Lists and Other Monoids

Tom Schrijvers

Leuven Haskell
User Group

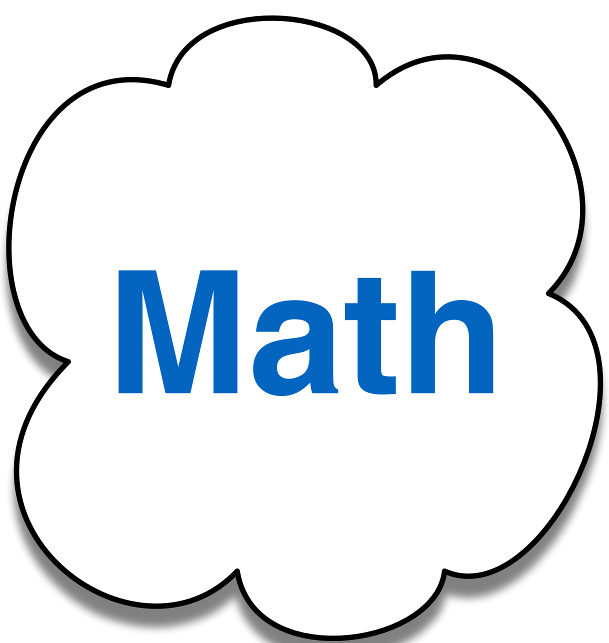
Data Genericity	Recursion Schemes	GADTs	DSLs
Expression Problem	Monads	Type Families	Type Classes
Lists and Other Monoids	Effect Handlers	Free Theorems	…

Data Genericity	Recursion Schemes	GADTs	DSLs
Expression Problem	Monads	Type Families	Type Classes
Lists and Other Monoids	Effect Handlers	Free Theorems	...

