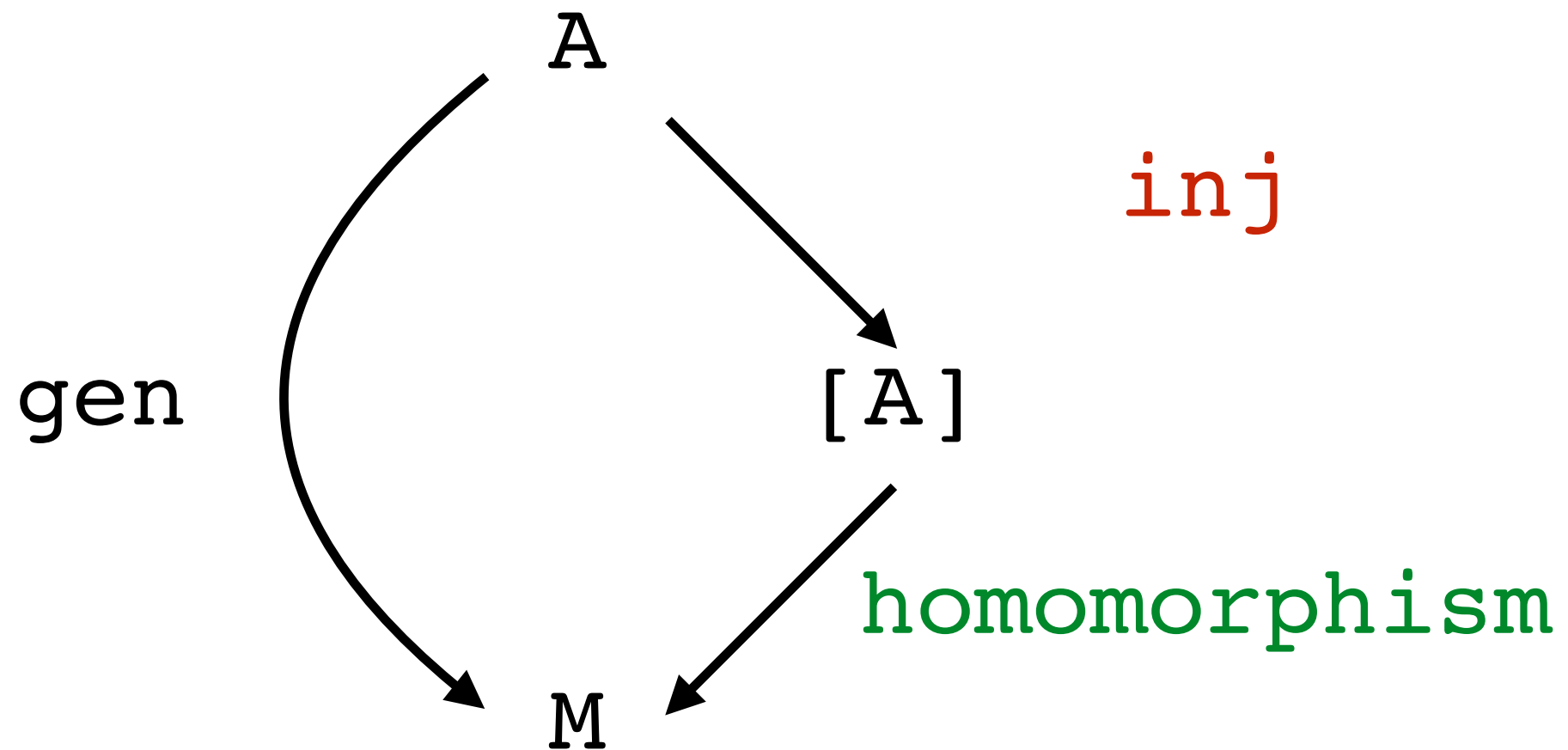
Free Monoid



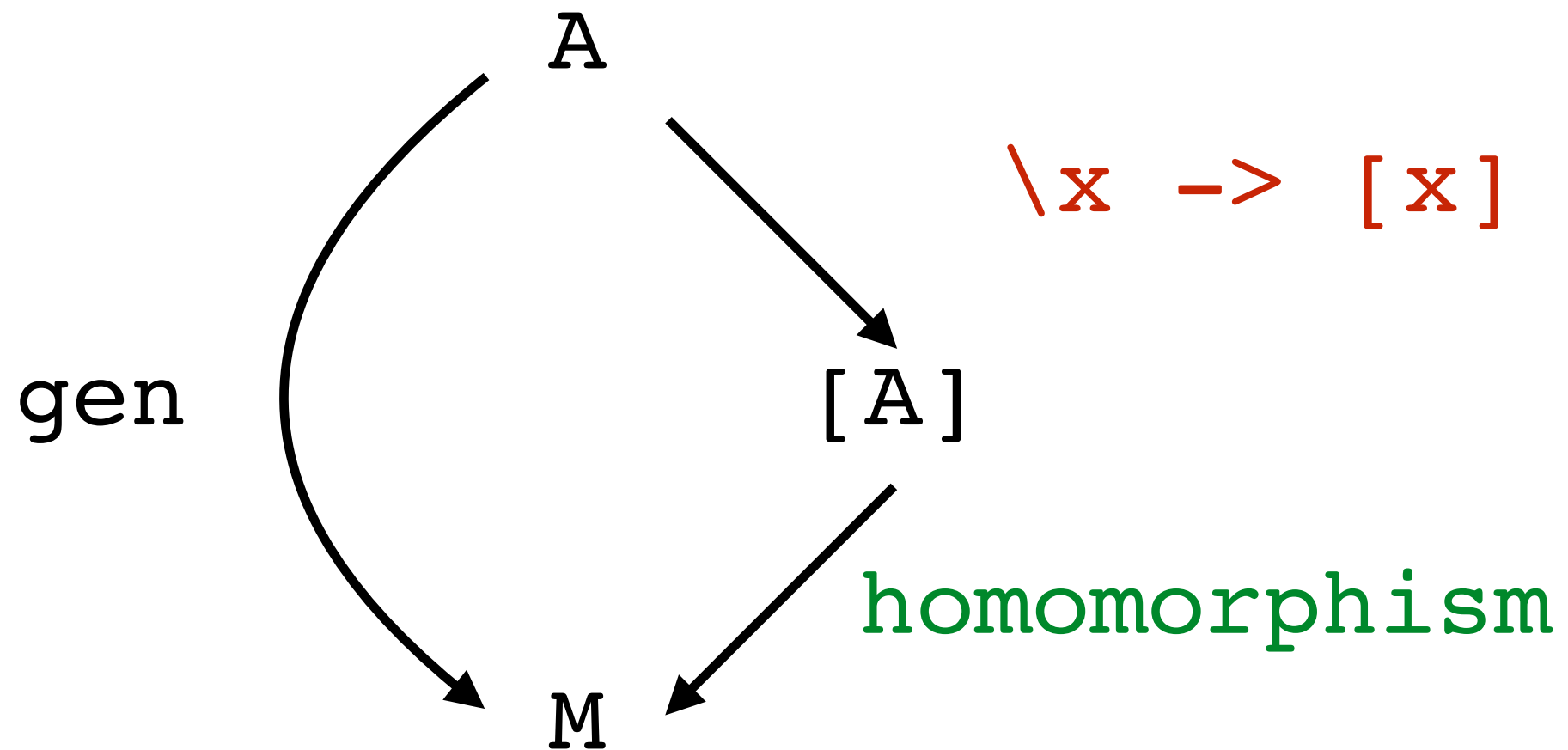
Free Monoid



