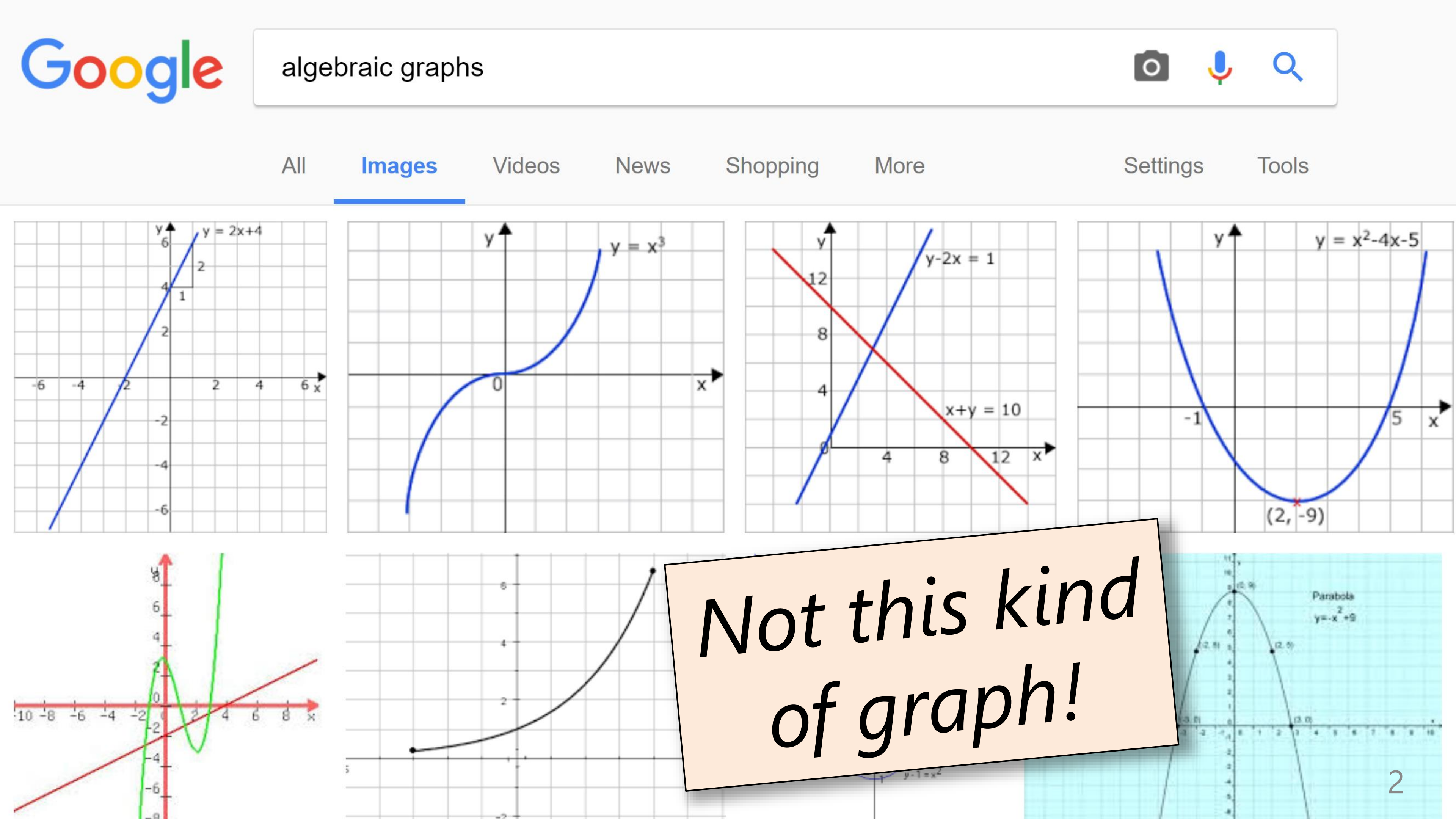
# Labelled Algebraic Graphs

## A Tale of Four Monoids

Andrey Mokhov

GitHub: @snowleopard, Twitter: @andreymokhov

*Haskell eXchange, October 2018*



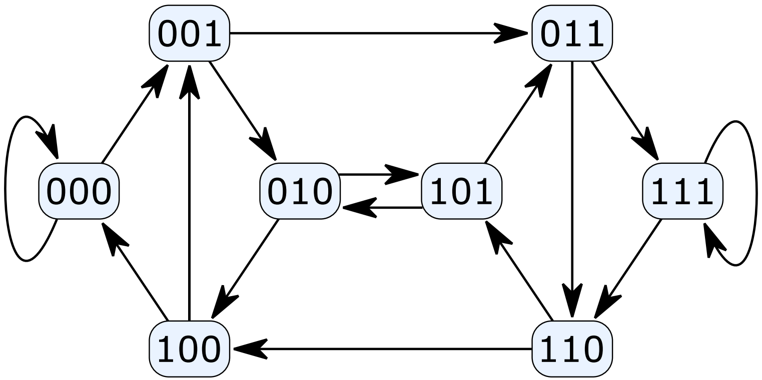

Not this kind of graph!

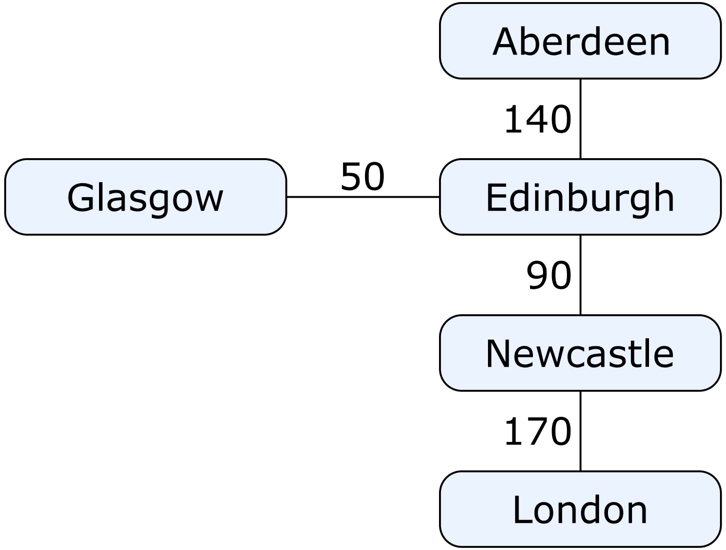
## This kind of graph:

- Labelled vertices
- Can have cycles
- Can have self-loops
- Directed/undirected
- **Labelled/unlabelled edges**

New!

- No vertex ports
- No ‘forbidden’ edges



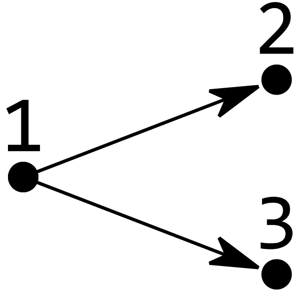



# **Part I:**
# Algebraic Graphs

# From math to Haskell

Pair $(V, E)$ such that $E \subseteq V \times V$

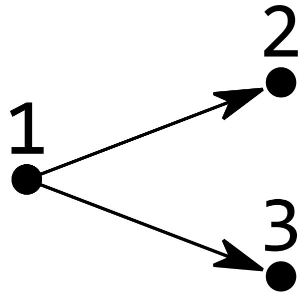
– Example: $(\{1,2,3\}, \{(1,2), (1,3)\})$



# From math to Haskell

Pair (V, E) such that $E \subseteq V \times V$

– Example: ({1,2,3}, {(1,2), (1,3)})




```haskell
data Graph a = Graph
     { vertices :: Set a
     , edges    :: Set (a,a) }

example :: Graph Int
example = Graph [1,2,3] [(1,2), (1,3)]
```

# Problem

Pair $(V, E)$ such that $E \subseteq V \times V$

- Non-example: $(\{1\}, \{(1,2)\})$

# Problem

Pair $(V, E)$ such that $E \subseteq V \times V$

– Non-example: $(\{1\}, \{(1,2)\})$

```
data Graph a = Graph
     { vertices :: Set a
     , edges    :: Set (a,a) }

nonExample :: Graph Int
nonExample = Graph [1] [(1,2)]
```

# Problem

Pair (V, E) such that $\boxed{E \subseteq V \times V}$

– Non-example: ({1}, {(1,2)})

Hard to express in types

```
data Graph a = Graph
    { vertices :: Set a
    , edges    :: Set (a,a) }

nonExample :: Graph Int
nonExample = Graph [1] [(1,2)]
```

# Problem

Pair (V, E) such that $\boxed{E \subseteq V \times V}$

– Non-example: ({1}, {(1,2)})

```
data Graph a = Graph
     { vertices :: Set a
     , edges    :: Set (a,a) }

nonExample :: Graph Int
nonExample = Graph [1] [(1,2)]
```

Hard to express in types

**Solution space:**

1. Fix Haskell
2. Fix math ✓

# Algebraic graphs

```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)
```

Every graph can be represented by a **Graph a** expression.
Non-graphs cannot be represented.

# Algebraic graphs

```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)
```

Every graph can be represented by a **Graph a** expression.
Non-graphs cannot be represented.

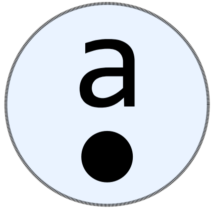
A. Mokhov, V. Khomenko. *"Algebra of Parameterised Graphs"*, ACM Transactions on Embedded Computing Systems, 2014