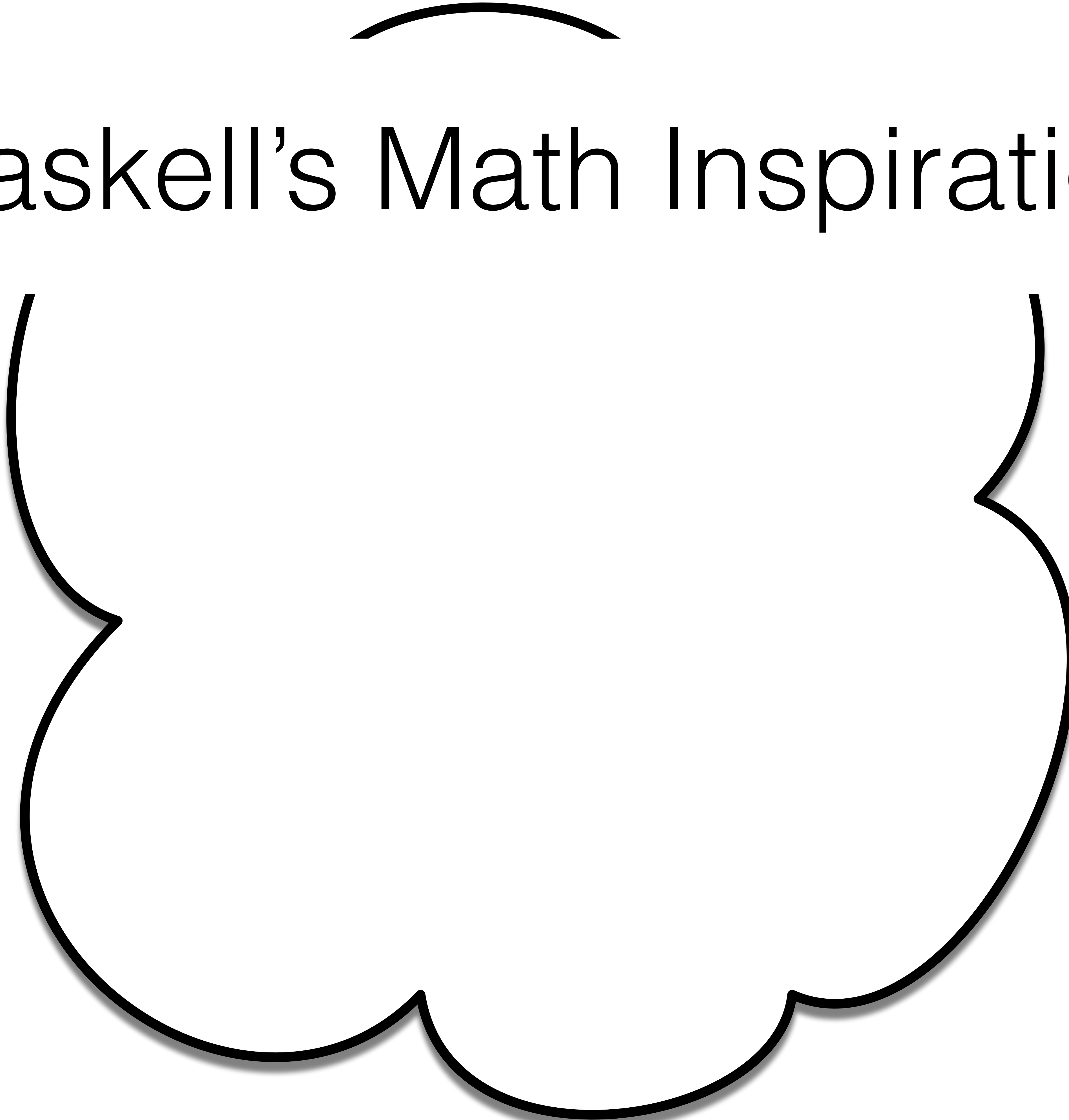
Introduction to Monoids

Abstract Patterns

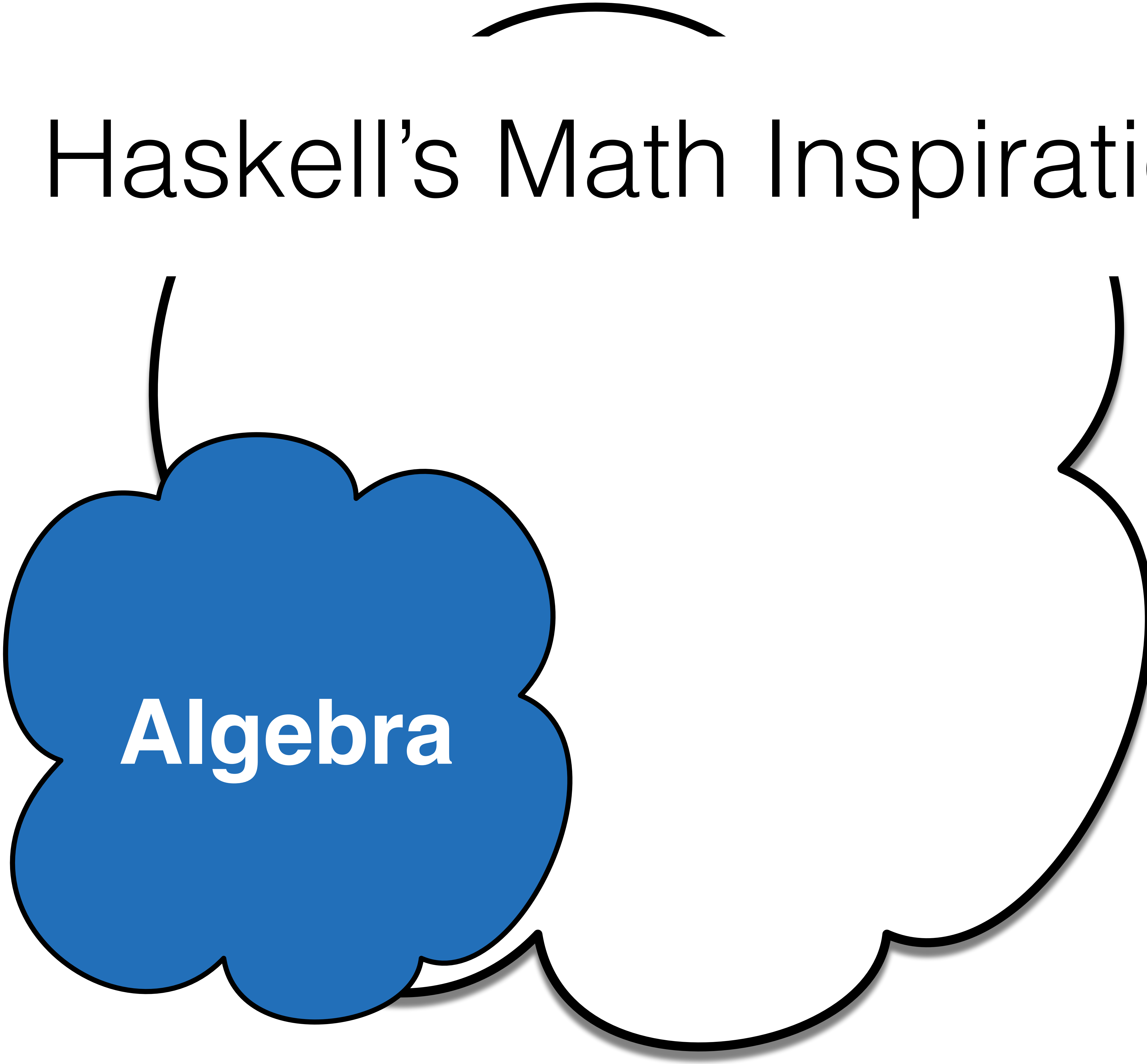
Abstract Patterns



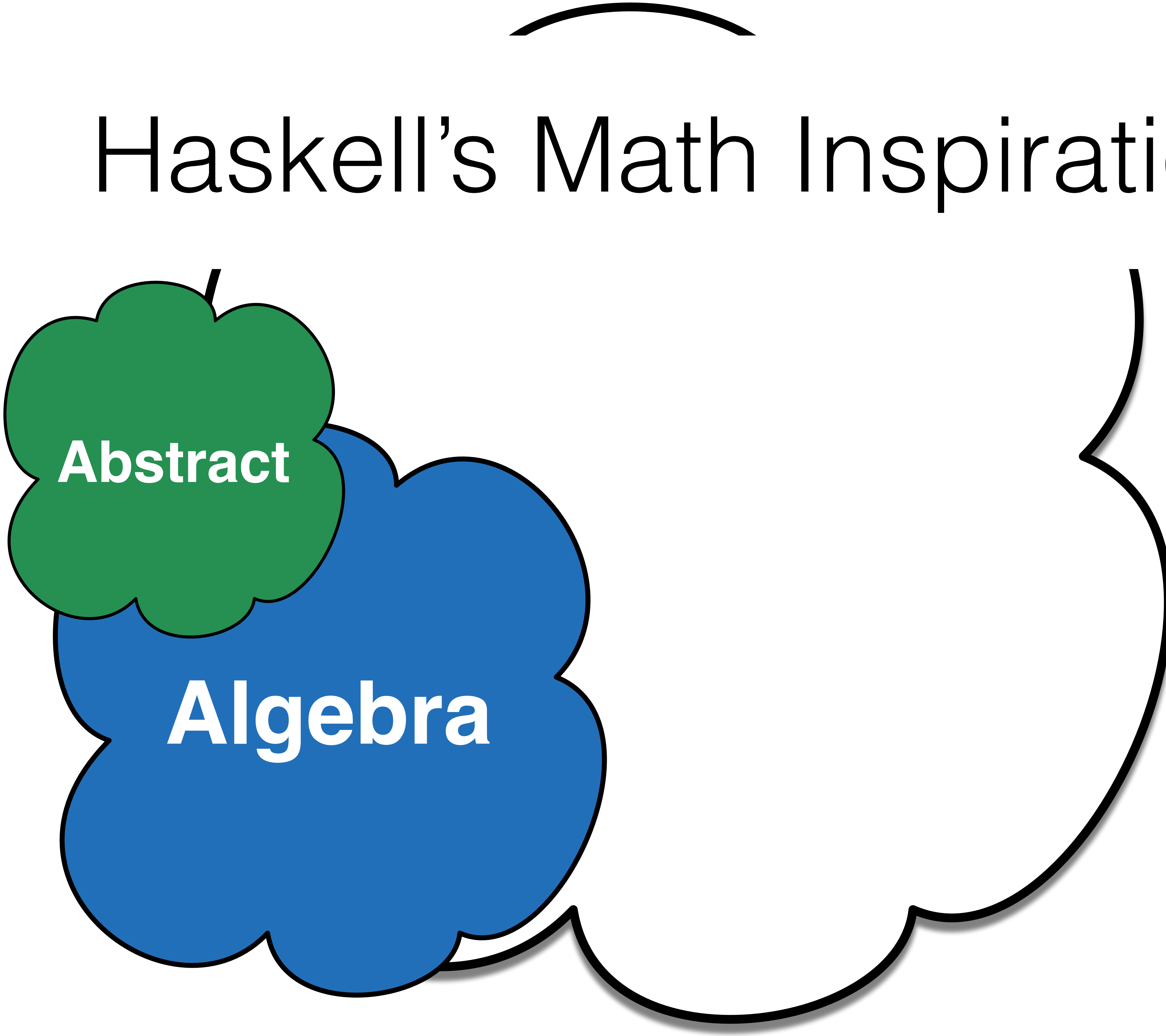
Haskell’s Math Inspiration



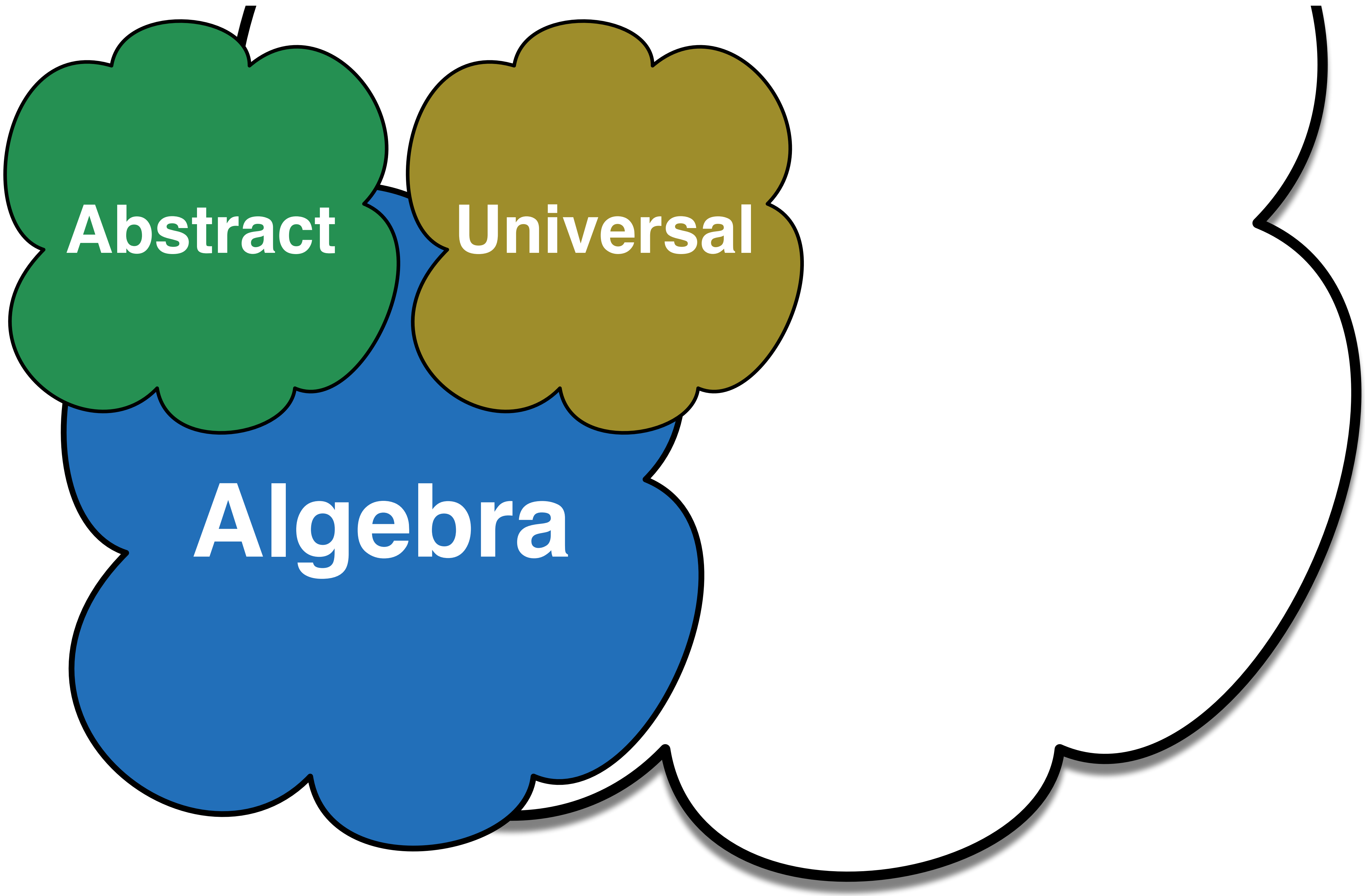
Haskell's Math Inspiration



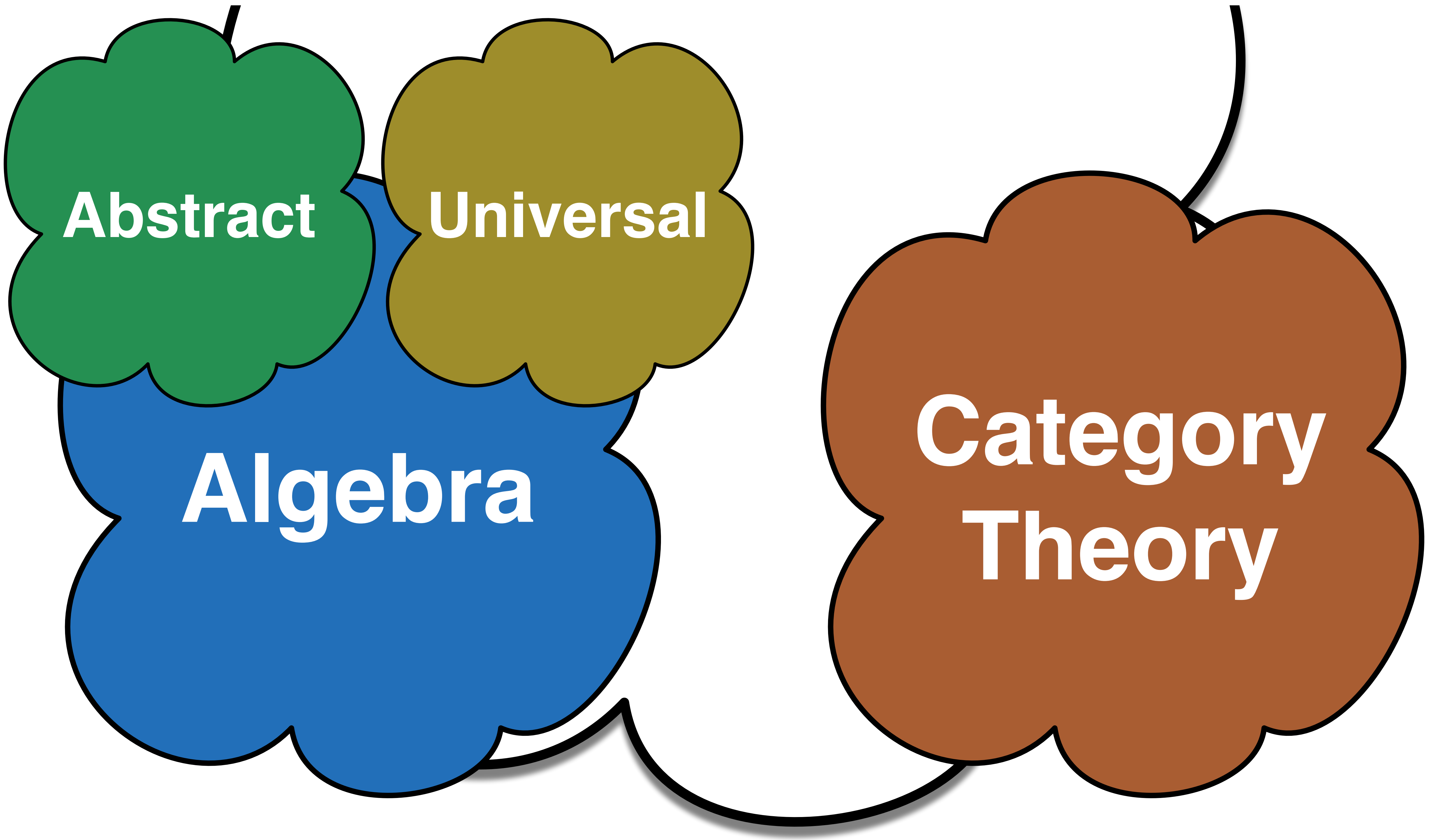
Haskell’s Math Inspiration

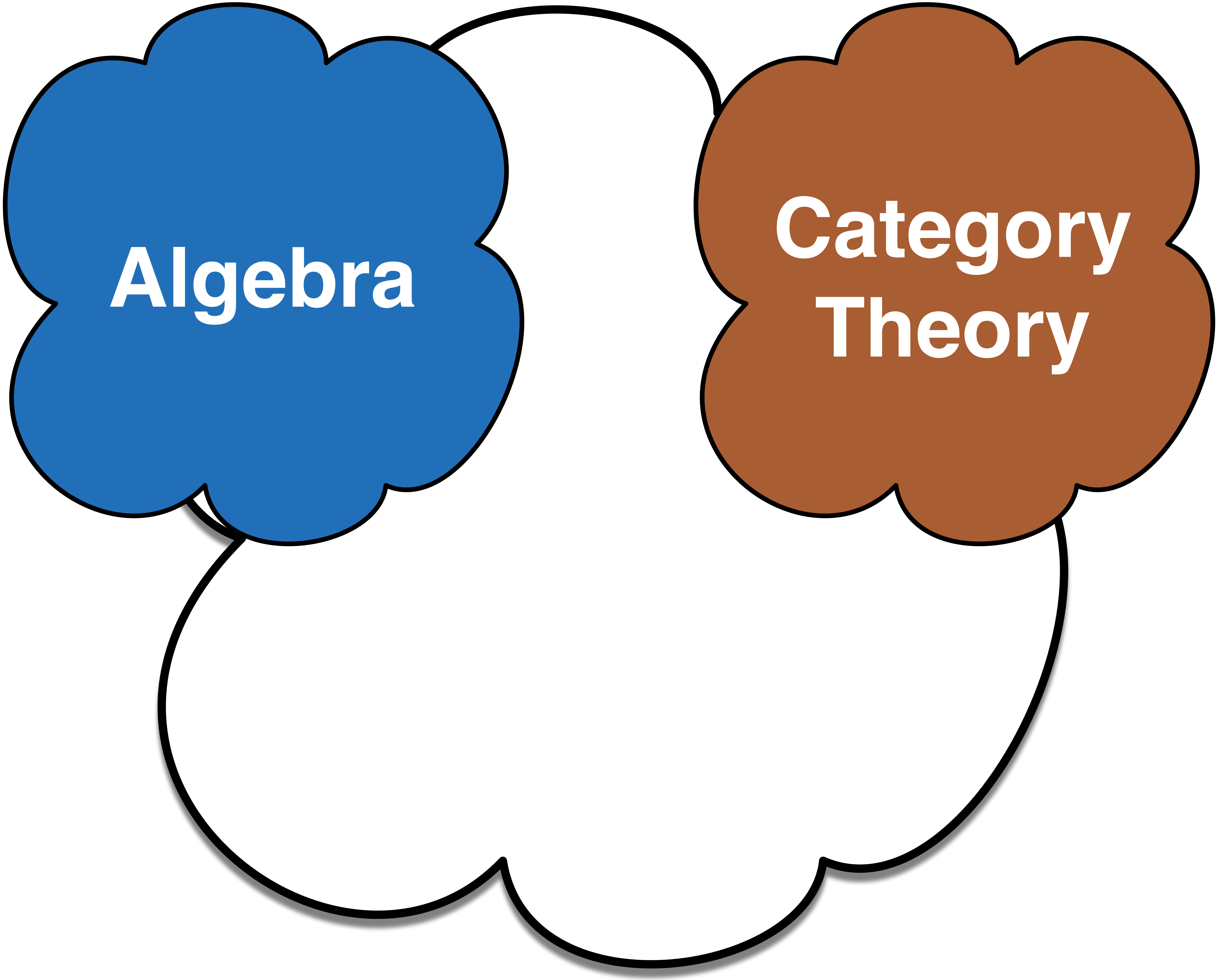


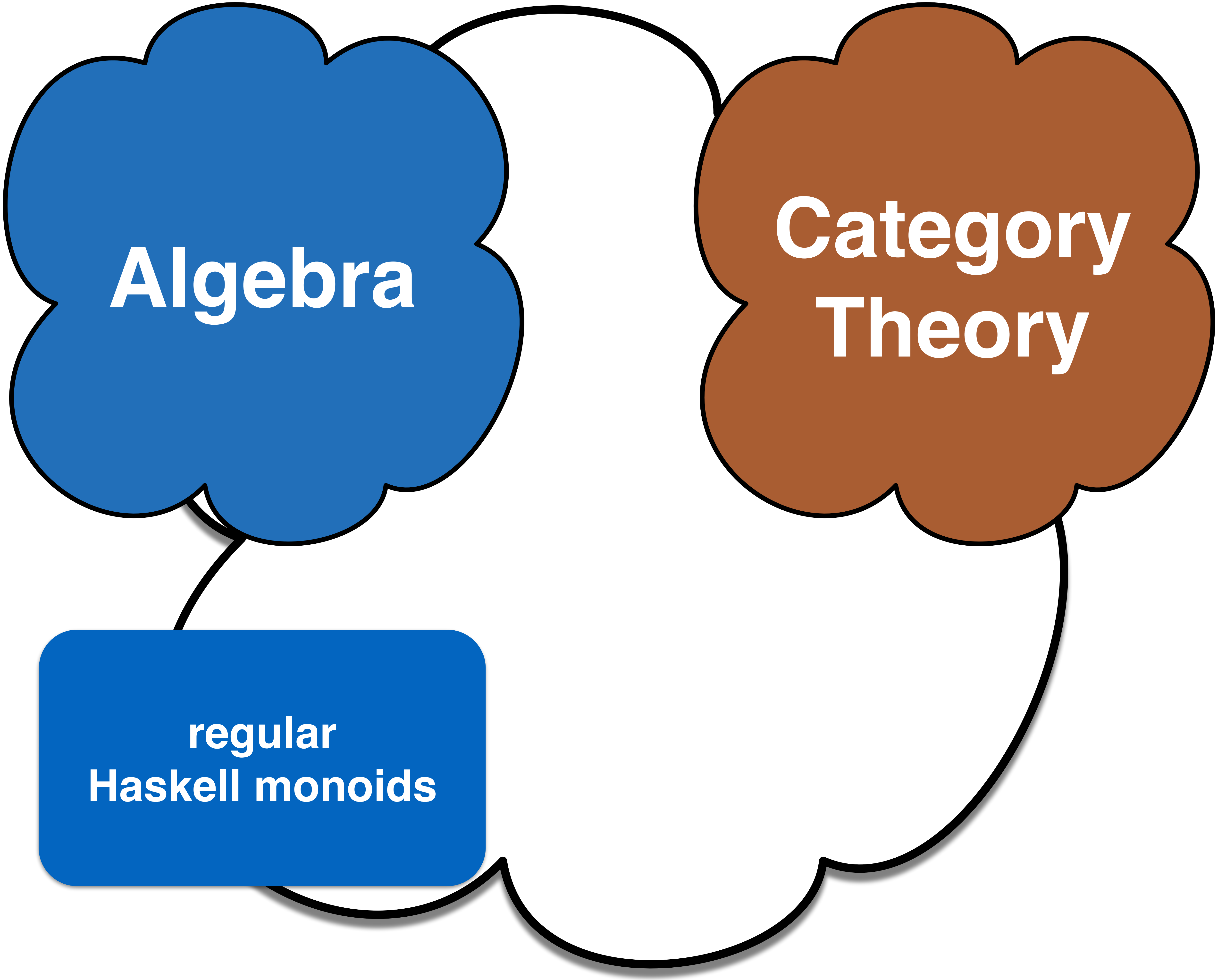
Haskell's Math Inspiration

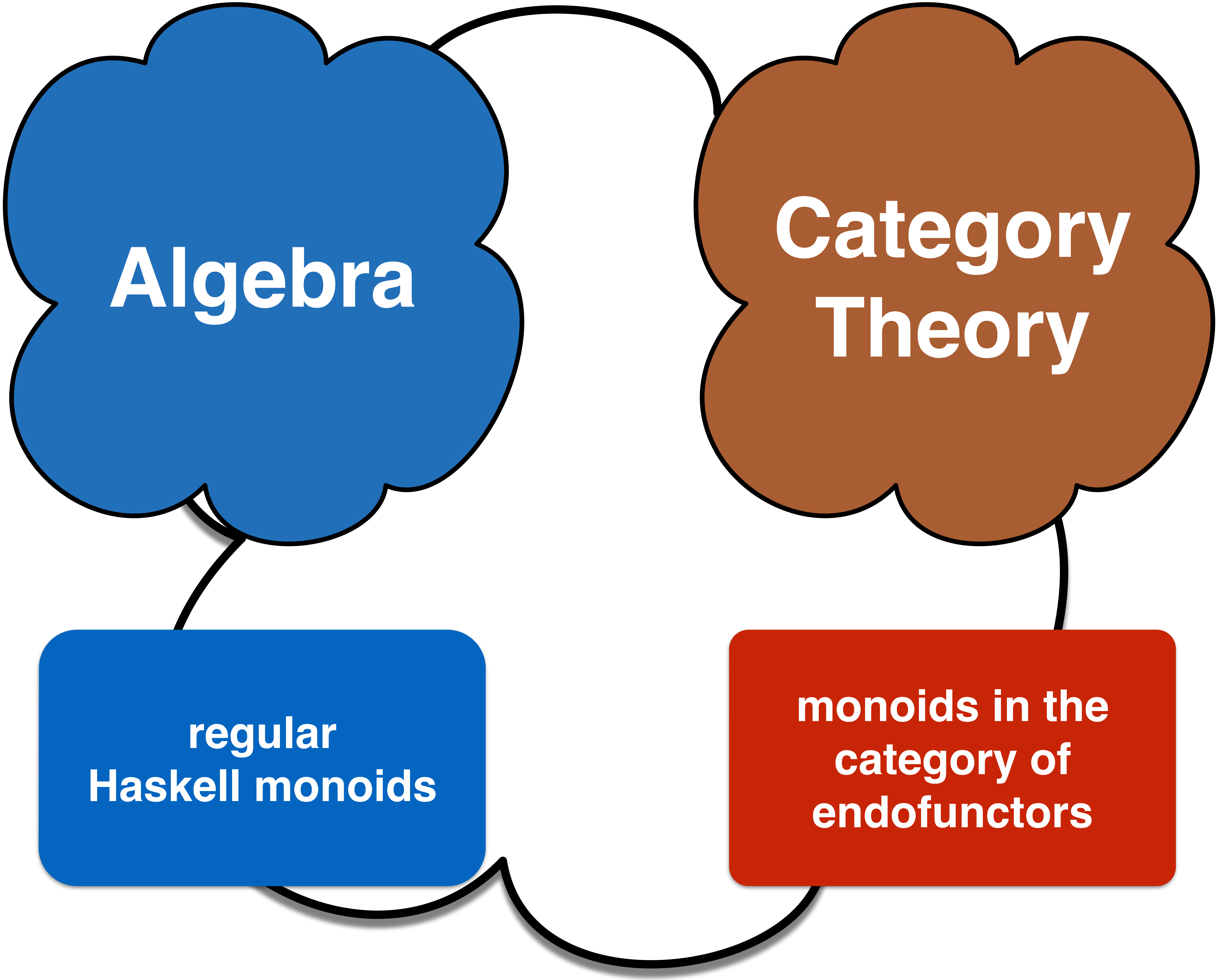


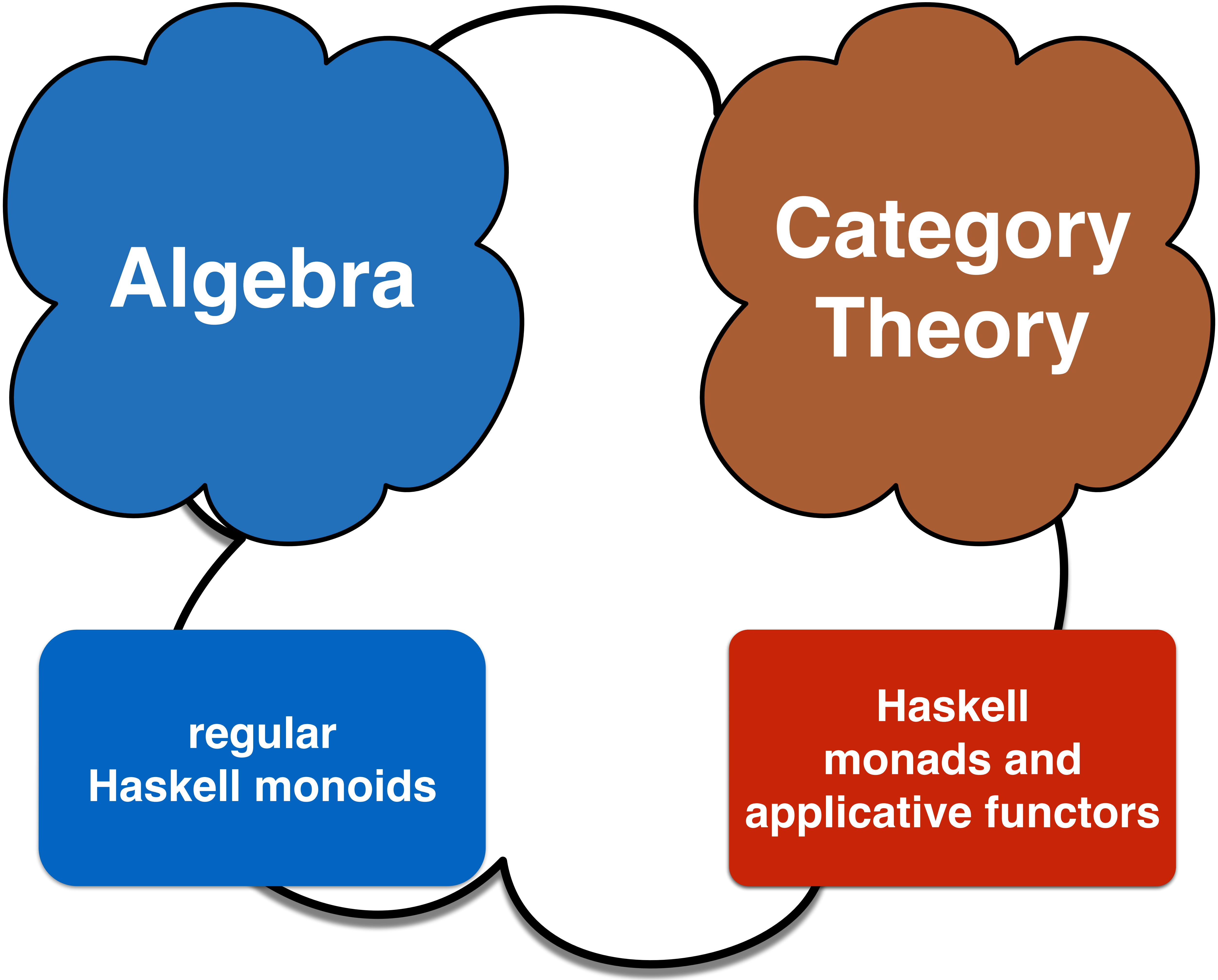
Haskell’s Math Inspiration



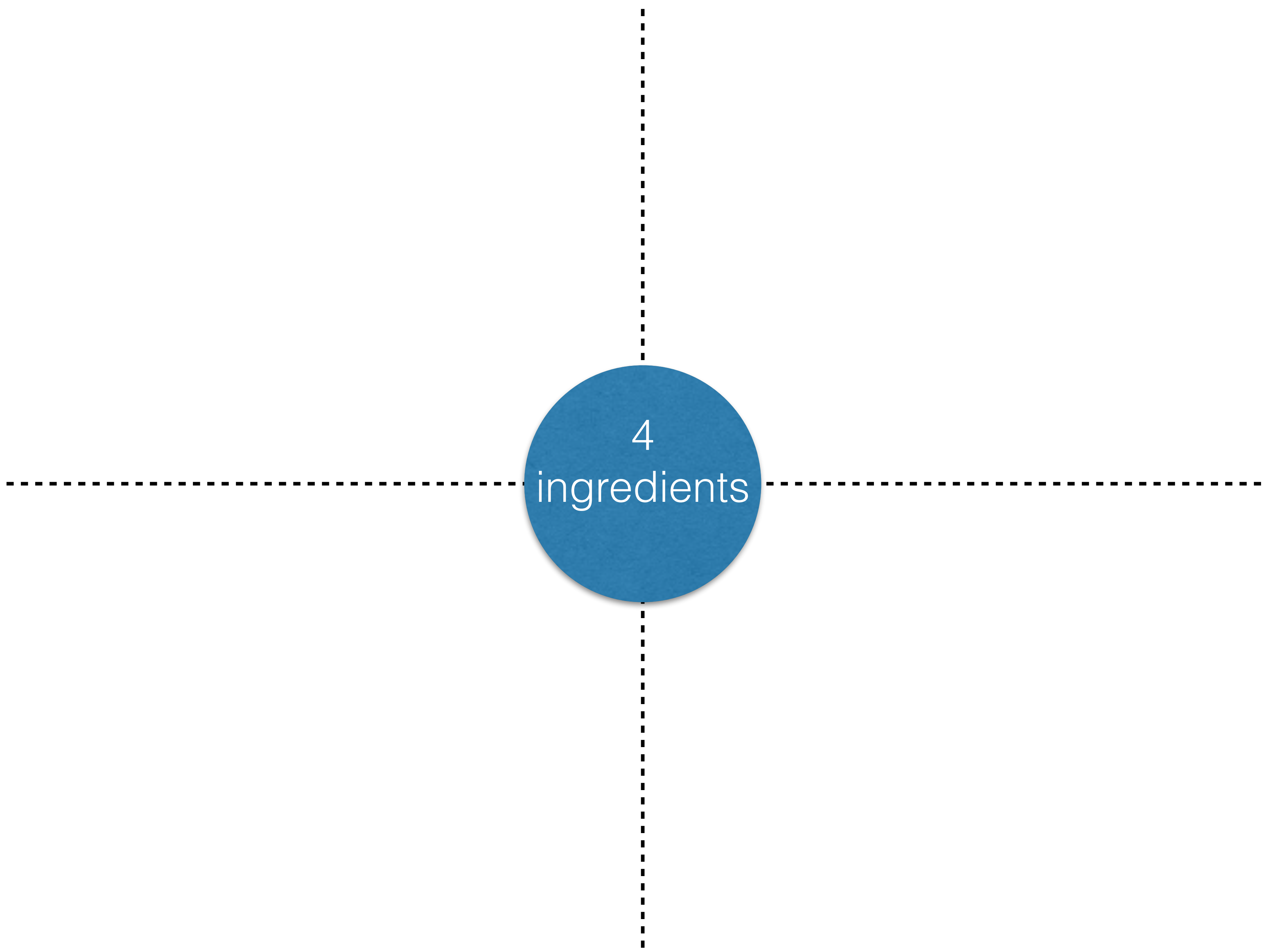

Algebra
Category Theory


Algebra
Category Theory
regular
Haskell monoids

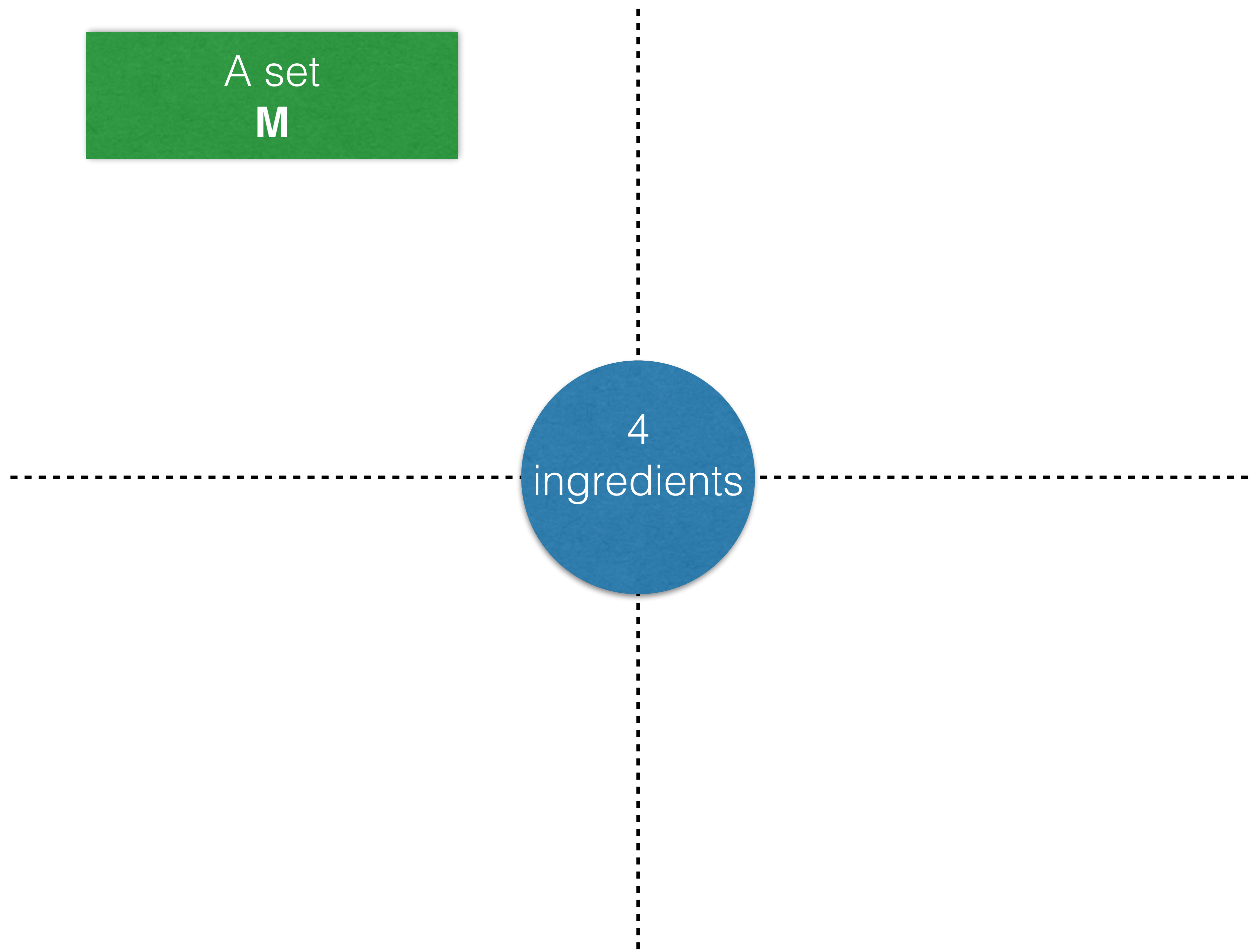

Algebra
Category Theory
regular Haskell monoids
monoids in the category of endofunctors


Algebra
Category Theory
regular
Haskell monoids
Haskell
monads and
applicative functors

The Monoid Structure



The Monoid Structure



The Monoid Structure

A set
M

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \rightarrow \mathbf{M}$

4
ingredients

The Monoid Structure

A set
M

A binary operator
$\otimes$: **M × M → M**

4
ingredients

An element
$\boldsymbol{\varepsilon} \in \mathbf{M}$

The Monoid Structure

A set
M

A binary operator
$\otimes$: **M** × **M** → **M**

4
ingredients

An element
ε ∈ **M**

3 Properties

The Monoid Structure

A set
M

A binary operator
$\otimes$: **M × M → M**

$\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$

4
ingredients

An element
ε ∈ **M**

3 Properties

The Monoid Structure

A set
M

A binary operator
$\otimes$ **: M × M → M**

$\mathbb{Z} = \{\ldots,-1,0,1,\ldots\}$

$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

4
ingredients

An element
ε $\in$ **M**

3 Properties

The Monoid Structure

A set
M

A binary operator
$\otimes$ **: M × M → M**

$\mathbb{Z} = \{\ldots,-1,0,1,\ldots\}$

$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

4
ingredients

$0 \in \mathbb{Z}$

An element
ε $\in$ **M**

3 Properties

The Monoid Structure

A set
M

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$

$\mathbb{Z} = \{\ldots,-1,0,1,\ldots\}$

$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

4
ingredients

Left Unit $\boldsymbol{\varepsilon} \otimes \mathbf{x} = \mathbf{x}$

$0 \in \mathbb{Z}$

An element
$\boldsymbol{\varepsilon} \in \mathbf{M}$

3 Properties

The Monoid Structure

A set
M

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$

$\mathbb{Z} = \{\ldots,-1,0,1,\ldots\}$

$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

4
ingredients