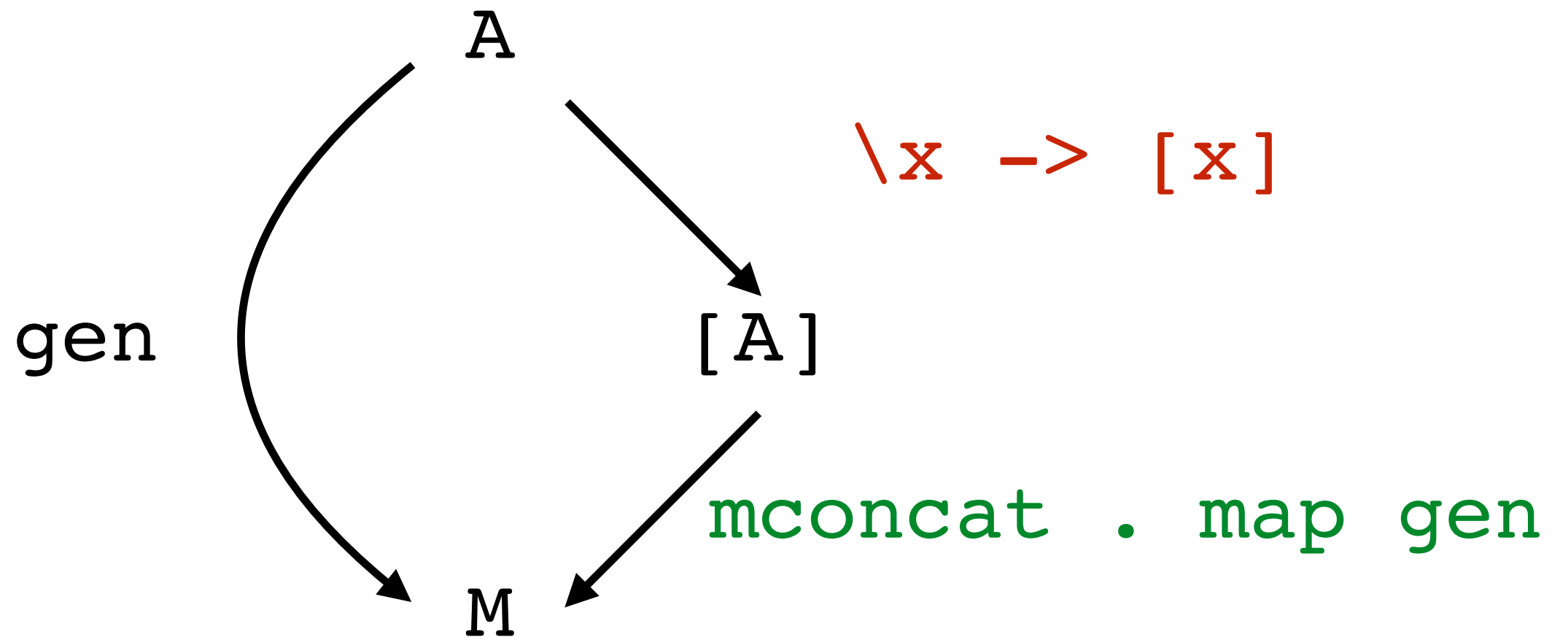
Free Monoid



Free Monoid



Free Monoid



What is a Data.Foldable?

T a