# Empty :: Graph a

# Empty :: Graph a

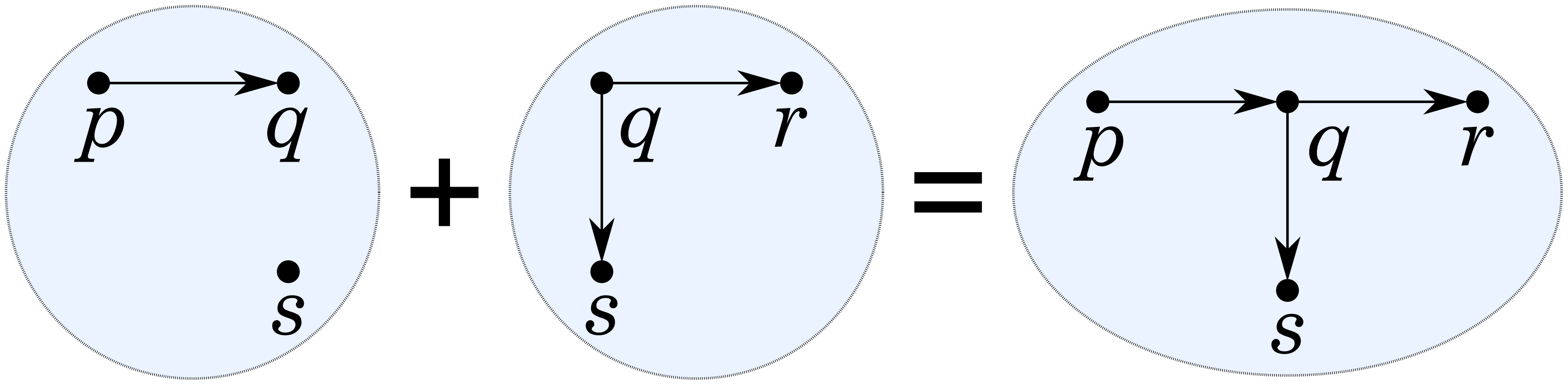
$(\emptyset, \emptyset)$

# Vertex :: a -> Graph a



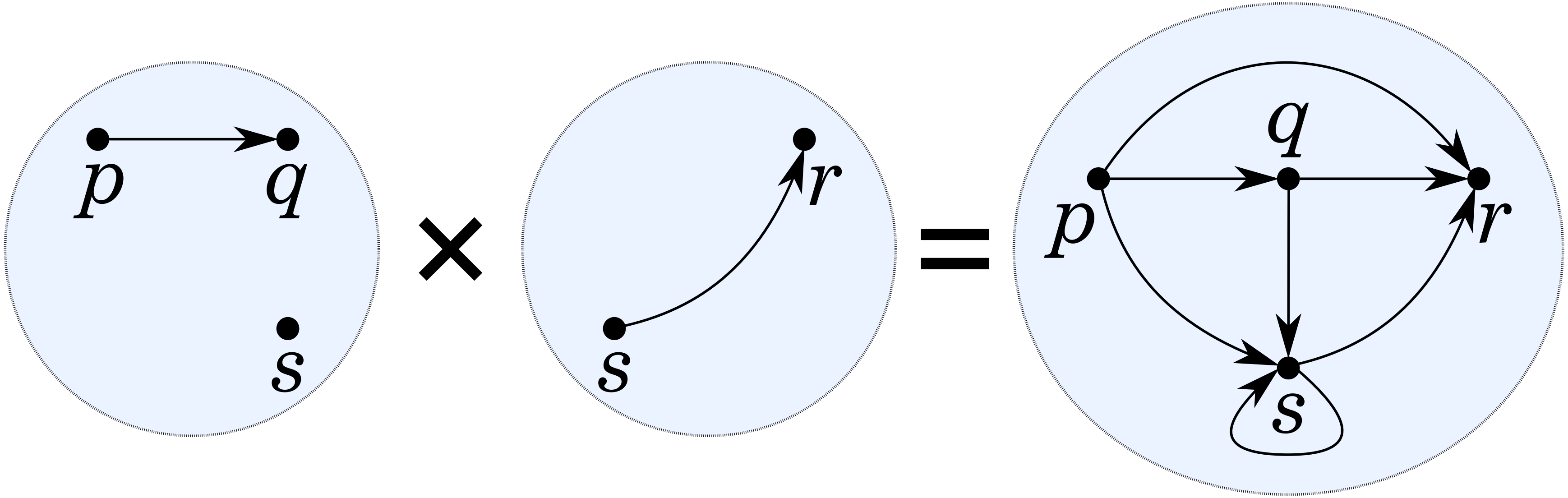

$(\{a\}, \emptyset)$

# Overlay :: Graph a -> Graph a -> Graph a




$$(V_1, E_1) + (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$$

# Connect :: Graph a -> Graph a -> Graph a




$(V_1, E_1) \times (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$

# Algebraic graphs

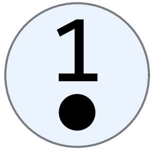
```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)
```

`Empty` is the empty graph $(\emptyset, \emptyset)$

`Vertex a` is the singleton graph $(\{a\}, \emptyset)$

`Overlay` of $(V_1, E_1)$ and $(V_2, E_2)$ is $(V_1 \cup V_2, E_1 \cup E_2)$

`Connect` of $(V_1, E_1)$ and $(V_2, E_2)$ is $(V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$

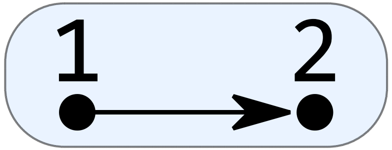



`Vertex 1`


`Vertex 2`

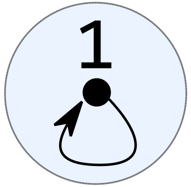

`Overlay (Vertex 1) (Vertex 2)`

Or simply 1 + 2



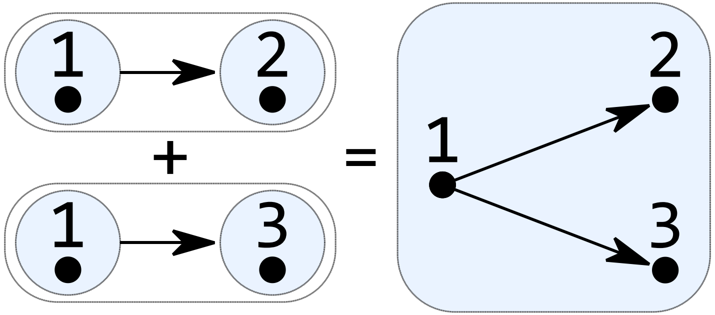

`Connect (Vertex 1) (Vertex 2)`

Or simply 1 × 2



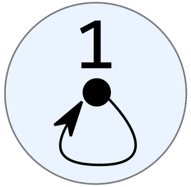

$1 \times 1$

```
Connect (Vertex 1) (Vertex 1)
```




$1 \times 2 + 1 \times 3$

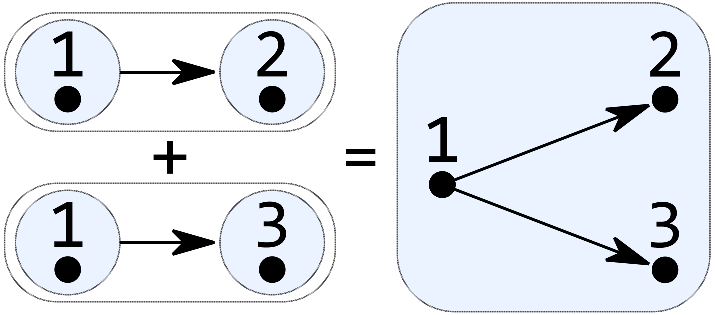
```
Overlay (Connect (Vertex 1) (Vertex 2))
        (Connect (Vertex 1) (Vertex 3))
```




$1 \times 1$

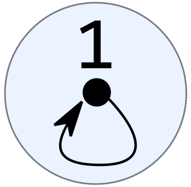
```
Connect (Vertex 1) (Vertex 1)
```

*Can we factor out* 1?




$1 \times 2 + 1 \times 3$

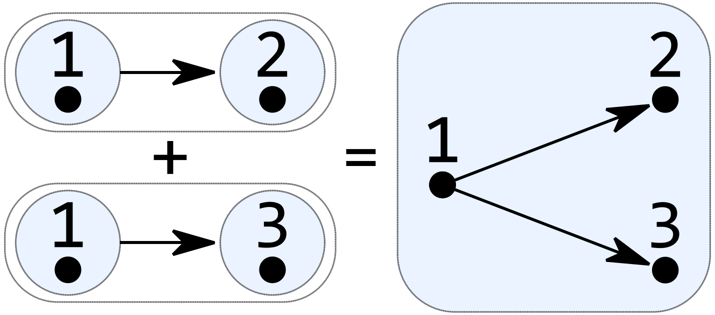
```
Overlay (Connect (Vertex 1) (Vertex 2))
        (Connect (Vertex 1) (Vertex 3))
```




$1 \times 1$

```
Connect (Vertex 1) (Vertex 1)
```

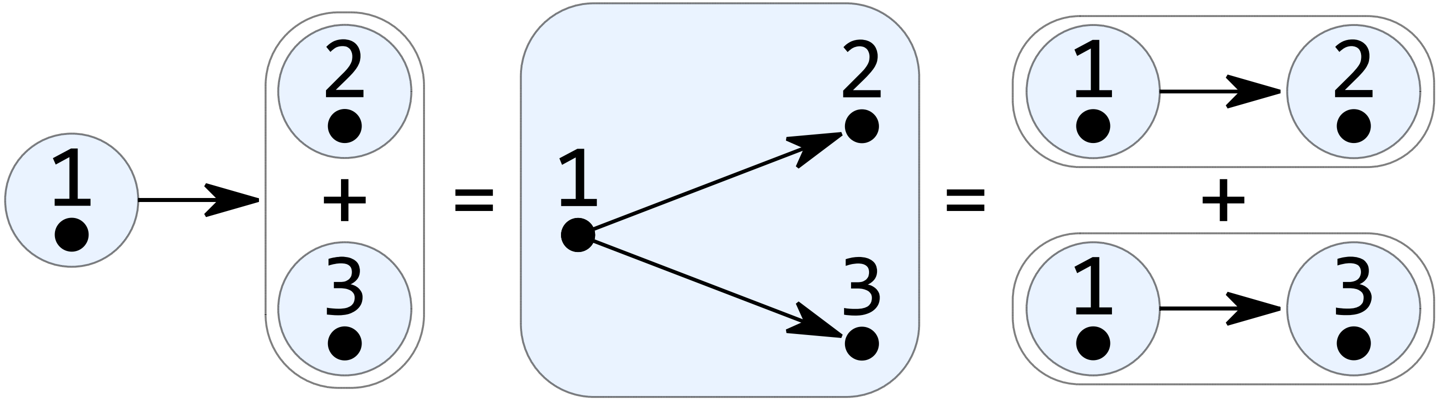
Can we factor out 1? Yes!




$1 \times 2 + 1 \times 3$

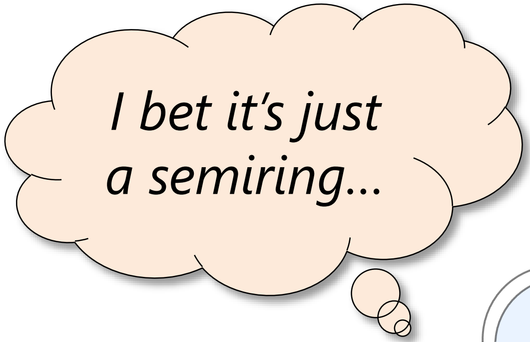
```
Overlay (Connect (Vertex 1) (Vertex 2))
        (Connect (Vertex 1) (Vertex 3))
```