Left Unit $\boldsymbol{\varepsilon} \otimes \mathbf{x} = \mathbf{x}$

Right Unit $\mathbf{x} \otimes \boldsymbol{\varepsilon} = \mathbf{x}$

$0 \in \mathbb{Z}$

An element
$\boldsymbol{\varepsilon} \in \mathbf{M}$

3 Properties

The Monoid Structure

A set
M

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \rightarrow \mathbf{M}$

$\mathbb{Z} = \{\ldots,-1,0,1,\ldots\}$

$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

4
ingredients

Left Unit	$\varepsilon \otimes x = x$
Right Unit	$x \otimes \varepsilon = x$
Associativity	$x \otimes (y \otimes z) = (x \otimes y) \otimes z$

$0 \in \mathbb{Z}$

An element
$\varepsilon \in \mathbf{M}$

3 Properties

Haskell Monoids

A set
$\mathbf{M}$

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$

4
ingredients

Left Unit $\varepsilon \otimes x = x$

Right Unit $x \otimes \varepsilon = x$

Associativity $x \otimes (y \otimes z) = (x \otimes y) \otimes z$

An element
$\varepsilon \in \mathbf{M}$

3 Properties

Haskell Monoids

A set
M

A type
M

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \rightarrow \mathbf{M}$

4
ingredients

Left Unit $\varepsilon \otimes x = x$

Right Unit $x \otimes \varepsilon = x$

Associativity $x \otimes (y \otimes z) = (x \otimes y) \otimes z$

An element
$\varepsilon \in \mathbf{M}$

3 Properties

Haskell Monoids

A set
M

A type
M

`Int`

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$

4
ingredients

Left Unit	$\varepsilon \otimes x$	$=$	x
Right Unit	$x \otimes \varepsilon$	$=$	x
Associativity	$x \otimes (y \otimes z)$	$=$	$(x \otimes y) \otimes z$

An element
$\varepsilon \in \mathbf{M}$

3 Properties

Haskell Monoids

A set
M

A type
M

`Int`

A binary operator
$\otimes : M \times M \to M$

A function
$\otimes :: M \to M \to M$

4
ingredients

An element
$\varepsilon \in M$

Left Unit	$\varepsilon \otimes x$	=	x
Right Unit	$x \otimes \varepsilon$	=	x
Associativity	$x \otimes (y \otimes z)$	=	$(x \otimes y) \otimes z$

3 Properties

Haskell Monoids

A set
M

A type
M

`Int`

A binary operator
$\otimes : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$

A function
$\otimes :: \mathbf{M} \to \mathbf{M} \to \mathbf{M}$

`(+) :: Int -> Int -> Int`

4
ingredients

An element
$\boldsymbol{\varepsilon} \in \mathbf{M}$

Left Unit $\quad \varepsilon \otimes x = x$

Right Unit $\quad x \otimes \varepsilon = x$

Associativity $\quad x \otimes (y \otimes z) = (x \otimes y) \otimes z$

3 Properties

Haskell Monoids

A set
M

A type
M

`Int`

4
ingredients

An element
ε :: M

An element
ε ∈ M

A binary operator
$\otimes$ **: M × M → M**

A function
$\otimes$ **:: M → M → M**