What is a Data.Foldable?

T a

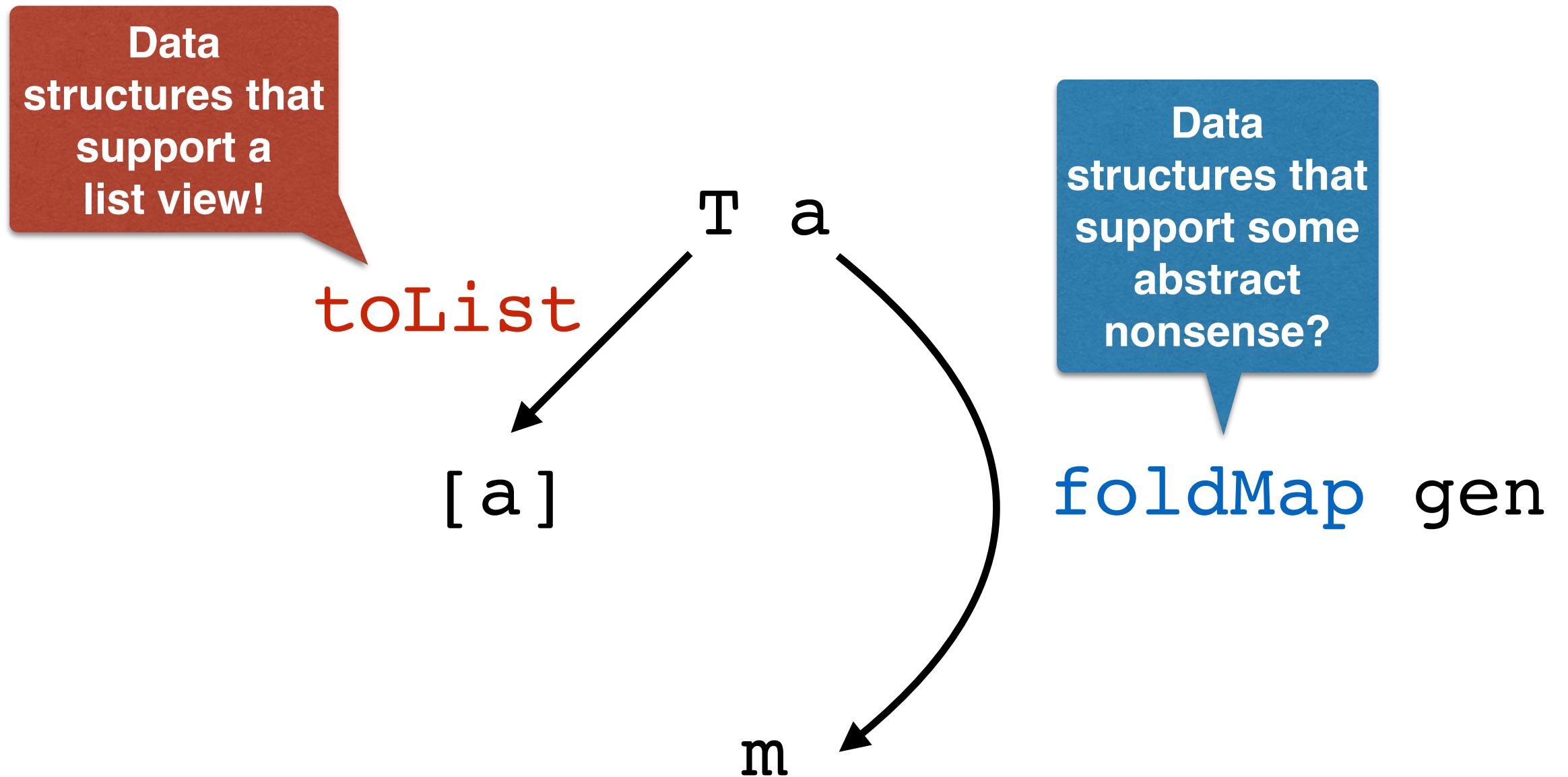
Data
structures that
support some
abstract
nonsense?