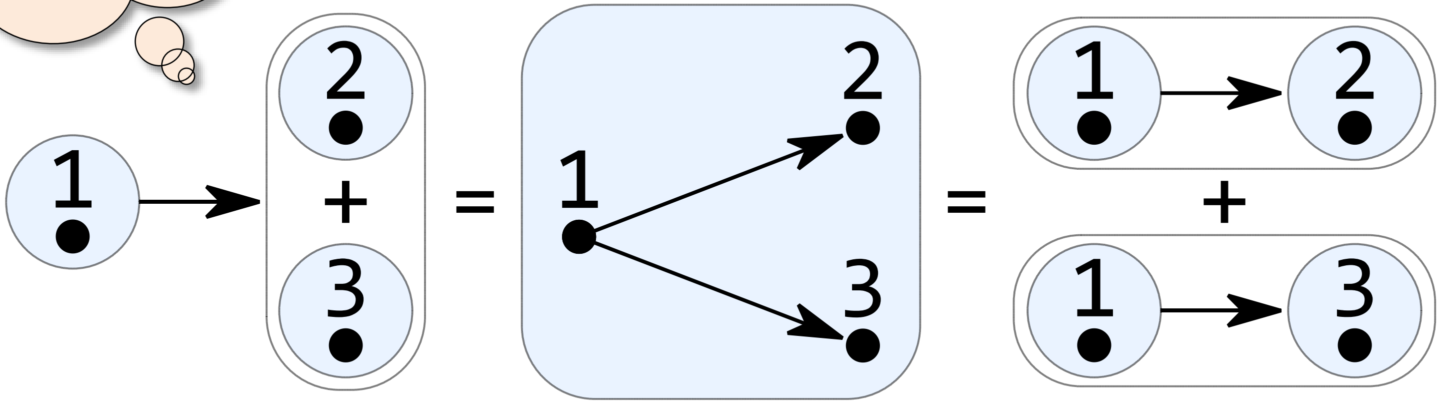
# Distributivity




$$x(y + z) = xy + xz$$
$$(x + y)z = xz + yz$$

# Distributivity

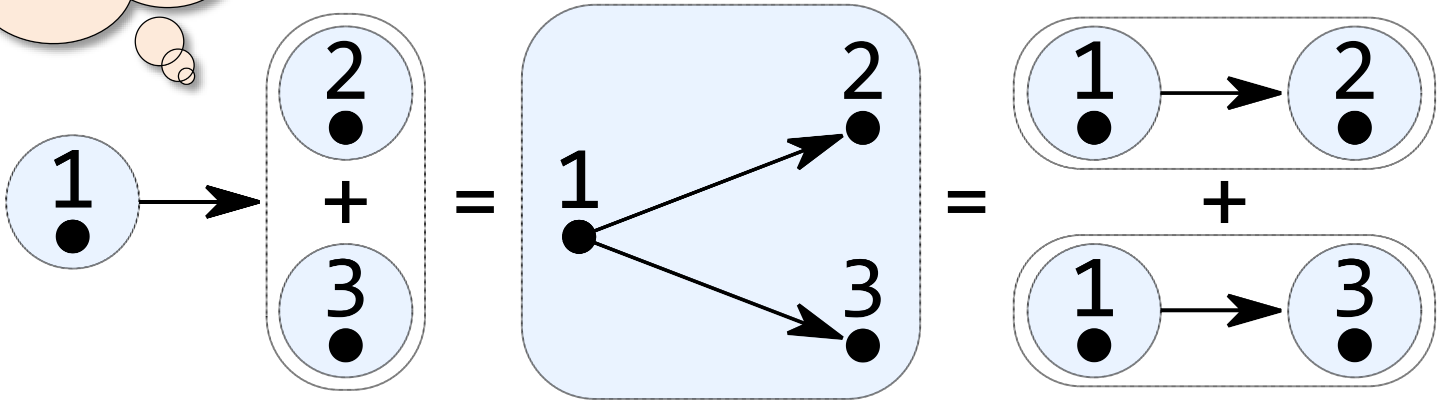



$$x(y + z) = xy + xz$$
$$(x + y)z = xz + yz$$

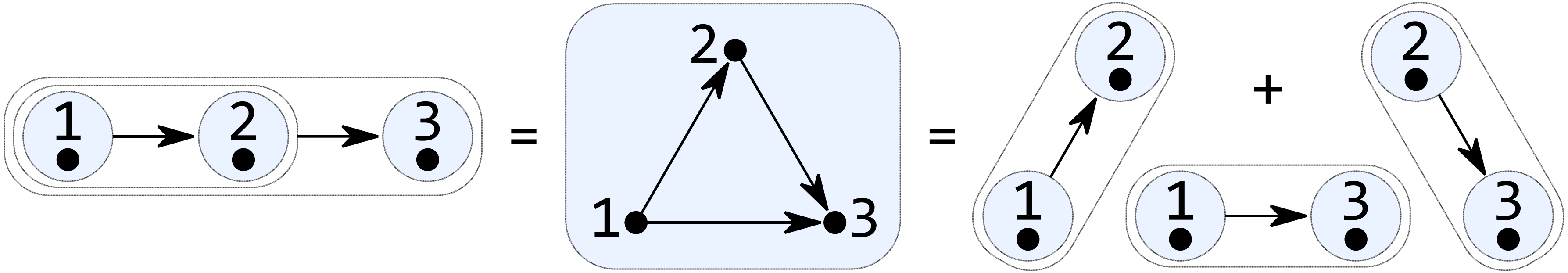
# Distributivity

*I bet it's just a semiring…* No!




$$x(y + z) = xy + xz$$
$$(x + y)z = xz + yz$$

# Decomposition




**xyz = xy + xz + yz**

**Intuition:** any graph expression can be broken down into an overlay of vertices and edges

# Algebraic structure

## Axioms:

Overlay $+$ is commutative and associative

Connect $\times$ is associative

The empty graph $\varepsilon$ is the identity of connect $\times$

Connect $\times$ distributes over overlay $+$

Decomposition: $xyz = xy + xz + yz$

## Theorems:

Overlay $+$ is idempotent and has $\varepsilon$ as the identity

# Algebraic structure

## Axioms:

Overlay + is commutative and associative

Connect × is associative

The empty graph ε is the identity of connect ×

Connect × distributes over overlay +

Decomposition: $xyz = xy + xz + yz$

## Theorems:

Overlay + is idempotent and has ε as the identity

Monoid count: 2

# Decomposition axiom is strange

A proof that $\mathbf{0} = \mathbf{1}$:

| | |
|---|---|
| $\mathbf{0} = 0 \times 1 \times 1$ | ($\mathbf{1}$ is identity of $\times$) |
| $= 0 \times 1 + 0 \times 1 + 1 \times 1$ | (decomposition) |
| $= 0 + 0 + 1$ | ($\mathbf{1}$ is identity of $\times$) |
| $= \mathbf{1}$ | ($\mathbf{0}$ is identity of $+$) |

# Decomposition axiom is strange

A proof that $0 = 1$:

($1$ is identity of $\times$)

(decomposition)

($1$ is identity of $\times$)

($0$ is identity of $+$)

# Other flavours of the algebra

Non-empty graphs: Drop the **Empty** constructor

Undirected graphs: Add $xy = yx$

Reflexive graphs: Add **Vertex v = Vertex v** $\times$ **Vertex v**

Transitive graphs: Add $y \neq \varepsilon \implies xy + yz = xy + xz + yz$

... and their various combinations:

- Preorders = Reflexive + Transitive
- Equivalence relations = Undirected + Reflexive + Transitive

# Part II:

## A library for algebraic graphs in just 100 lines of code

# Reusing functional programming abstractions

```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)

instance Eq a  => Eq  (Graph a) -- via normal form
instance Num a => Num (Graph a)
instance Functor      Graph
instance Applicative  Graph
instance Monad        Graph
instance MonadPlus    Graph
...
```

# Reusing functional programming abstractions

```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)

instance Eq a  => Eq  (Graph a) -- via normal form
instance Num a => Num (Graph a)
instance Functor       Graph
instance Applicative   Graph
instance Monad         Graph
instance MonadPlus     Graph
...
```

Correspond to basic graph transformations: merging, splitting, removing vertices, etc.

# Graph as a Num

```
instance Num a => Num (Graph a) where
    fromInteger = Vertex . fromInteger
    (+)         = Overlay
    (*)         = Connect
    signum      = const Empty
    abs         = id
    negate      = id
```

```
example :: Graph Int
example = 1 * (2 + 3)
-- Instead of: Graph [1,2,3] [(1,2), (1,3)]
```

# From four primitives to a library

```
-- An abstract interface or a type class
empty   :: Graph a
vertex  :: a -> Graph a
overlay :: Graph a -> Graph a -> Graph a
connect :: Graph a -> Graph a -> Graph a
```

# From four primitives to a library

```haskell
-- An abstract interface or a type class
empty   :: Graph a
vertex  :: a -> Graph a
overlay :: Graph a -> Graph a -> Graph a
connect :: Graph a -> Graph a -> Graph a
```

```haskell
-- Combine primitives into larger graphs
vertices :: [a] -> Graph a
vertices vs = foldr overlay empty (map vertex vs)

edge :: a -> a -> Graph a
edge u v = connect (vertex u) (vertex v)
```

# Folding algebraic graphs

```haskell
-- Like foldr but for graphs
foldg :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b)
      -> Graph a -> b
foldg e v o c = go
  where
    go Empty          = e
    go (Vertex  x  ) = v x
    go (Overlay x y) = o (go x) (go y)
    go (Connect x y) = c (go x) (go y)
```

# Folding algebraic graphs

```
-- Like foldr but for graphs
foldg :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b)
      -> Graph a -> b
foldg e v o c = go
  where
    go Empty           = e
    go (Vertex  x  ) = v x
    go (Overlay x y) = o (go x) (go y)
    go (Connect x y) = c (go x) (go y)
```

```
isEmpty :: Graph a -> Bool
isEmpty = foldg True (const False) (&&) (&&)
```

# Folding algebraic graphs

```haskell
-- Like foldr but for graphs
foldg :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b)
      -> Graph a -> b
foldg e v o c = go
  where
    go Empty          = e
    go (Vertex  x  ) = v x
    go (Overlay x y) = o (go x) (go y)
    go (Connect x y) = c (go x) (go y)
```

The arguments **(e, v, o, c)** must satisfy the laws of the algebra

```haskell
isEmpty :: Graph a -> Bool
isEmpty = foldg True (const False) (&&) (&&)
```

# Folding algebraic graphs

```
hasVertex :: Eq a => a -> Graph a -> Bool
hasVertex x = foldg False (==x) (||) (||)

vertexSet :: Ord a => Graph a -> Set a
vertexSet = foldg Set.empty singleton union union

transpose :: Graph a -> Graph a
transpose = foldg empty vertex overlay (flip connect)

size :: Graph a -> Int
size = foldg 1 (const 1) (+) (+)
```

# Folding algebraic graphs

```
hasVertex :: Eq a => a -> Graph a -> Bool
hasVertex x = foldg False (==x) (||) (||)

vertexSet :: Ord a => Graph a -> Set a
vertexSet = foldg Set.empty singleton union union

transpose :: Graph a -> Graph a
transpose = foldg empty vertex overlay (flip connect)

size :: Graph a -> Int
size = foldg 1 (const 1) (+) (+)
```

Breaks laws:
size(x) ≠ size(x+ε)

# **Part III:**
# Labelled Algebraic Graphs

# Labelled algebraic graphs

```haskell
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)
```

```haskell
data Graph e a = Empty
               | Vertex a
               | Connect e (Graph e a) (Graph e a)
```

# Labelled algebraic graphs

```haskell
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)
```

```haskell
data Graph e a = Empty
               | Vertex a
               | Connect e (Graph e a) (Graph e a)
```

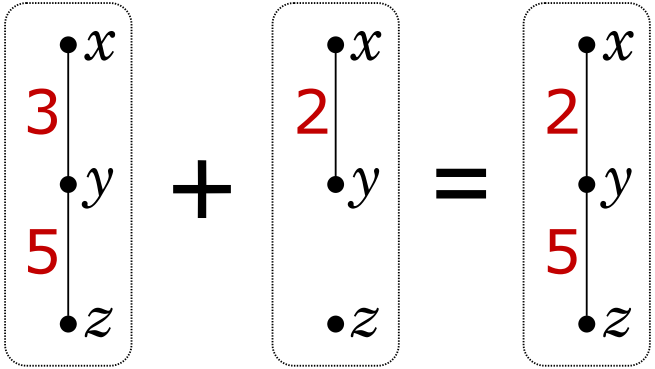
**Main idea:**

`Overlay = Connect 0`

# Labels

We need **zero label** `0` to indicate a missing edge

- Labels are **edge capacities**: `0` is just `0`
- Labels are **distances** between vertices: `0` is ∞
- Labels are **regular expressions**: `0` is ∅