`(+) :: Int -> Int -> Int`

Left Unit	$\varepsilon \otimes x$	=	x
Right Unit	$x \otimes \varepsilon$	=	x
Associativity	$x \otimes (y \otimes z)$	=	$(x \otimes y) \otimes z$

3 Properties

Haskell Monoids

A set
M

A type
M

`Int`

A binary operator
$\otimes : M \times M \to M$

A function
$\otimes :: M \to M \to M$

`(+) :: Int -> Int -> Int`

4 ingredients

`0 :: Int`

An element
$\varepsilon :: M$

An element
$\varepsilon \in M$

Left Unit	$\varepsilon \otimes x$	=	x
Right Unit	$x \otimes \varepsilon$	=	x
Associativity	$x \otimes (y \otimes z)$	=	$(x \otimes y) \otimes z$

3 Properties

Haskell Monoids

A set
M

A type
M

`Int`

A binary operator
$\otimes : M \times M \to M$

A function
$\otimes :: M \to M \to M$

`(+) :: Int -> Int -> Int`

4
ingredients

`0 :: Int`

An element
$\varepsilon :: M$

An element
$\varepsilon \in M$

Left Unit	$\varepsilon \otimes x$	=	x
Right Unit	$x \otimes \varepsilon$	=	x
Associativity	$x \otimes (y \otimes z)$	=	$(x \otimes y) \otimes z$

3 Properties

Data.Monoid

Monoid Type Class

Data.Monoid

Monoid Type Class

A type
M

Data.Monoid

Monoid Type Class

A type
M

```
class Monoid m where
```

Data.Monoid

Monoid Type Class

```
class Monoid m where
```

A type
M

An element
ε :: M

Data.Monoid

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
⊗ :: M → M → M

Monoid Type Class

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

A type
M

An element
ε :: M

A function
$\otimes$ **:: M → M → M**

Data.Monoid

Monoid Type Class

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

A type
M

An element
ε :: M

A function
⊗ :: M → M → M

3 Properties

Data.Monoid

Monoid Type Class

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

A type
M

An element
ε :: M

A function
⊗ :: M → M → M

3 Properties

not in the Haskell language 😕

Data.Monoid

Monoid Type Class

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

A type
M

An element
ε :: M

A function
⊗ :: M → M → M

3 Properties

```
(<>) = mappend
```

convenient binary operator

Monoid Instance

```haskell
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

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

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

Two Possible Instances

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

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

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

Data.Monoid

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

Data.Monoid

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 }
```

Data.Monoid

Newtype to the Rescue!