foldMap gen

m

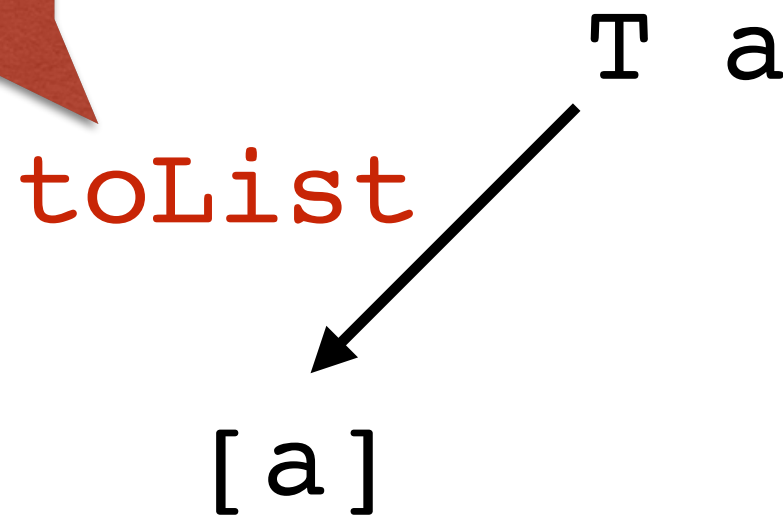


What is a Data.Foldable?



What is a Data.Foldable?

Data
structures that
support a
list view!



`mconcat . map gen`

```
graph TD; list["[a]"] -- "mconcat . map gen" --> m["m"]
```

The diagram shows a list `[a]` at the top, with an arrow pointing down to a monad `m`. The arrow is labeled `mconcat . map gen` in green text.