We need a way to **compose 'parallel' labels**:

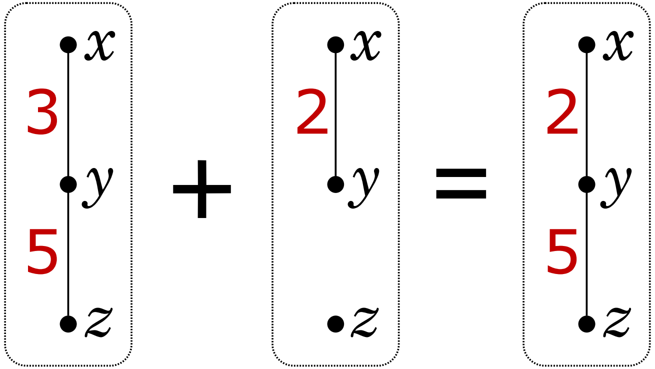
- Labels are **edge capacities**: `<+>` is **max**
- Labels are **distances** between vertices: `<+>` is **min**
- Labels are **regular expressions**: `<+>` is **|**

To stay sane we better require `<+>` to be **associative** and have **identity** `0`

# Labels

We need **zero label** 0 to indicate a missing edge

- Labels are **edge capacities**: 0 is just 0
- Labels are **distances** between vertices: 0 is ∞
- Labels are **regular expressions**: 0 is ∅




We need a way to **compose 'parallel' labels**:

- Labels are **edge capacities**: <+> is **max**
- Labels are **distances** between vertices: <+> is **min**
- Labels are **regular expressions**: <+> is **|**

Monoid count: 3

To stay sane we better require <+> to be **associative** and have **identity** 0

# Overlaying edge-labelled graphs

```haskell
data Graph e a = Empty
               | Vertex a
               | Connect e (Graph e a) (Graph e a)

-- Convenient aliases
zero :: Monoid e => e       (<+>) :: Monoid e => e -> e -> e
zero = mempty               (<+>) = mappend

overlay :: Monoid e => Graph e a -> Graph e a -> Graph e a
overlay = Connect zero
```

We will continue using + to denote the graph overlay operation.

# Connecting edge-labelled graphs

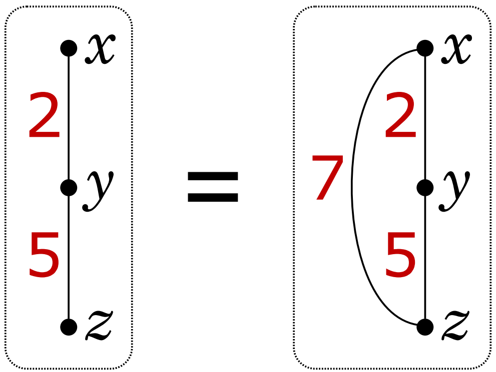
```haskell
data Graph e a = Empty
               | Vertex a
               | Connect e (Graph e a) (Graph e a)

edge :: e -> a -> a -> Graph e a
edge e x y = Connect e (Vertex x) (Vertex y)

-- Convenient ternary-ish operator
(-<) :: a -> e -> (a,e)        (>-) :: (a,e) -> a -> Graph e a
x -< e = (x,e)                 (x,e) >- y = edge e x y
```

We'll use `x-<e>-y` to denote an edge connecting `x` and `y` with label `e`

# Composing labels in sequence

We need a way to **compose 'sequences' of labels**:

- Labels are **edge capacities**: <.> is **min**
- Labels are **distances** between vertices: <.> is **+**
- Labels are **regular expressions**: <.> is **;**




We need **label** 1 to indicate the empty sequence

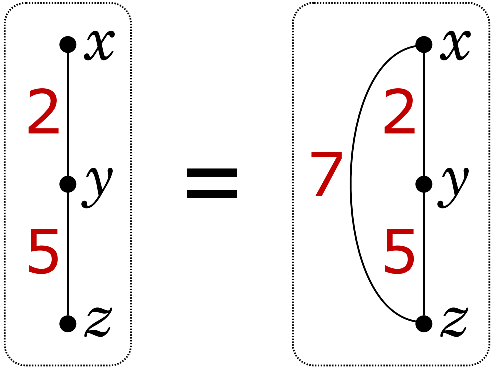
- Labels are **edge capacities**: 1 is **∞**
- Labels are **distances** between vertices: 1 is **0**
- Labels are **regular expressions**: 1 is **ε**

To stay sane we better require <.> to be **associative** and have **identity** 1

# Composing labels in sequence

We need a way to **compose 'sequences' of labels**:

- Labels are **edge capacities**: `<.>` is **min**
- Labels are **distances** between vertices: `<.>` is **+**
- Labels are **regular expressions**: `<.>` is **;**




We need **label 1** to indicate the empty sequence

- Labels are **edge capacities**: 1 is ∞
- Labels are **distances** between vertices: 1 is **0**
- Labels are **regular expressions**: 1 is **ε**

Monoid count: **4**

To stay sane we better require `<.>` to be **associative** and have **identity** 1

# Composing labels in sequence

```haskell
data Graph e a = Empty
               | Vertex a
               | Connect e (Graph e a) (Graph e a)

class Monoid e => Semiring e where
    one   :: e
    (<.>) :: e -> e -> e

-- The connect operator from unlabelled algebraic graphs
(×) :: Semiring e => Graph e a -> Graph e a -> Graph e a
(×) = Connect one
```

# Unlabelled graphs are Bool-labelled

```haskell
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)
```

```
e=Bool  0=False  1=True  (<+>)=(||)  (<.>)=(&&)
```

```haskell
data Graph e a = Empty
               | Vertex a
               | Connect e (Graph e a) (Graph e a)
```

# Unlabelled graphs are Bool-labelled

```
data Graph a = Empty
             | Verte
             | Overlay      aph a) (Graph a)
             | Connect (Graph       Graph
```

1

2

3

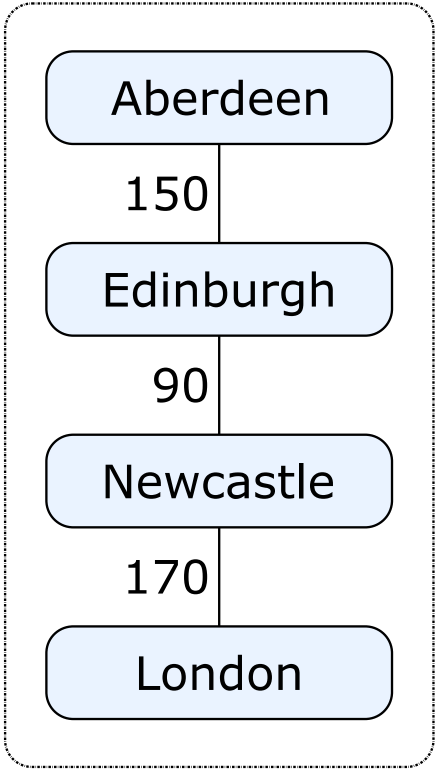
4

```
e=Bool  0=False  1=True  (<+>)=(||)  (<.>)=(&&)
```

```
data Graph e a = Empty
               | Vertex a
               | Connect e (Graph e a) (Graph e a)
```

# Example 1: transportation networks

$EastCoast\ network$

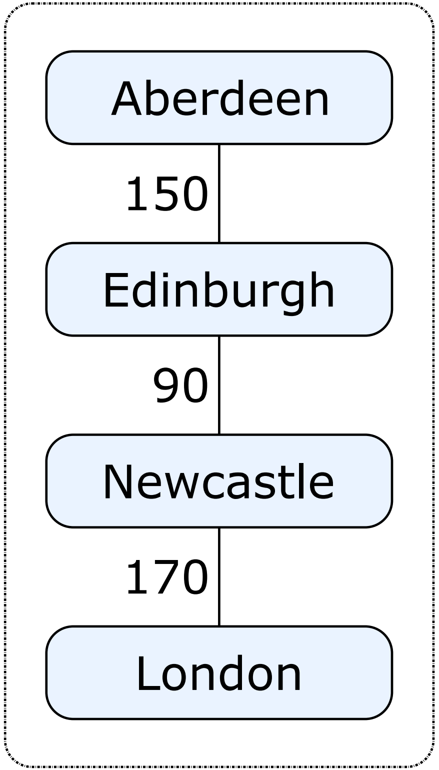


# Example 1: transportation networks

```haskell
type Network e a = Graph (Distance e) a
```

```haskell
type JourneyTime = Int -- In minutes

data City = Aberdeen | Edinburgh | Glasgow
          | London   | Newcastle
```

*EastCoast network*



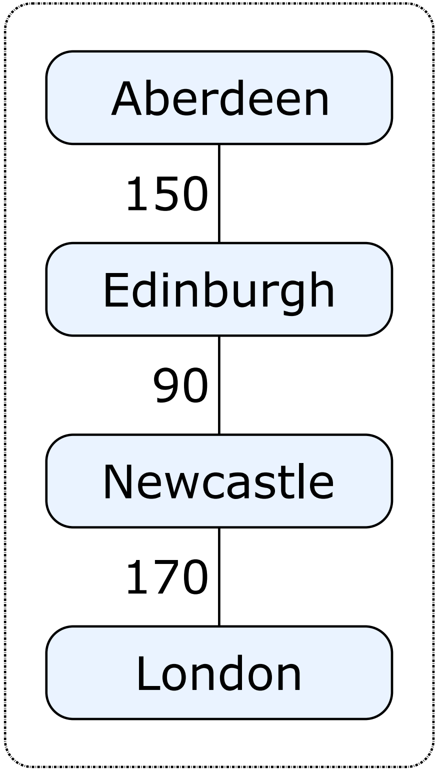
# Example 1: transportation networks

```haskell
type Network e a = Graph (Distance e) a
```

```haskell
type JourneyTime = Int -- In minutes

data City = Aberdeen | Edinburgh | Glasgow
          | London   | Newcastle
```