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

Both isomorphic to `Bool`

```haskell
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 }
```

Data.Monoid

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)
```

Data.Monoid

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

```
Sum a
Num a => a
Product a
```

Data.Monoid

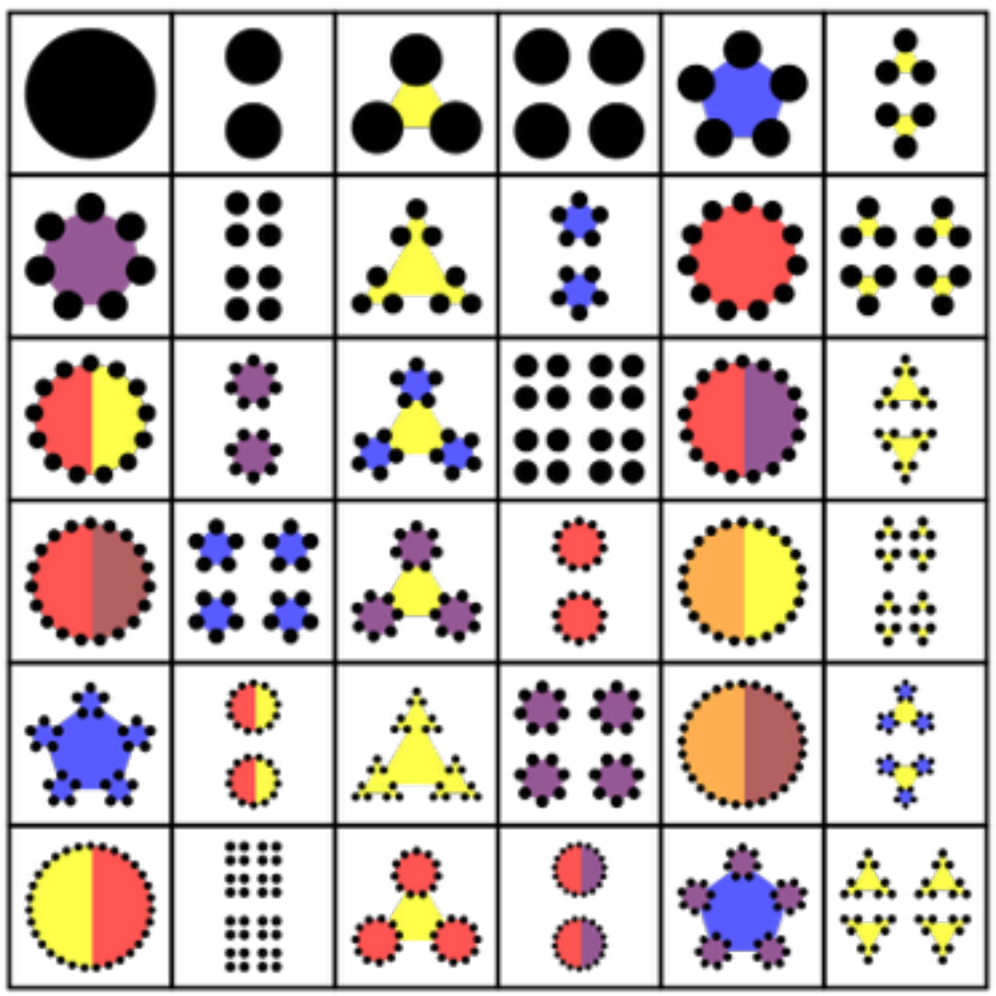
Same Problem for Num

```
                 Sum a
Num a => a
                 Product a
```

Homework
Invent 10 more monoid structures for Int

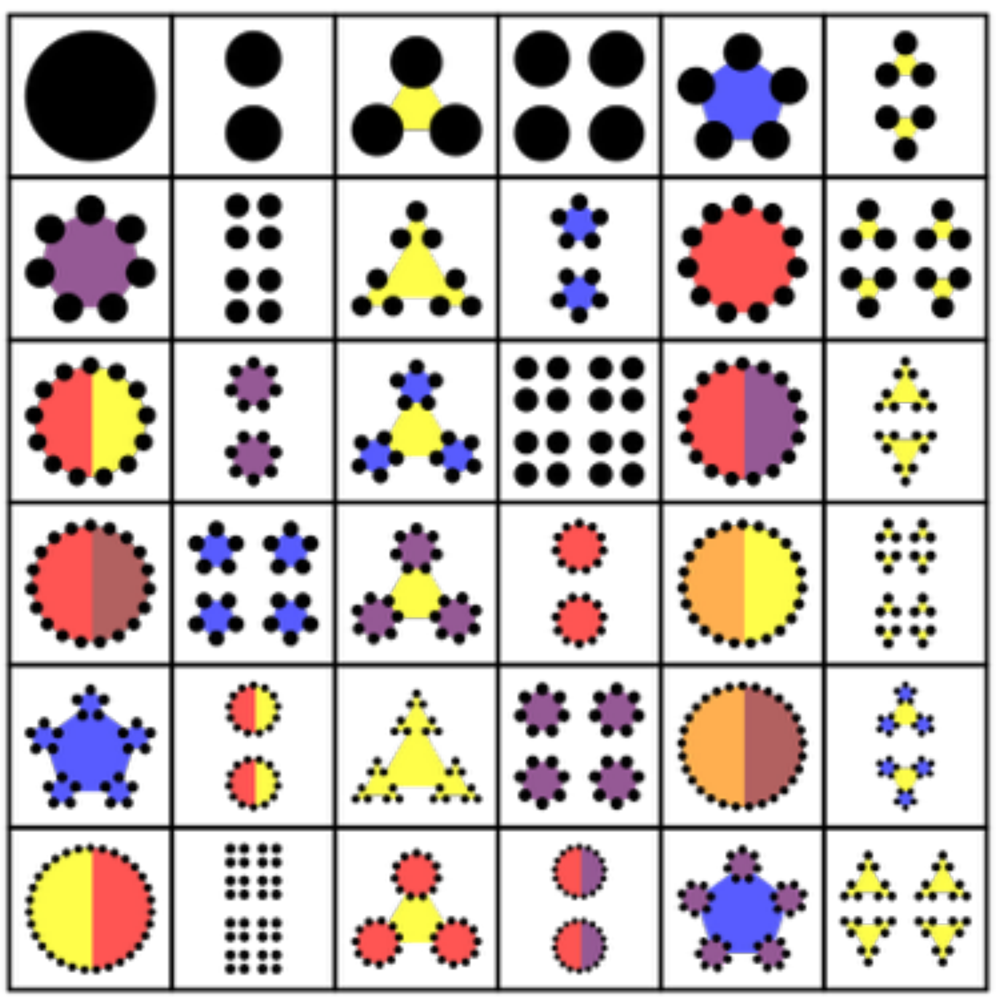
Applications

The `diagrams` Package

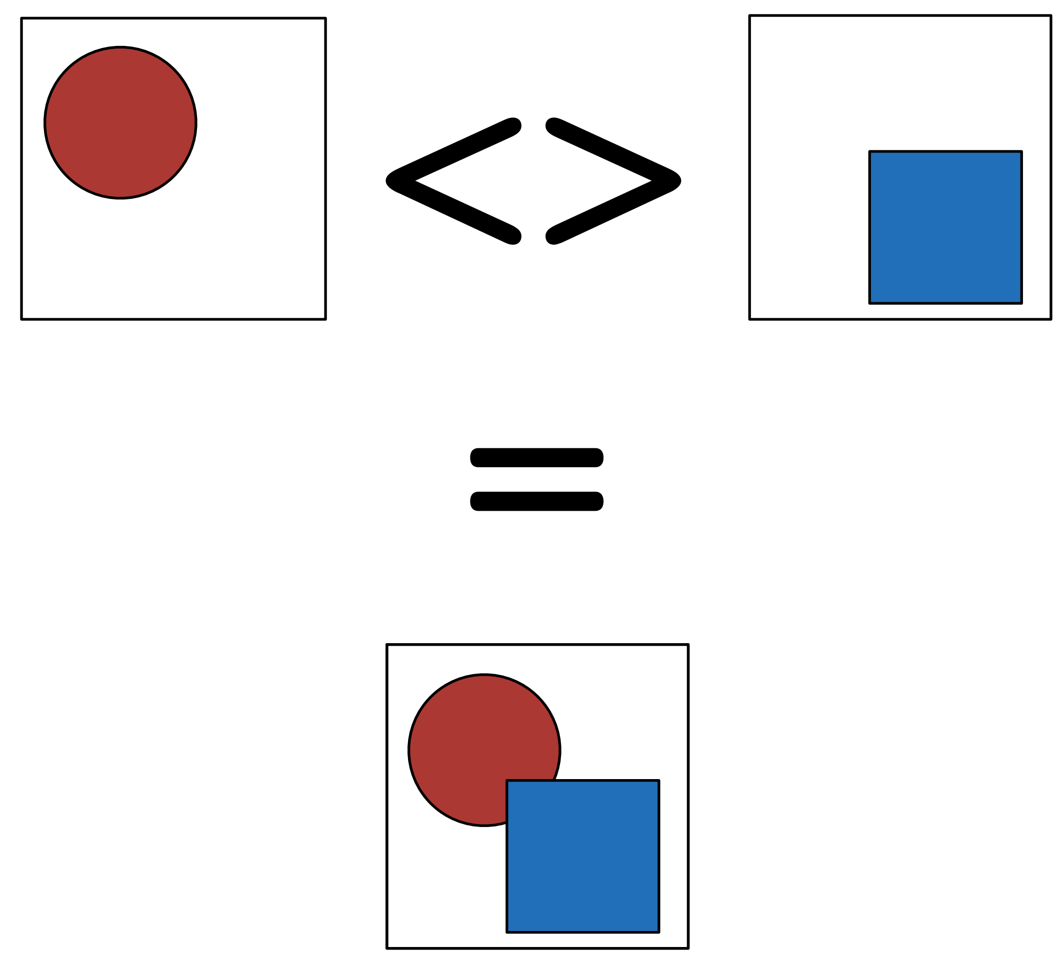


Brent Yorgey

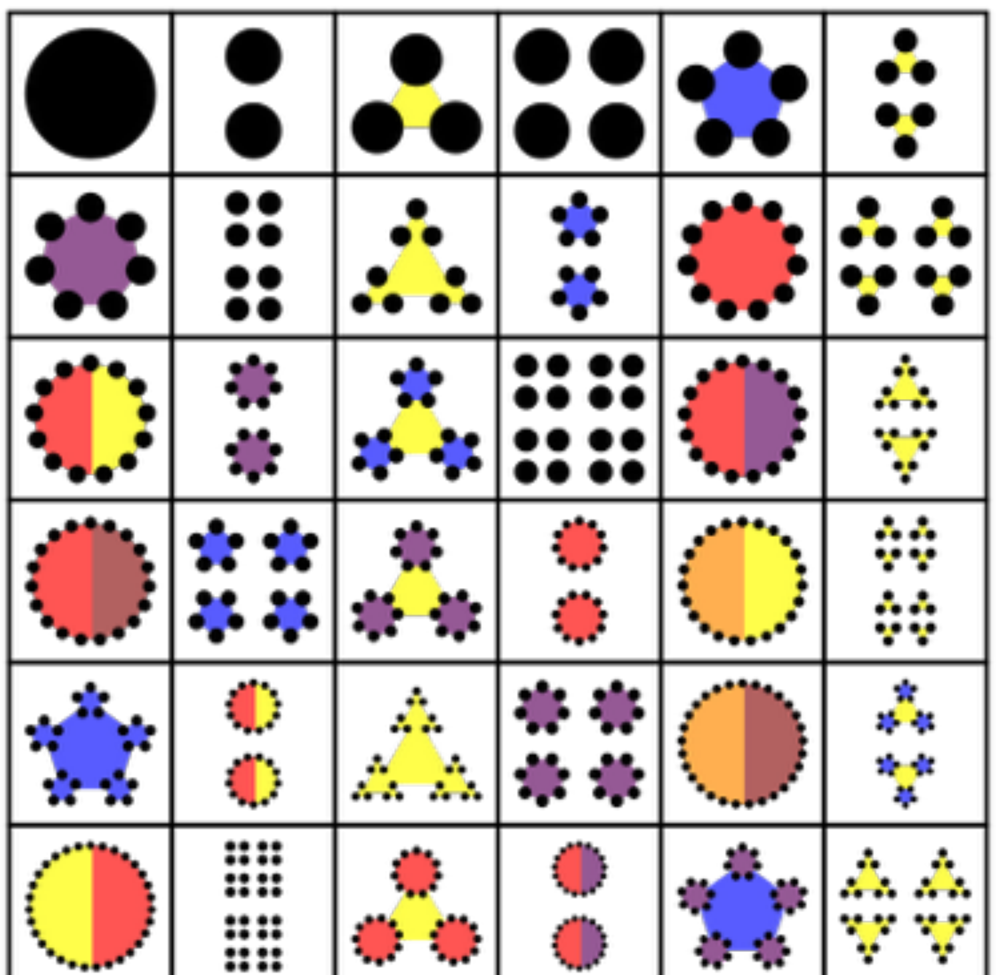
The `diagrams` Package



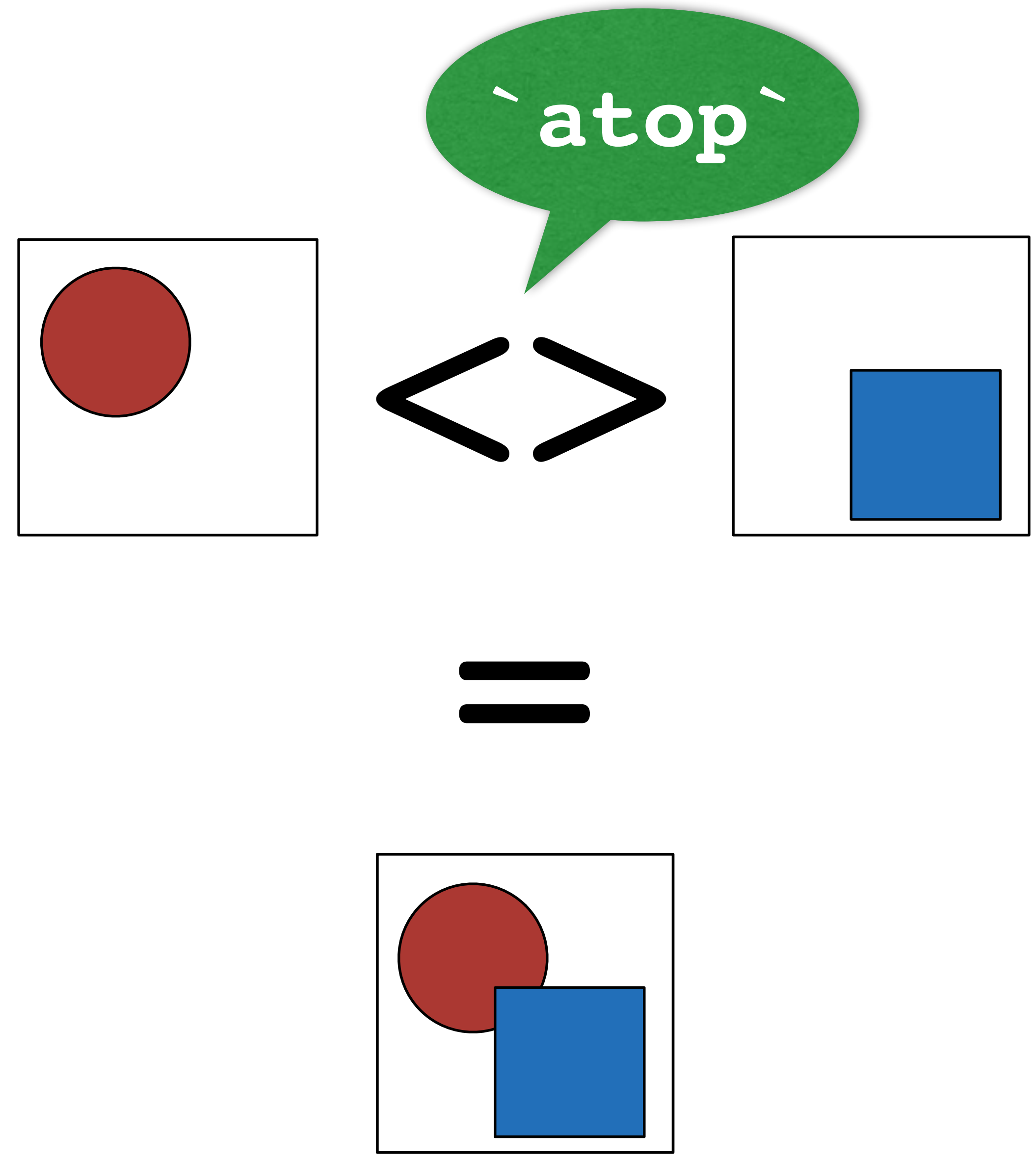
Brent Yorgey



The `diagrams` Package



Brent Yorgey



2

Compositional Settings

e.g., Command-Line Options:

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

Cabal

Duncan Coutts

2

Compositional Settings

e.g., Command-Line Options:

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

Cabal

Duncan Coutts

default setting <> setting update <> … <> setting update

2

Compositional Settings

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

2

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

2

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
     }
```