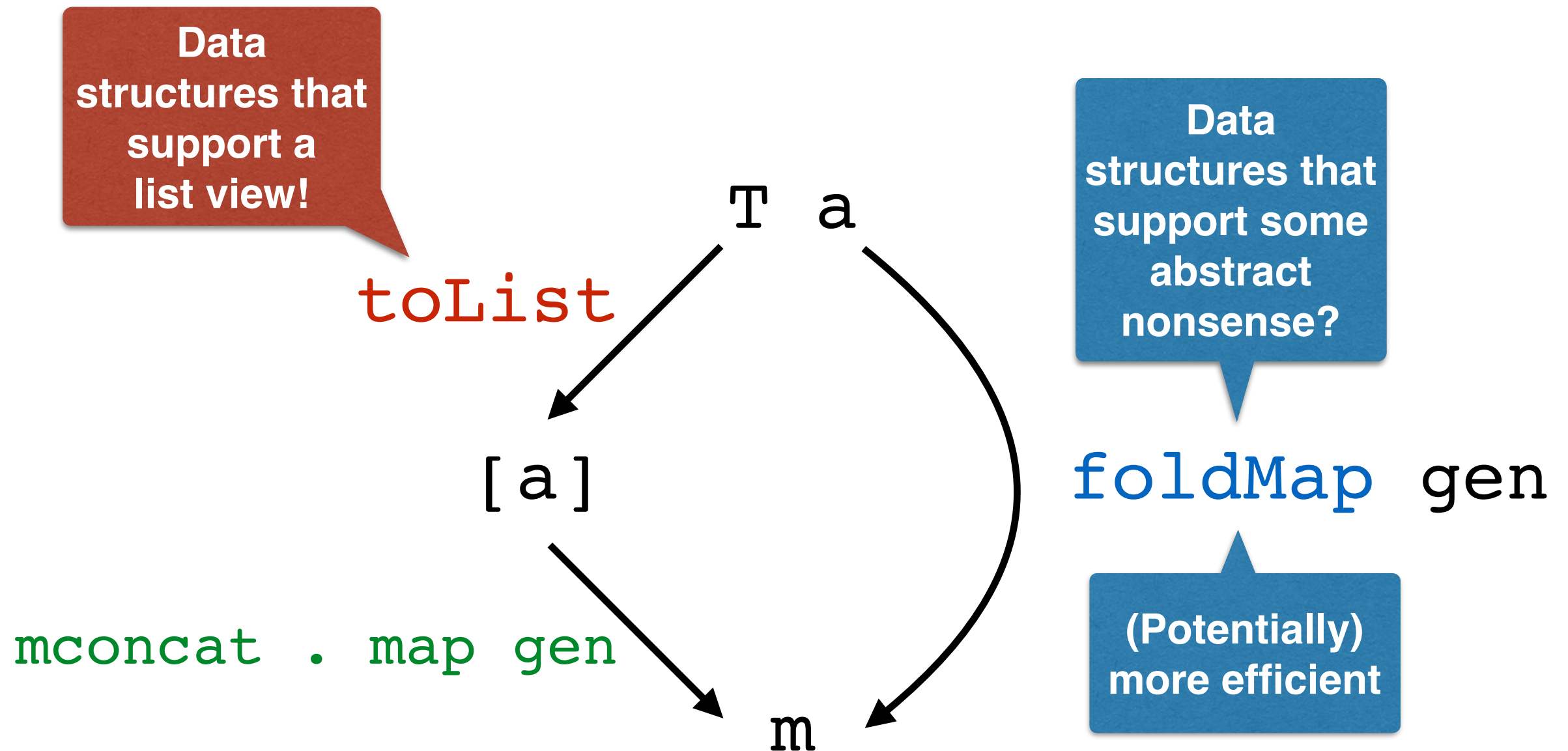
Data
structures that
support some
abstract
nonsense?

`foldMap` `gen`

```
graph TD; a -- "foldMap gen" --> m["m"]
```

The diagram shows a type `a` at the top, with a curved arrow pointing down to a monad `m`. The arrow is labeled `foldMap gen` in blue text.

What is a Data.Foldable?



A large, stylized 'X' logo composed of two overlapping, slightly offset rectangular shapes, creating a three-dimensional effect. The logo is a dark purple color, matching the background.

Summary

Monoids

- ★ Simple concept from Algebra
- ★ Ubiquitous in Haskell
- ★ Cool Applications
- ★ List is the Free Monoid

A large, semi-transparent, stylized 'X' watermark is centered in the background of the slide. It is composed of two overlapping diagonal bars.

Next time: 5/5/2015



Data
Genericity



Recursion
Schemes

GADTs

DSLs



Expression
Problem

Monads

Type
Families

Type
Classes



Lists
and
other
Monoids

Effect
Handlers

Free
Theorems

...



Data
Genericity



Recursion
Schemes

GADTs

DSLs



Expression
Problem

Monads

Type
Families

Type
Classes



Lists
and
other
Monoids

Effect
Handlers

Free
Theorems

...

A large, stylized 'X' logo composed of two overlapping chevron shapes, one pointing up-right and the other down-right, in a dark purple color. It is centered in the background of the slide.

Join the Google Group
Leuven Haskell User Group