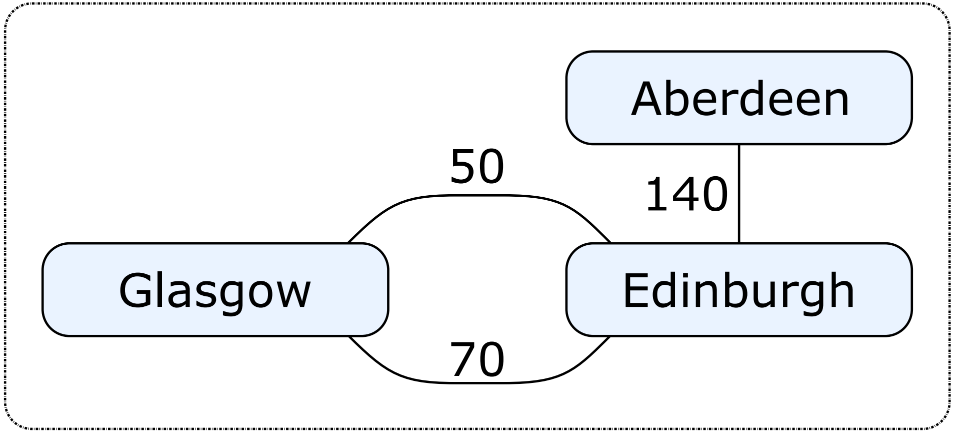
```haskell
eastCoast :: Network JourneyTime City
eastCoast = overlays
    [ Aberdeen  -<150>- Edinburgh
    , Edinburgh -< 90>- Newcastle
    , Newcastle -<170>- London ]
```

*EastCoast network*



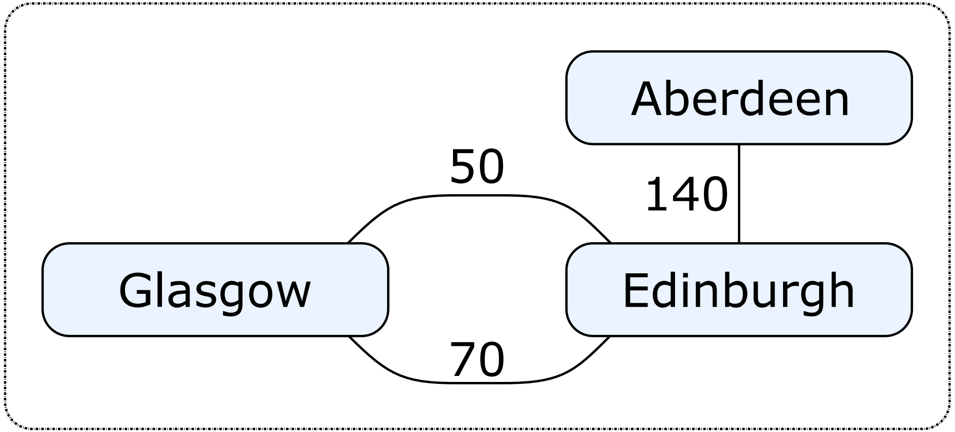
# Example 1: transportation networks

*ScotRail network*



# Example 1: transportation networks

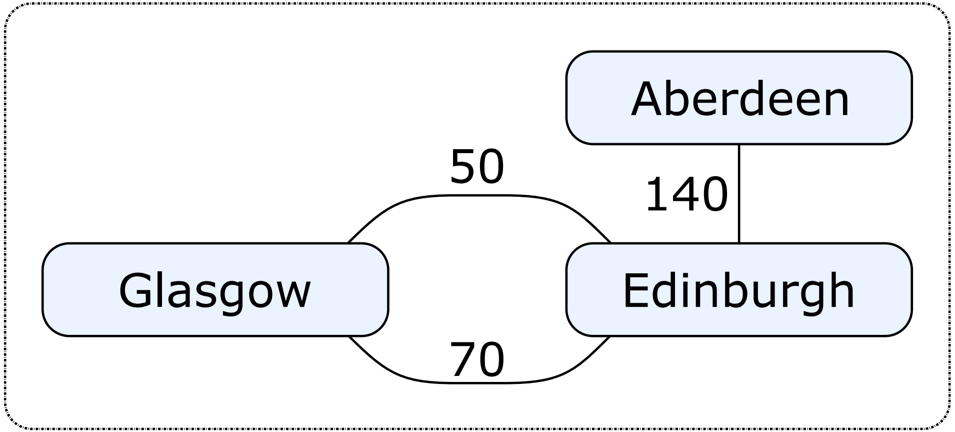
*ScotRail network*




```
scotRail :: Network JourneyTime City
scotRail = overlays
    [ Aberdeen -<140>- Edinburgh
    , Glasgow  -< 50>- Edinburgh
    , Glasgow  -< 70>- Edinburgh ]
```

# Example 1: transportation networks

*ScotRail network*

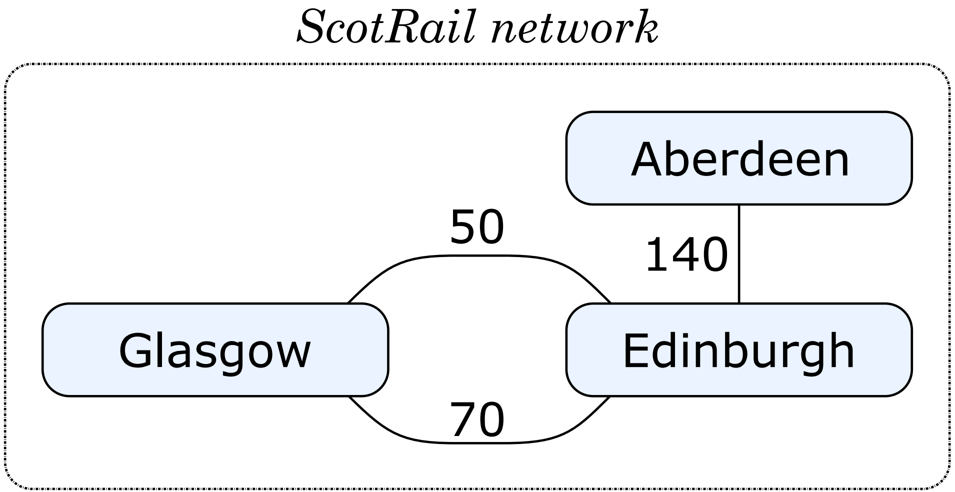


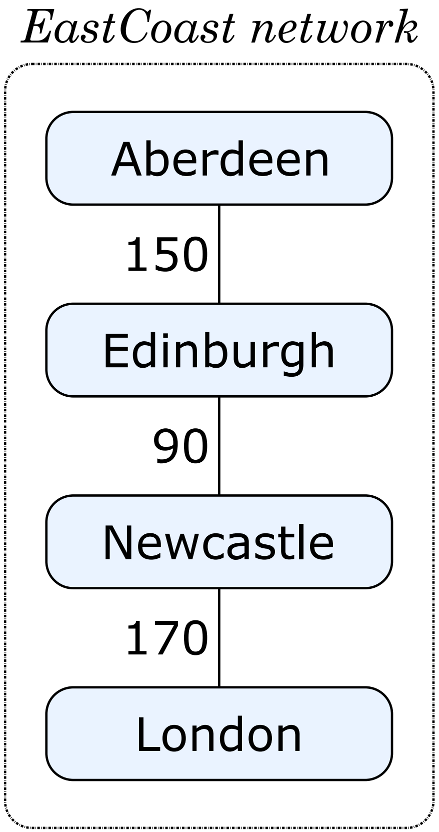

```
scotRail :: Network JourneyTime City
scotRail = overlays
    [ Aberdeen -<140>- Edinburgh
    , Glasgow  -< 50>- Edinburgh
    , Glasgow  -< 70>- Edinburgh ]
```

In the **Distance semiring** we can simplify this network algebraically:

```
x-<50>-y + x-<70>-y
=
x-<min 50 70>-y
=
x-<50>-y
```

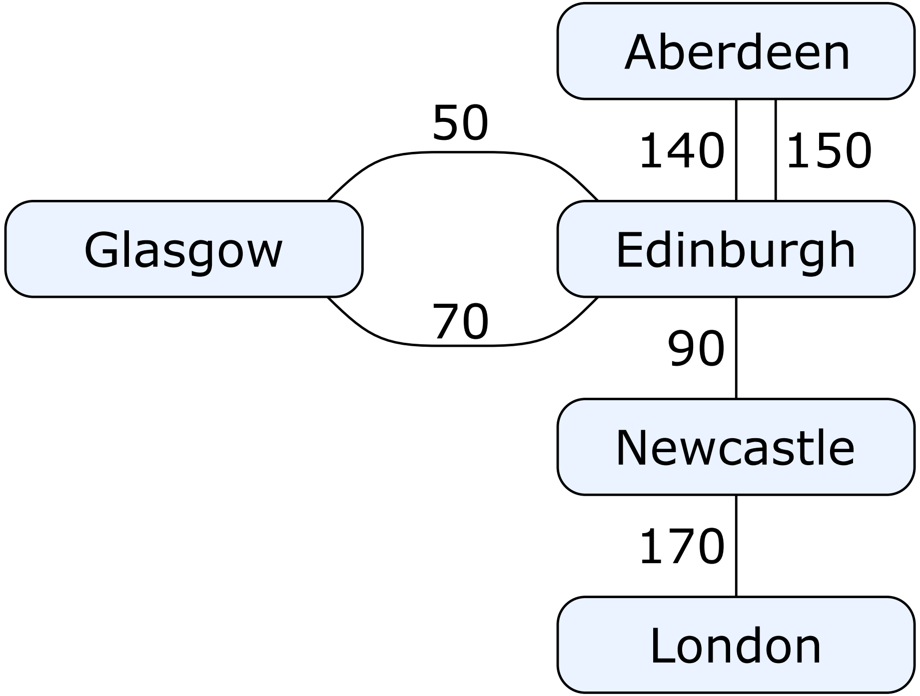
# Example 1: transportation networks



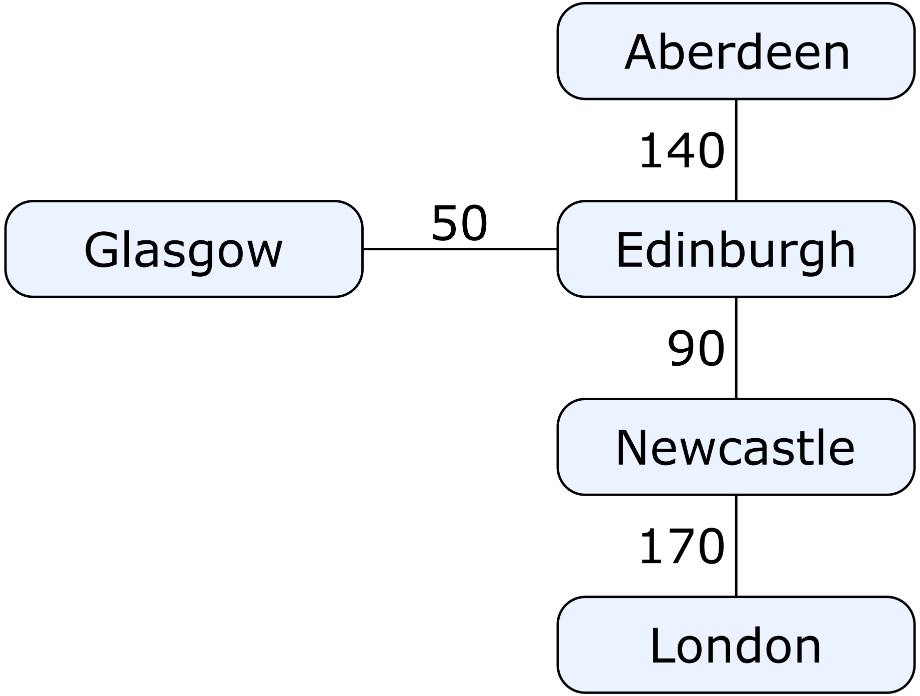

```
network :: Network JourneyTime City
network = overlay scotRail eastCoast
```