3

Data Aggregation

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

Data.Foldable

3

Data Aggregation

polymorphic collection

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

Data.Foldable

Data Aggregation

3

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.Foldable

3

Data Aggregation

```haskell
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
```

3

Data Aggregation

```haskell
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
```

3

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
…
```

Data.Foldable

3

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
…
```

Data.Foldable

3

Foldable/Traversable Proposal

aka
Burning Bridges Proposal

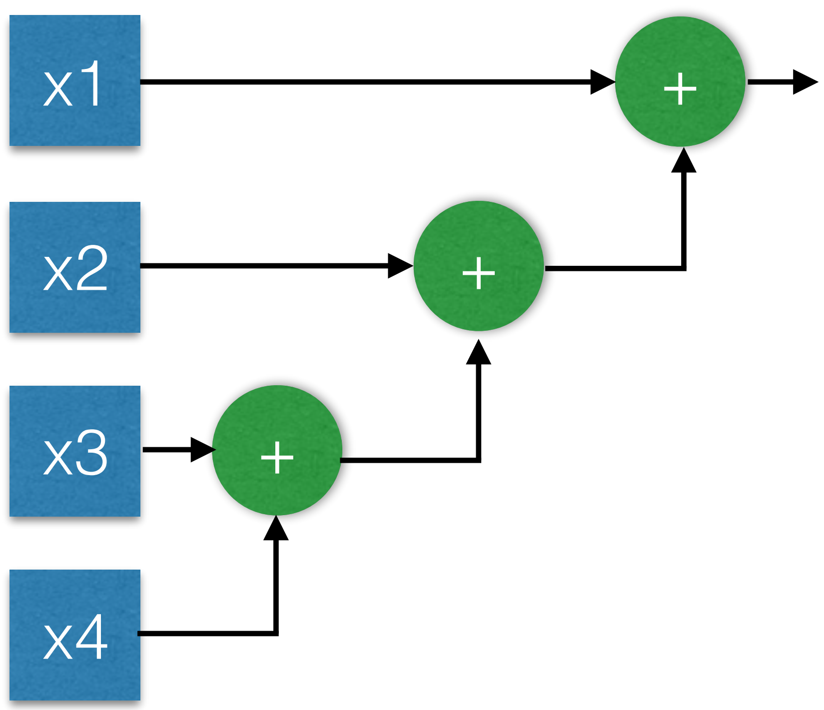
In Prelude as of GHC 7.10

```
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
…
```

Prelude

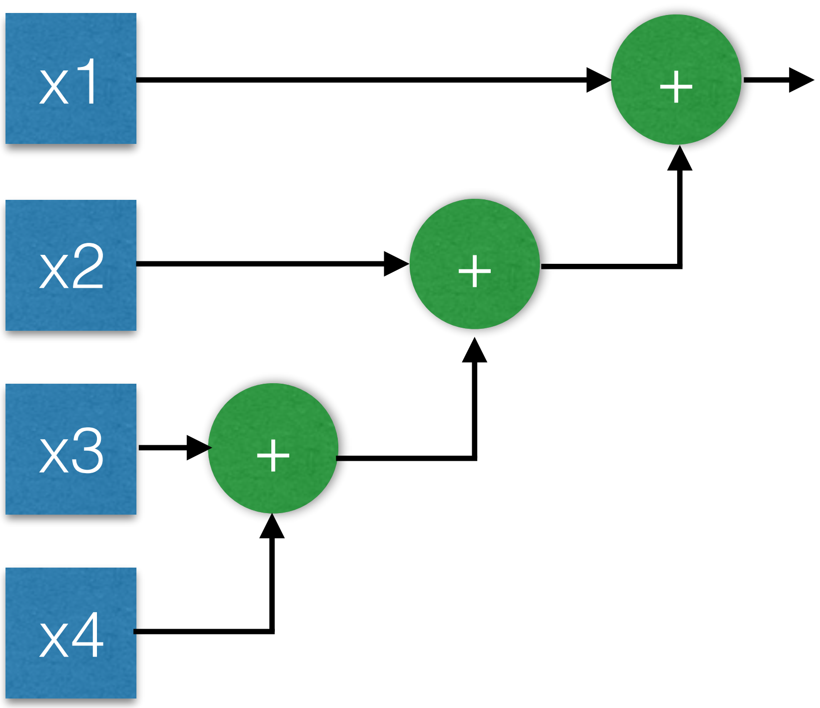
Divide & Conquer

Linear Processing



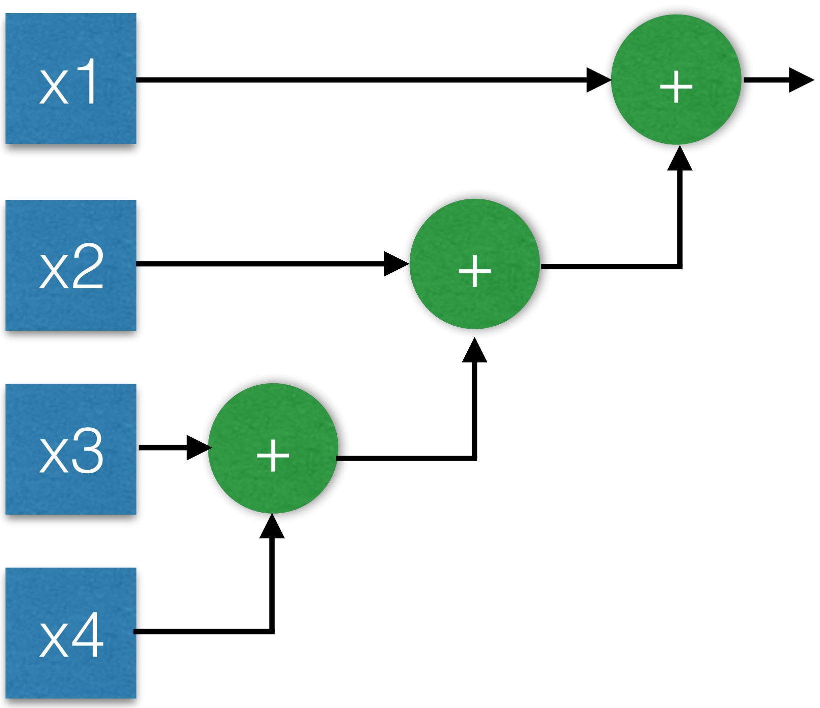
Linear Processing

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



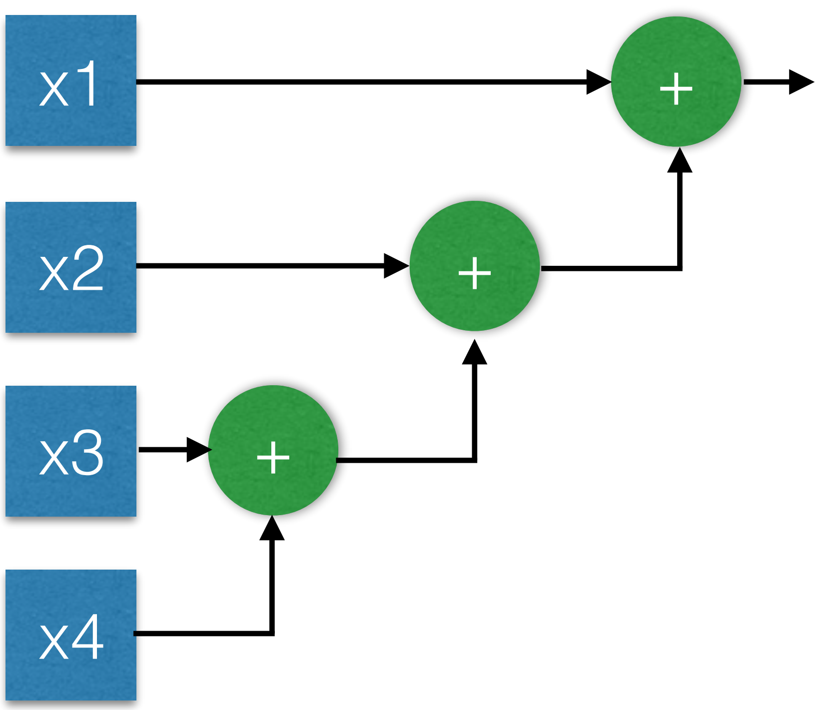
Linear Processing

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



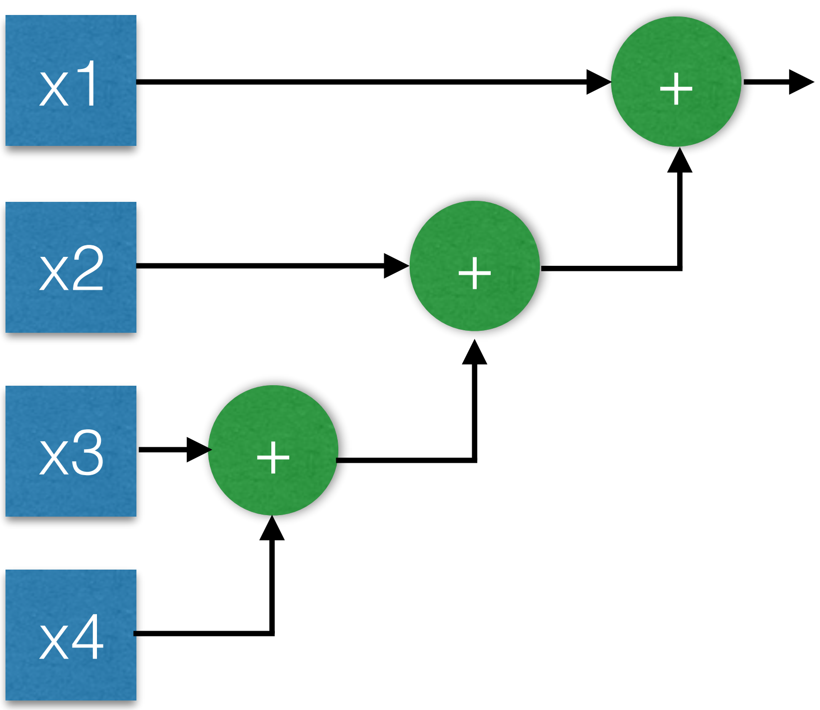

```
x1 <> (x2 <> x34)
```

Linear Processing



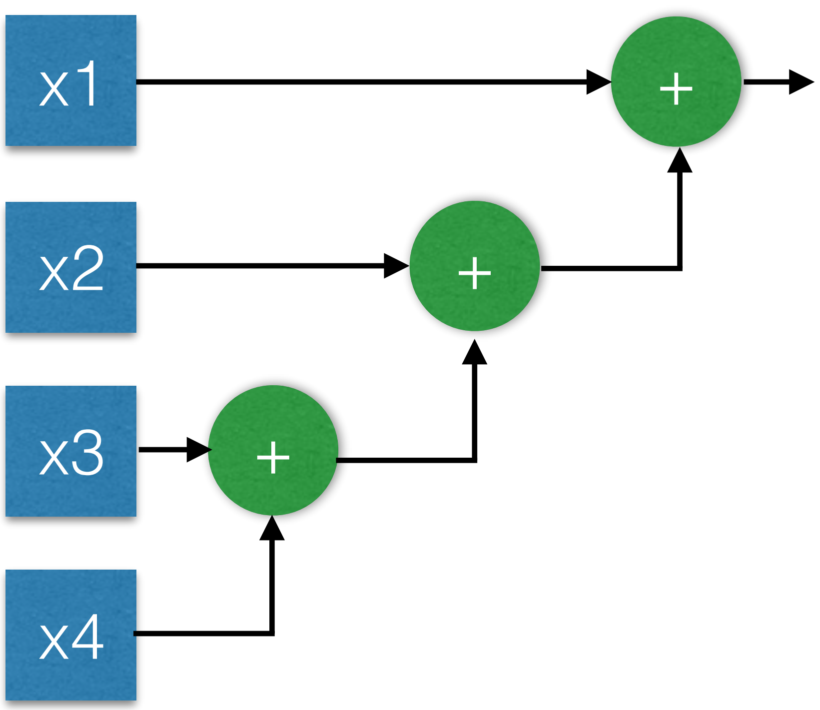

```
x1 <> (x2 <> (x3 <> x4))

x1 <> (x2 <> x34)

x1 <> (x234)
```

Linear Processing




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

x1 <> (x2 <> x34)

x1 <> (x234)

x1234
```

Linear Processing



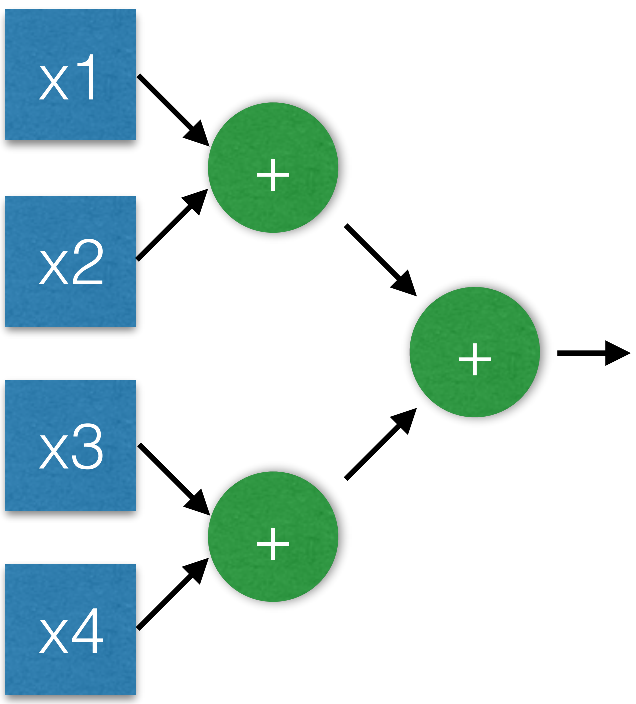
n-1 gate delays

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

x1 <> (x2 <> x34)

x1 <> (x234)

x1234
```

Linear Strategy

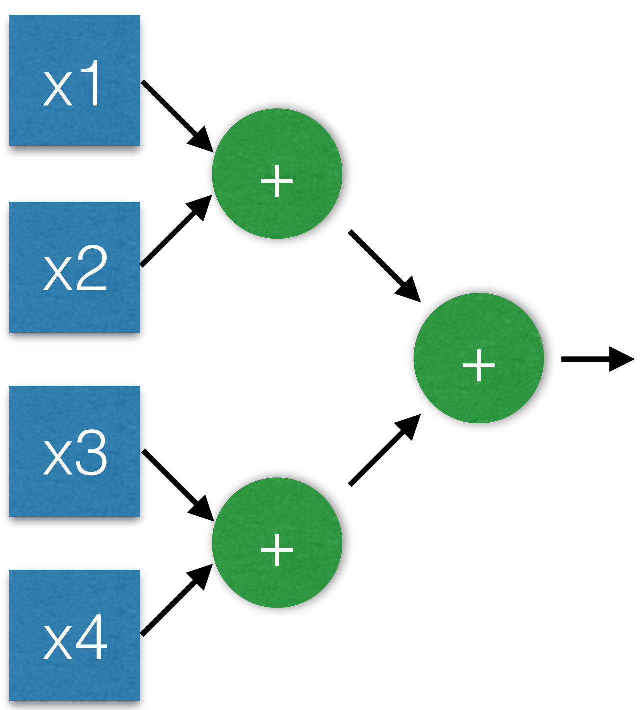
```
mconcat :: Monoid m => [m] -> m
mconcat  =  foldr mappend mempty
```

Parallel Processing

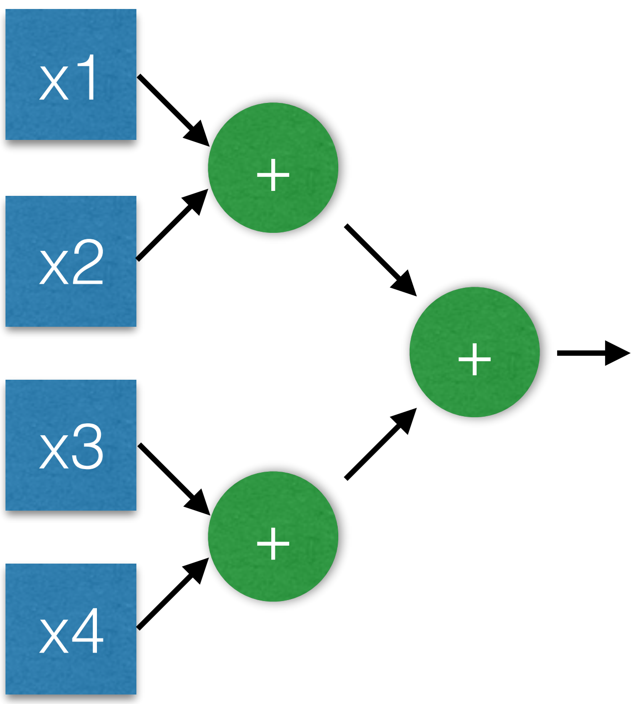


Parallel Processing

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

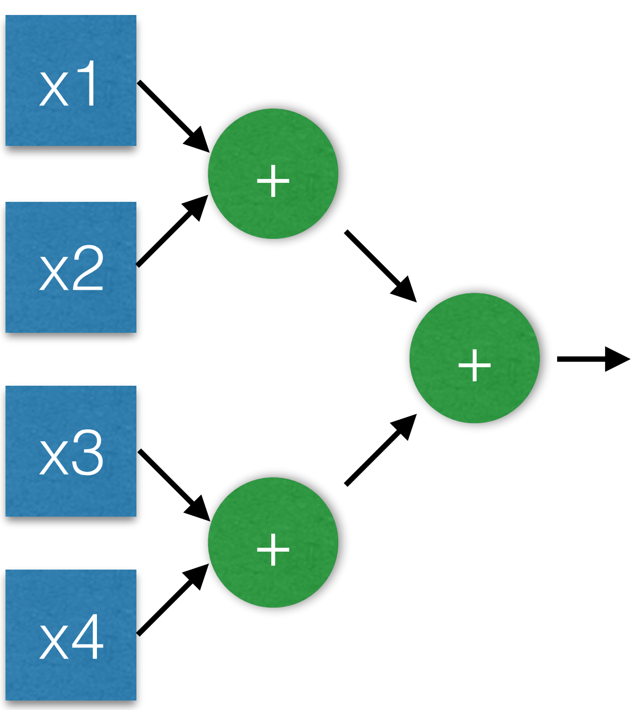


Parallel Processing




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

      x12 <> x34
```

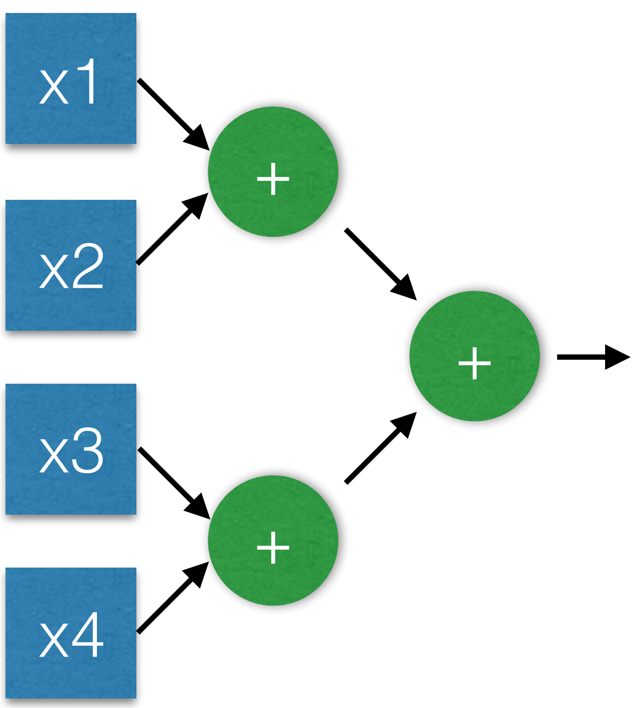
Parallel Processing




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

x12 <> x34

x1234
```

Parallel Processing

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

x12 <> 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 = (++)
```

Equational Reasoning

Left Unit Proof

```
mempty `mappend` ys

=

  ys
```

Proof Style: Equational Reasoning

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

Left Unit Proof

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

=



 ys
```

Proof Style: Equational Reasoning

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

Left Unit Proof

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



 ys
```

Proof Style: Equational Reasoning

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

Left Unit Proof

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

 ys
```

Proof Style: Equational Reasoning

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

Left Unit Proof

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

Proof Style: Equational Reasoning

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

Right Unit Proof

```
  l ++ []
=
  l
```

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

Right Unit Proof

```
  l ++ []
=
  l
```

Proof Style:
Structural Induction
+
Equational Reasoning

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

Base Case: `l = []`

```
  [] ++ []
=
  []
```

Proof Style: Structural Induction + Equational Reasoning

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

Base Case: `l = []`

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

Proof Style: Structural Induction + Equational Reasoning

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

Inductive Case: `l = x:xs`

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



  x:xs
```

Proof Style: Structural Induction + Equational Reasoning

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

Inductive Case: `l = 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: `l = x:xs`

```
  (x:xs) ++ []
= {- def. of (++) -}



  x:xs
```

Proof Style: Structural Induction + Equational Reasoning

Induction Hypothesis `xs ++ [] = xs`

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

Inductive Case: `l = x:xs`

```
  (x:xs) ++ []
= {- 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: `l = 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

```
 xs ++ (ys ++ zs)
=
 (xs ++ ys) ++ zs
```