# Example 1: transportation networks



# Example 1: transportation networks

Axioms of labelled algebraic graphs

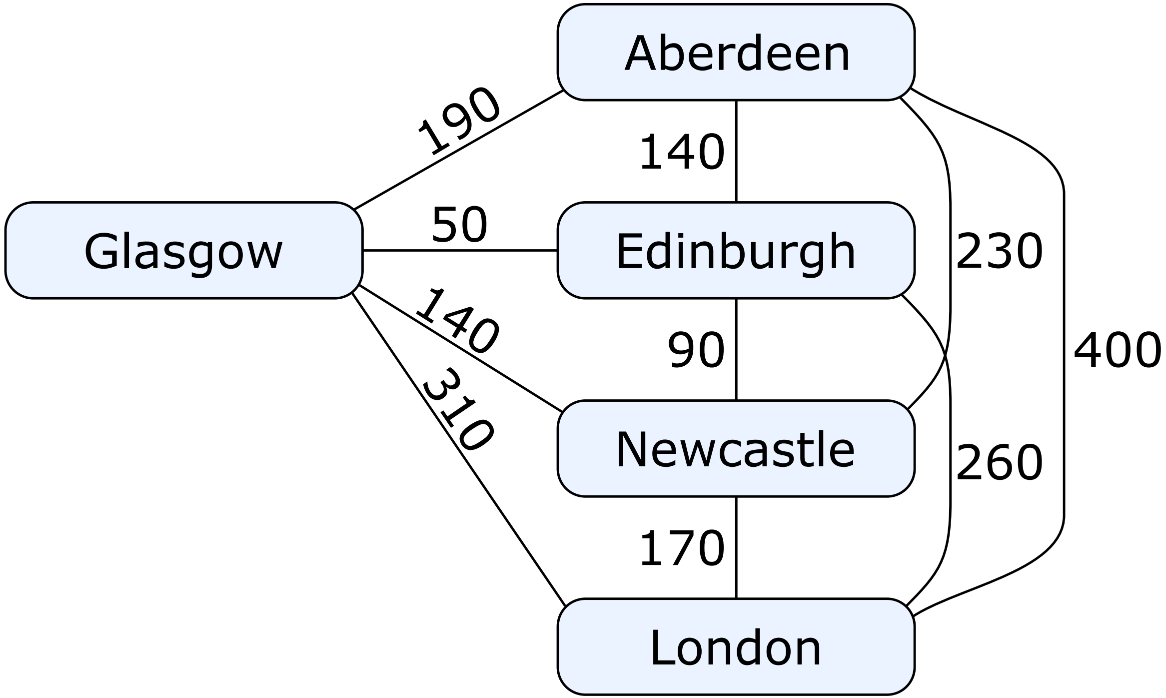


# Example 1: transportation networks

Axioms of labelled algebraic graphs

Add **transitivity**:

```
y≠ε  ⟹
x-<a>-y + y-<b>-z =
x-<a>-y + y-<b>-z +
x-<a+b>-z
```



# Example 1: transportation networks

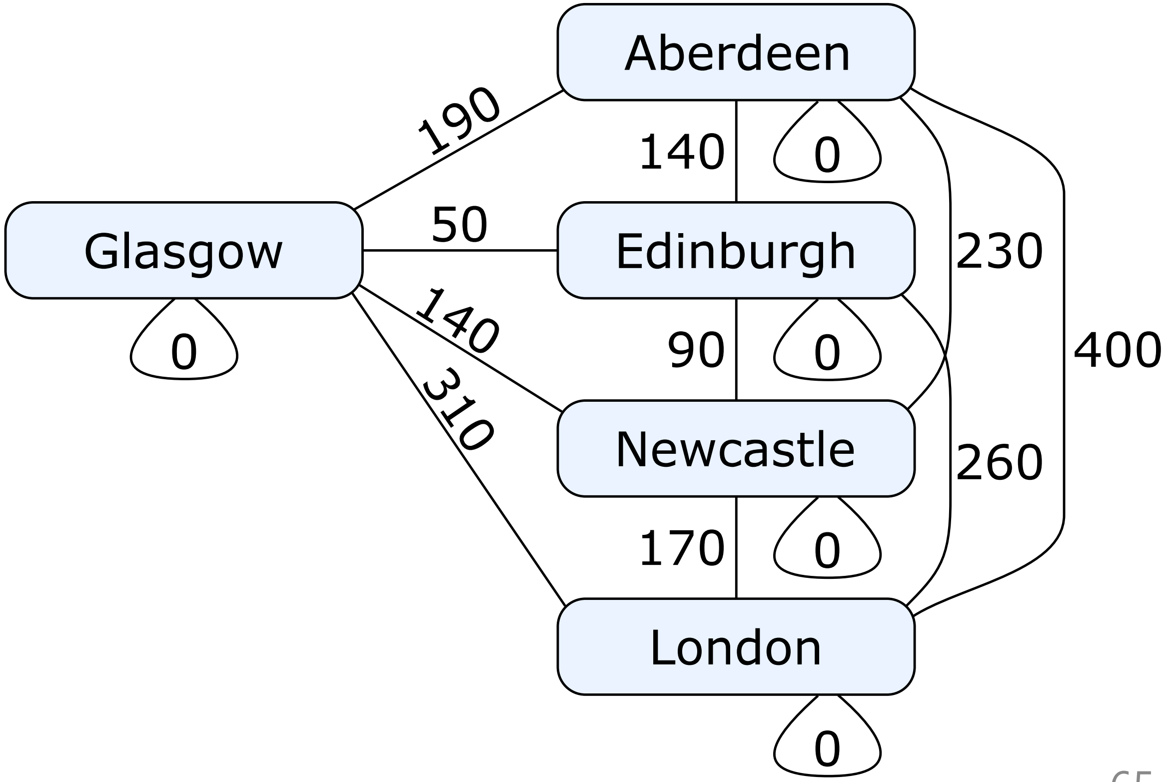
Axioms of labelled algebraic graphs

Add **transitivity**:

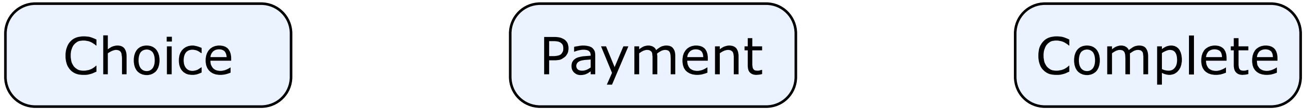
```
y≠ε  ⟹
x-<a>-y + y-<b>-z =
x-<a>-y + y-<b>-z +
x-<a+b>-z
```

Add **reflexivity**:

```
v = v-<0>-v
```

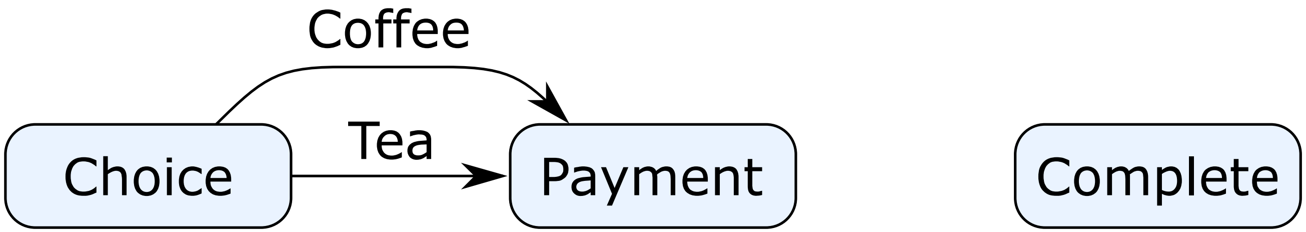


# Example 2: finite automata



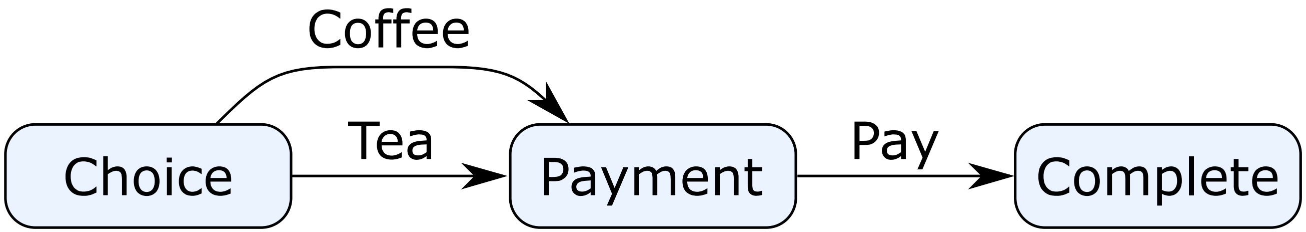

```haskell
type Automaton a s = Graph (RegularExpression a) s

data State = Choice | Payment | Complete
data Alphabet = Coffee | Tea | Cancel | Pay
```

# Example 2: finite automata



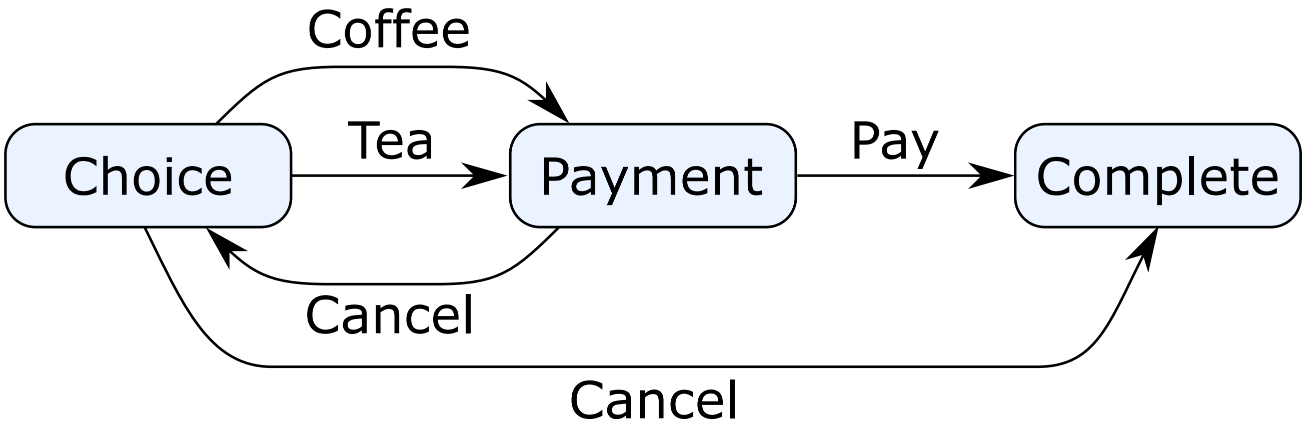

```
automaton = overlays [ Choice  -<[Coffee, Tea]>- Payment
                     , Payment -<[           ]>- Complete

                                                        ]
```

# Example 2: finite automata



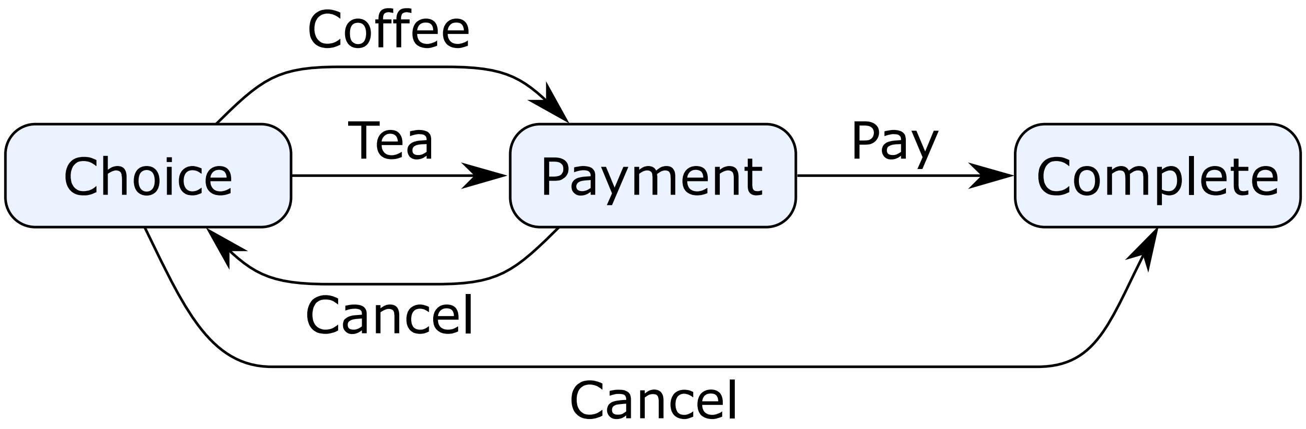

```
automaton = overlays [ Choice  -<[Coffee, Tea]>- Payment
                     , Payment -<[Pay        ]>- Complete

                                                        ]
```

# Example 2: finite automata




```
automaton = overlays [ Choice  -<[Coffee, Tea]>- Payment
                     , Payment -<[Pay         ]>- Complete
                     , Choice  -<[Cancel      ]>- Complete
                     , Payment -<[Cancel      ]>- Choice ]
```

# Example 2: finite automata




After **closure**, we also have the following edges:

– `Payment -<(Cancel;(Coffee | Tea))*>- Payment`

– `Payment -<(Cancel;(Coffee | Tea))*;(Pay | Cancel;Cancel)>- Complete`

# **Part IV:**
# Algebraic Graphs Library

# Algebraic graphs library

Algebraic graphs are available on Hackage

- Graph construction & transformation API
- http://hackage.haskell.org/package/algebraic-graphs
- https://github.com/snowleopard/alga

More theory and examples in Haskell Symposium 2017 paper:

- https://github.com/snowleopard/alga-paper

Parts of the API are formally verified in Agda:

- https://github.com/algebraic-graphs/agda

600+ QuickCheck properties…

# Performance

Google Summer of Code project:

- Student: Alexandre Moine
- https://github.com/haskell-perf/graphs

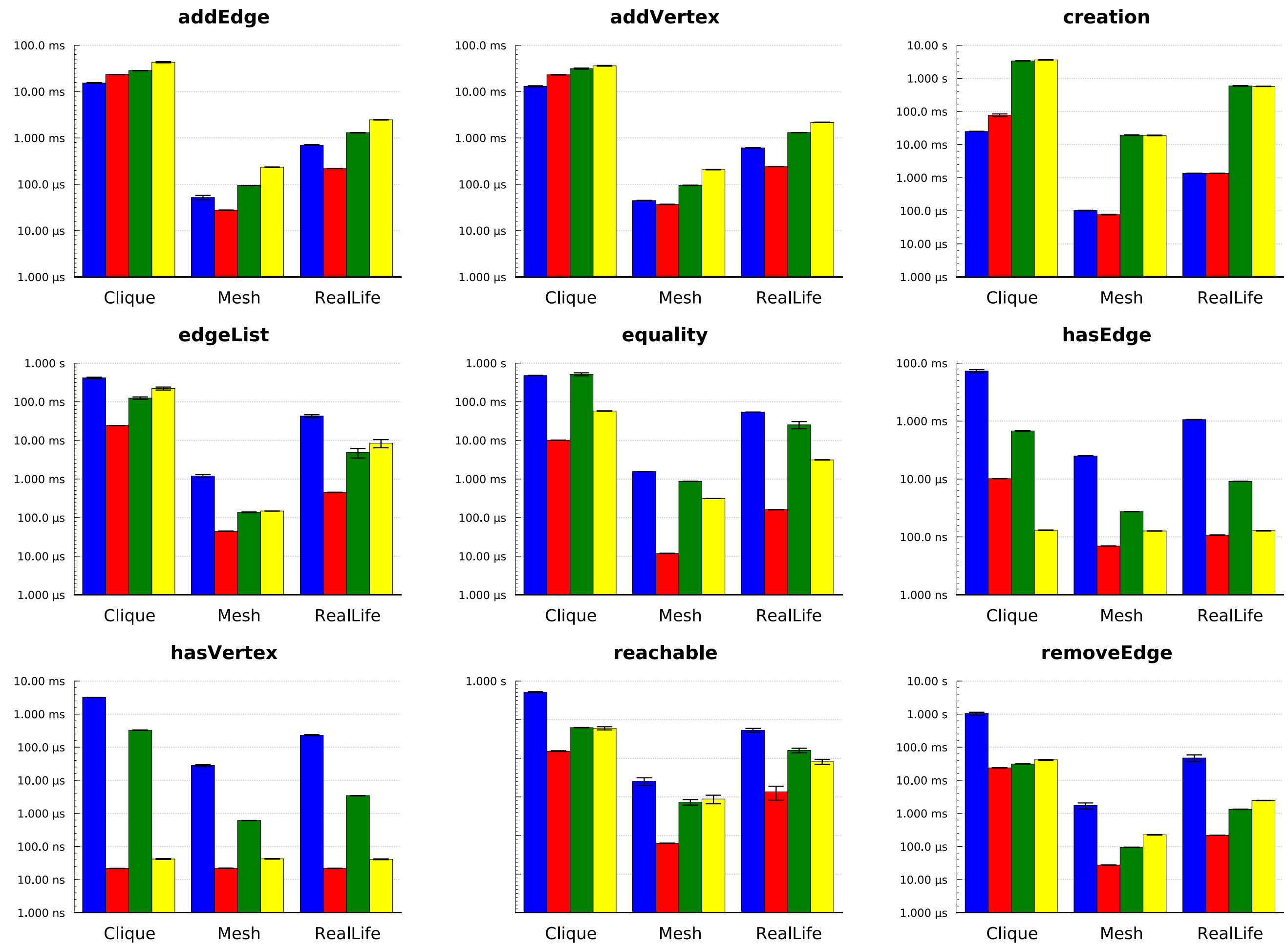
Benchmark suite for **Alga**, **containers**, **fgl**, **Hash-Graph**

Various performance optimisations

- e.g. use rewrite rules to make **transpose . star** as fast as:

```
transposeStar :: a -> [a] -> Graph a
transposeStar x [] = vertex x
transposeStar x ys = connect (vertices ys) (vertex x)
```

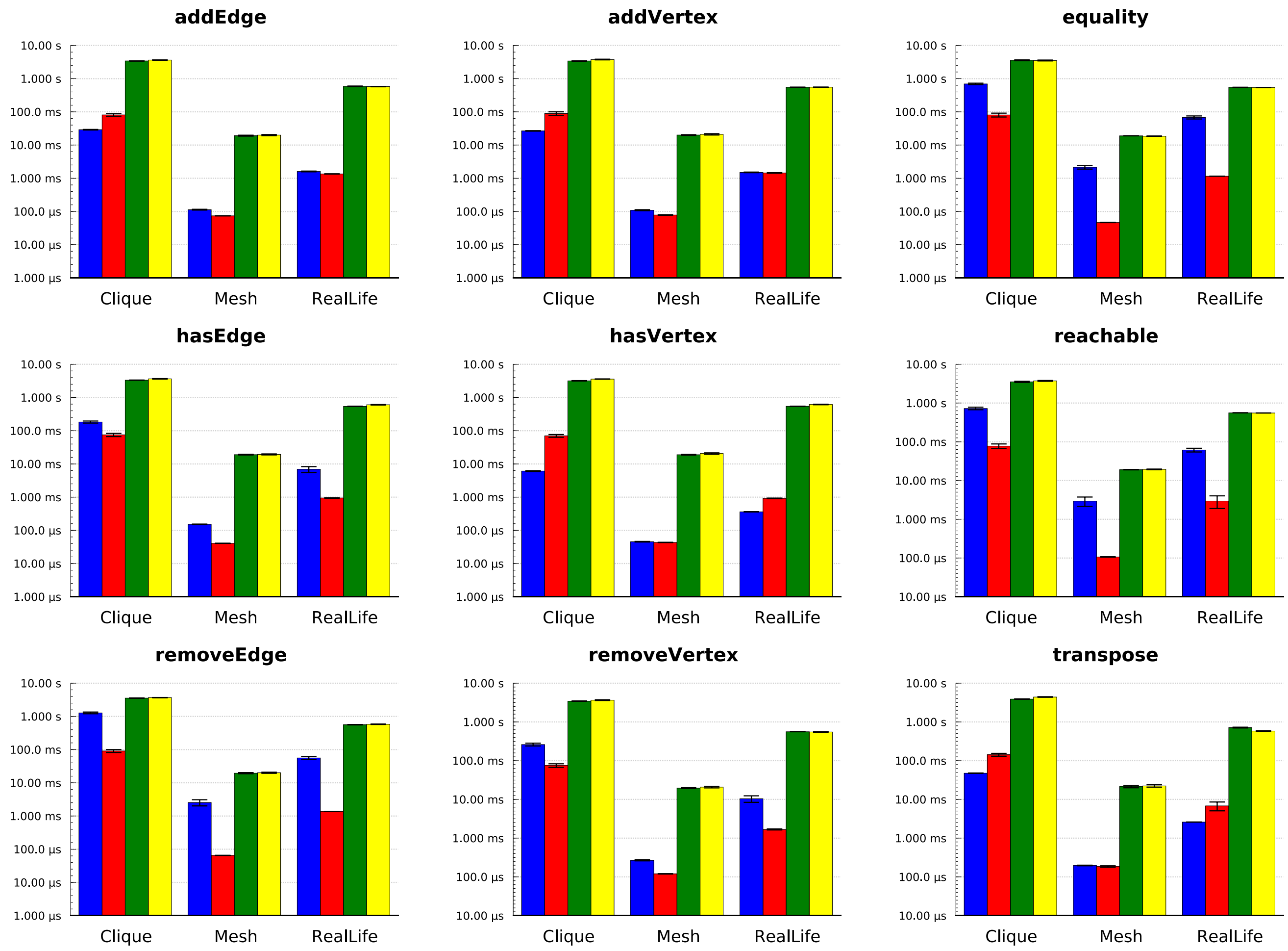
# Performance




Alga Containers Fgl Hash-Graph

Graphs used: Clique with 1000 vertices, Mesh with 1000 vertices

# Performance (fusion)




Alga Containers Fgl Hash-Graph

Graphs used: Clique with 1000 vertices, Mesh with 1000 vertices

# Why not use Alga?

**Alga** is new, experimental and unstable

- Version 0.2 released recently, with many breaking changes
- Every new algorithm is a (cool!) research problem

Why use the **containers** library instead:

- Mature, bundled with GHC
- Performance
- A textbook data structure, no surprises

Why use the **fgl** library instead:

- Mature, comes with a lot of algorithms
- Convenient for expressing many algorithms (DFS, BFS, etc.)