Proof Style:
Structural Induction
+
Equational Reasoning

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

Associativity Proof

```
 xs ++ (ys ++ zs)
=
 (xs ++ ys) ++ zs
```

Homework → **Proof Style: Structural Induction + Equational Reasoning**

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

The Free Monoid

Monoid (Homo)morphism

a function between monoids

```
f :: M1 -> M2
```

such that:

```
f mempty     = 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

$$\begin{tikzcd} \mathtt{A} \arrow[d, bend right, "\mathtt{gen}"'] \\ \mathtt{M} \end{tikzcd}$$

Free Monoid

$$
\begin{tikzcd}
\texttt{A} \arrow[dr] \arrow[dd, bend right, "\texttt{gen}"'] & \\
& \texttt{FreeMonoid} \arrow[dl] \\
\texttt{M} &
\end{tikzcd}
$$

Free Monoid

$$\begin{tikzcd}
\mathtt{A} \arrow[dr] \arrow[dd, bend right, "\mathtt{gen}"'] & \\
 & \mathtt{FreeMonoid} \arrow[dl, "\mathtt{homomorphism}"] \\
\mathtt{M} &
\end{tikzcd}$$

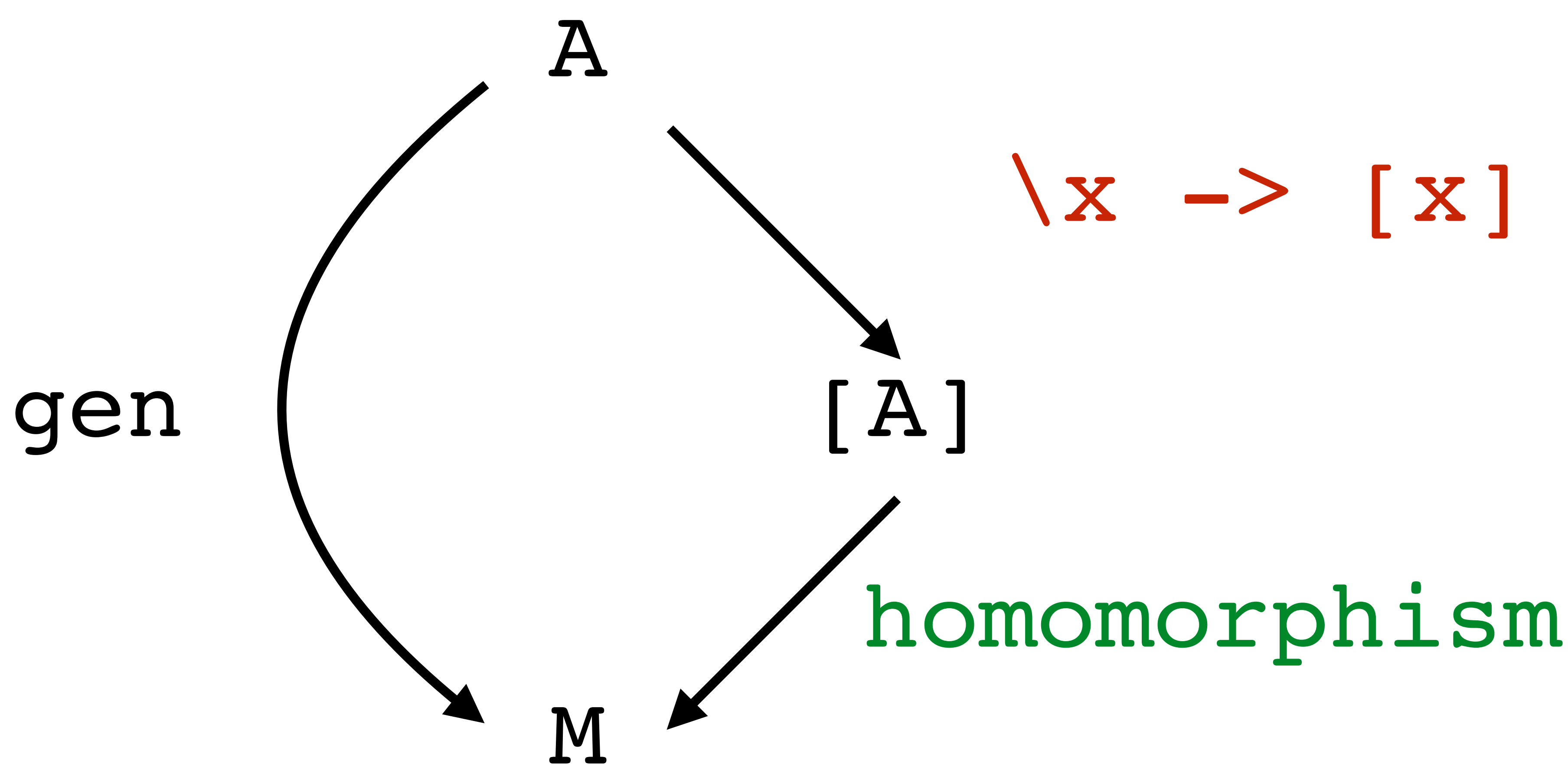
Free Monoid

$$
\begin{tikzcd}
\texttt{A} \arrow[dr, "\texttt{inj}"] \arrow[dd, bend right, "\texttt{gen}"'] & \\
& \texttt{FreeMonoid} \arrow[dl, "\texttt{homomorphism}"] \\
\texttt{M} &
\end{tikzcd}
$$

Free Monoid

$$
\begin{tikzcd}
 & \texttt{A} \arrow[dr, "\texttt{inj}"] \arrow[dd, bend right, "\texttt{gen}"'] & \\
 & & \texttt{[A]} \arrow[dl, "\texttt{homomorphism}"] \\
 & \texttt{M} &
\end{tikzcd}
$$

Free Monoid



Free Monoid

$$\begin{tikzcd}
\texttt{A} \arrow[dr, "\texttt{\textbackslash x -> [x]}"] \arrow[dd, bend right, "\texttt{gen}"'] & \\
& \texttt{[A]} \arrow[dl, "\texttt{mconcat . map gen}"] \\
\texttt{M} &
\end{tikzcd}$$

What is a Data.Foldable?

```
T a
```

What is a Data.Foldable?

Data structures that support some abstract nonsense?

$$T\ a \xrightarrow{\texttt{foldMap gen}} m$$

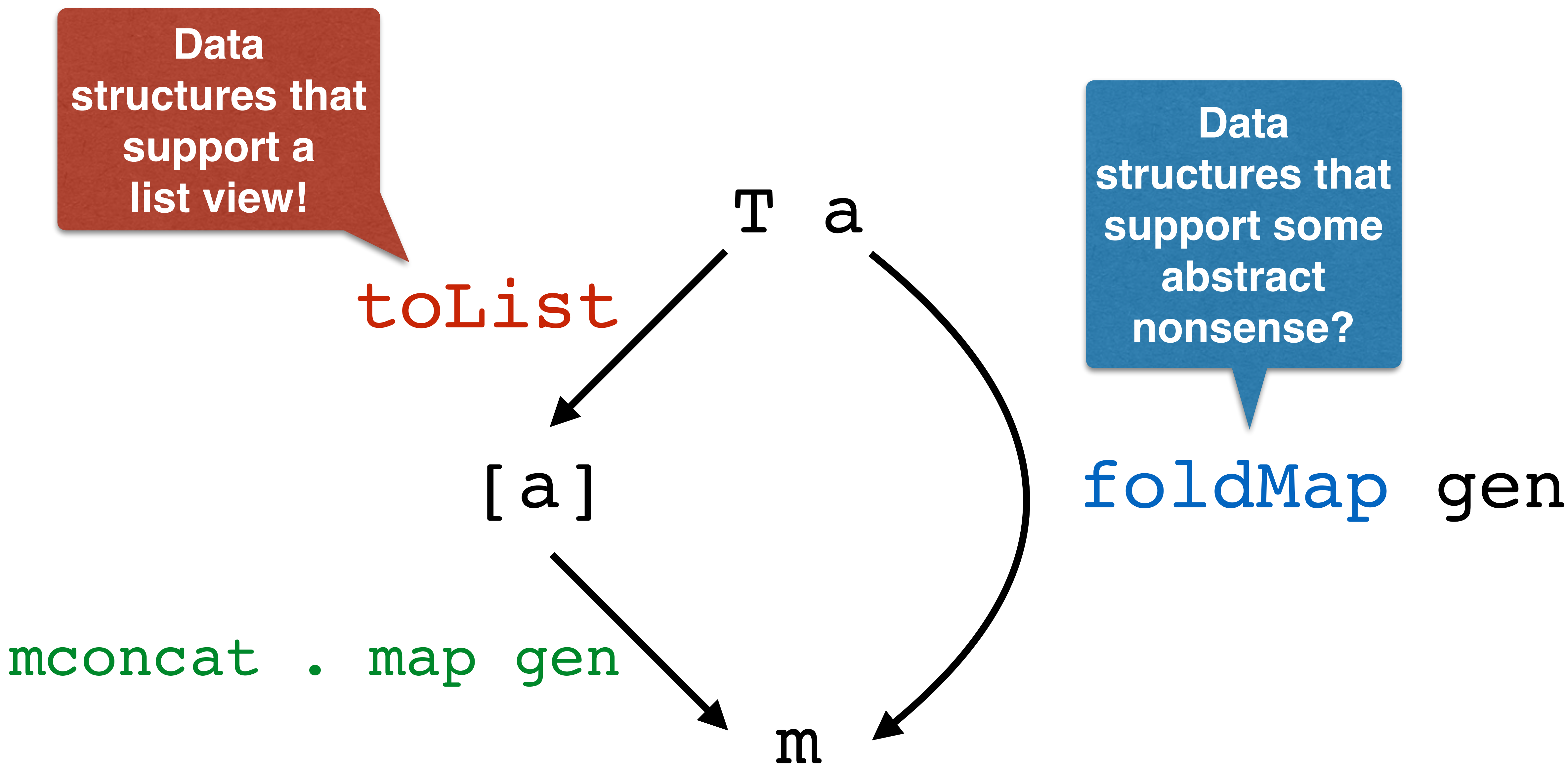
What is a Data.Foldable?

Data structures that support a list view!

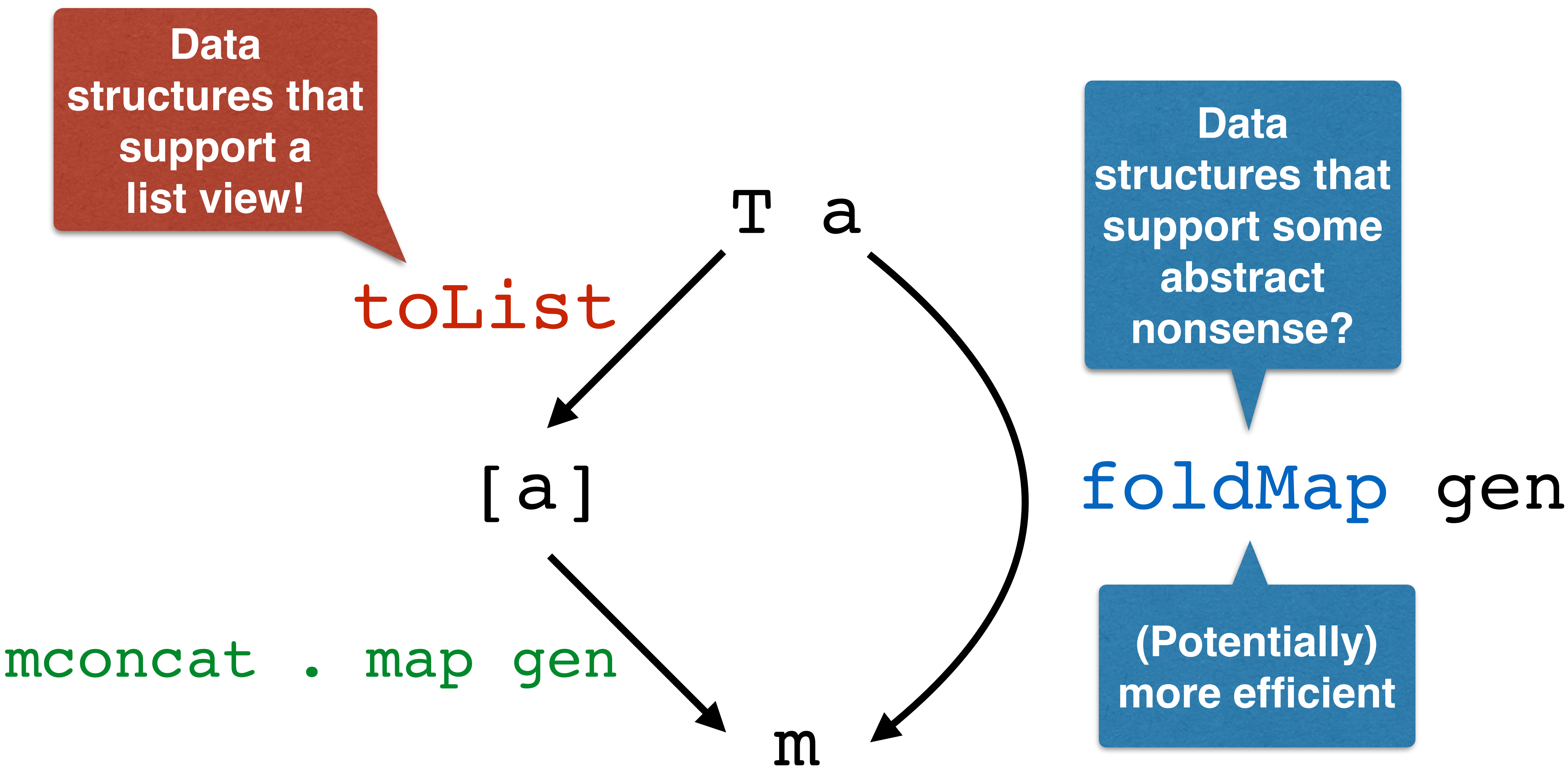
Data structures that support some abstract nonsense?

$$
\begin{tikzcd}
& \mathtt{T\ a} \arrow[dl, "\mathtt{toList}"'] \arrow[dd, bend left, "\mathtt{foldMap\ gen}"] \\
\mathtt{[a]} & \\
& \mathtt{m}
\end{tikzcd}
$$

What is a Data.Foldable?



What is a Data.Foldable?



Summary

Monoids

★ Simple concept from Algebra

★ Ubiquitous in Haskell

★ Cool Applications

★ List is the Free Monoid

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	...

Join the Google Group

Leuven Haskell User Group