# Thank you!

andrey.mokhov@ncl.ac.uk
@andreymokhov

P.S.: Have you come across decomposition **xyz = xy + xz + yz**?

P.P.S.: Plenty of open research directions: graph algorithms, compact graph representation, links to topology, etc. Help me!

# A library for algebraic graphs in just 100 lines of code

# Reusing functional programming abstractions

```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)

instance Eq a  => Eq  (Graph a) -- via normal form
instance Num a => Num (Graph a)
instance Functor       Graph
instance Applicative   Graph
instance Monad         Graph
instance MonadPlus     Graph
...
```

# Reusing functional programming abstractions

```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)

instance Eq a  => Eq  (Graph a) -- via normal form
instance Num a => Num (Graph a)
instance Functor          Graph
instance Applicative      Graph
instance Monad            Graph
instance MonadPlus        Graph
...
```

Correspond to basic graph transformations: merging, splitting, removing vertices, etc.

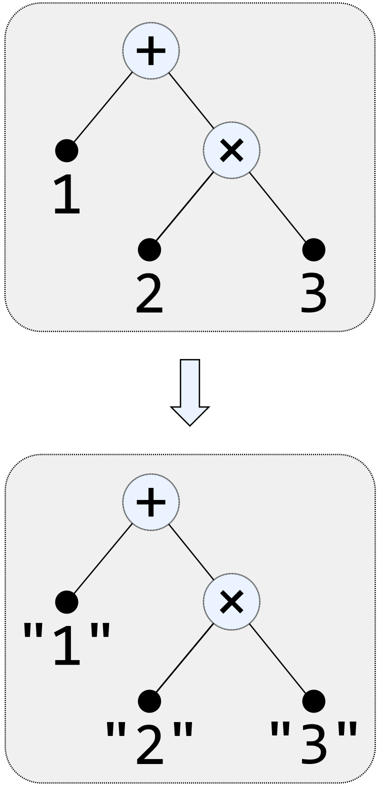
# Graph as a Num

```
instance Num a => Num (Graph a) where
    fromInteger = Vertex . fromInteger
    (+)         = Overlay
    (*)         = Connect
    signum      = const Empty
    abs         = id
    negate      = id
```

```
example :: Graph Int
example = 1 * (2 + 3)
-- Instead of: Graph [1,2,3] [(1,2), (1,3)]
```

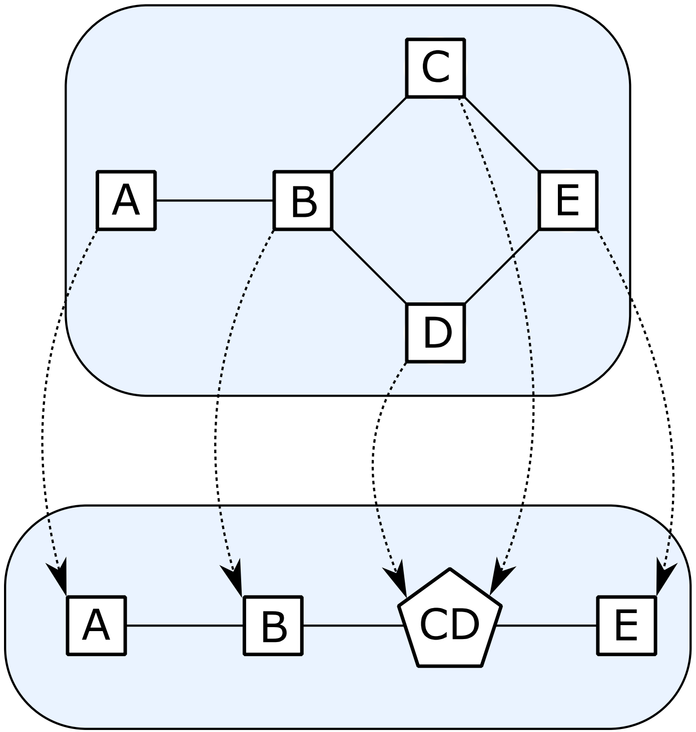
# Graph as a Functor

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b

-- Lists
fmap (+1) [1, 2, 3] == [2, 3, 4]
fmap show [1, 2, 3] == ["1", "2", "3"]

-- Graphs
fmap (+1) (1 + 2 * 3) == 2 + 3 * 4
fmap show (1 + 2 * 3) == "1" + "2" * "3"
```

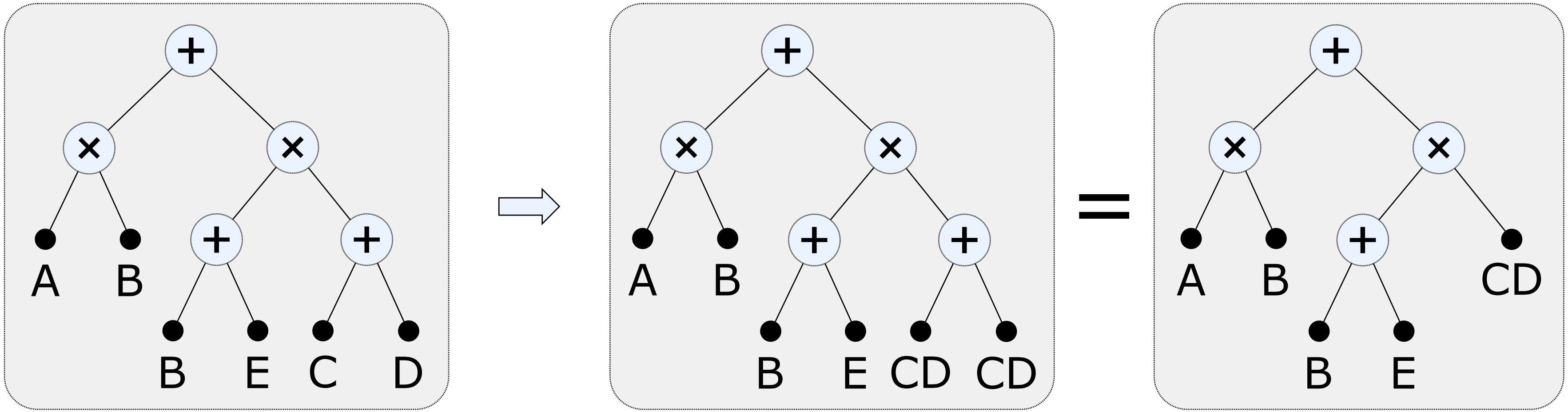


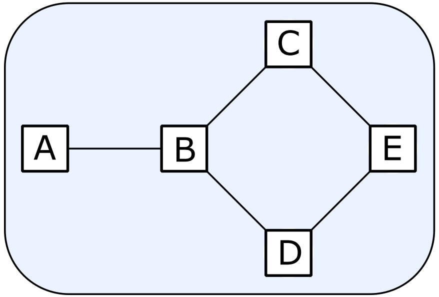
# Merge vertices using Functor

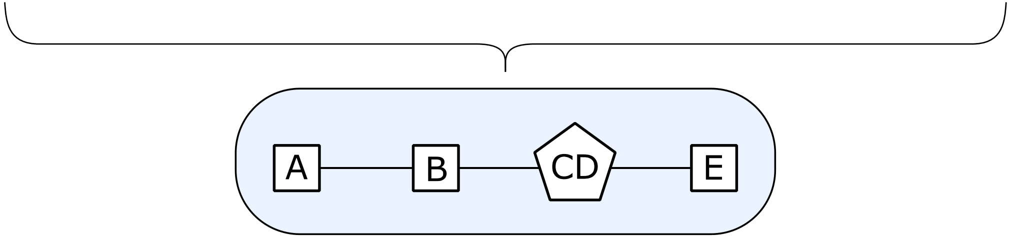
```
mergeCD :: Graph String
        -> Graph String
mergeCD g = fmap f g
  where
    f "C" = "CD"
    f "D" = "CD"
    f x   = x
```



# Merge vertices using Functor

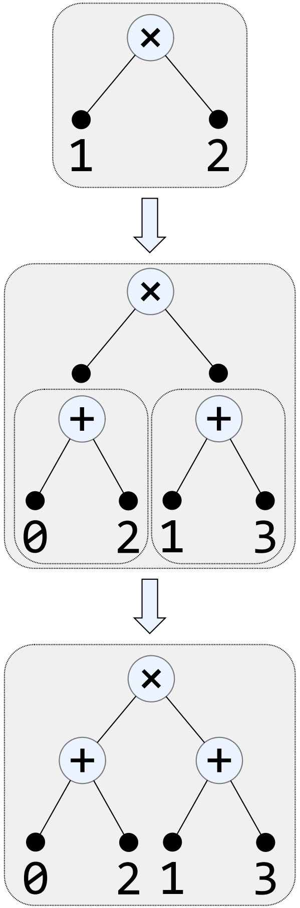


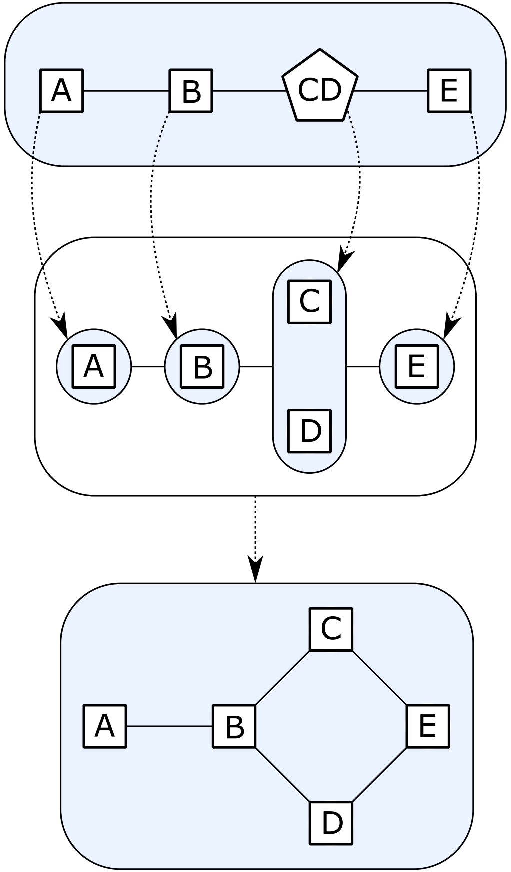
# Graph as a Monad

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

-- Lists
neighbours x = [x - 1, x + 1]
fmap neighbours [1, 2] == [[0, 2], [1, 3]]
[1, 2] >>= neighbours == [0, 2, 1, 3]

-- Graphs
neighbours x = Vertex (x - 1) + Vertex (x + 1)
(1 * 2) >>= neighbours == (0 + 2) * (1 + 3)
```

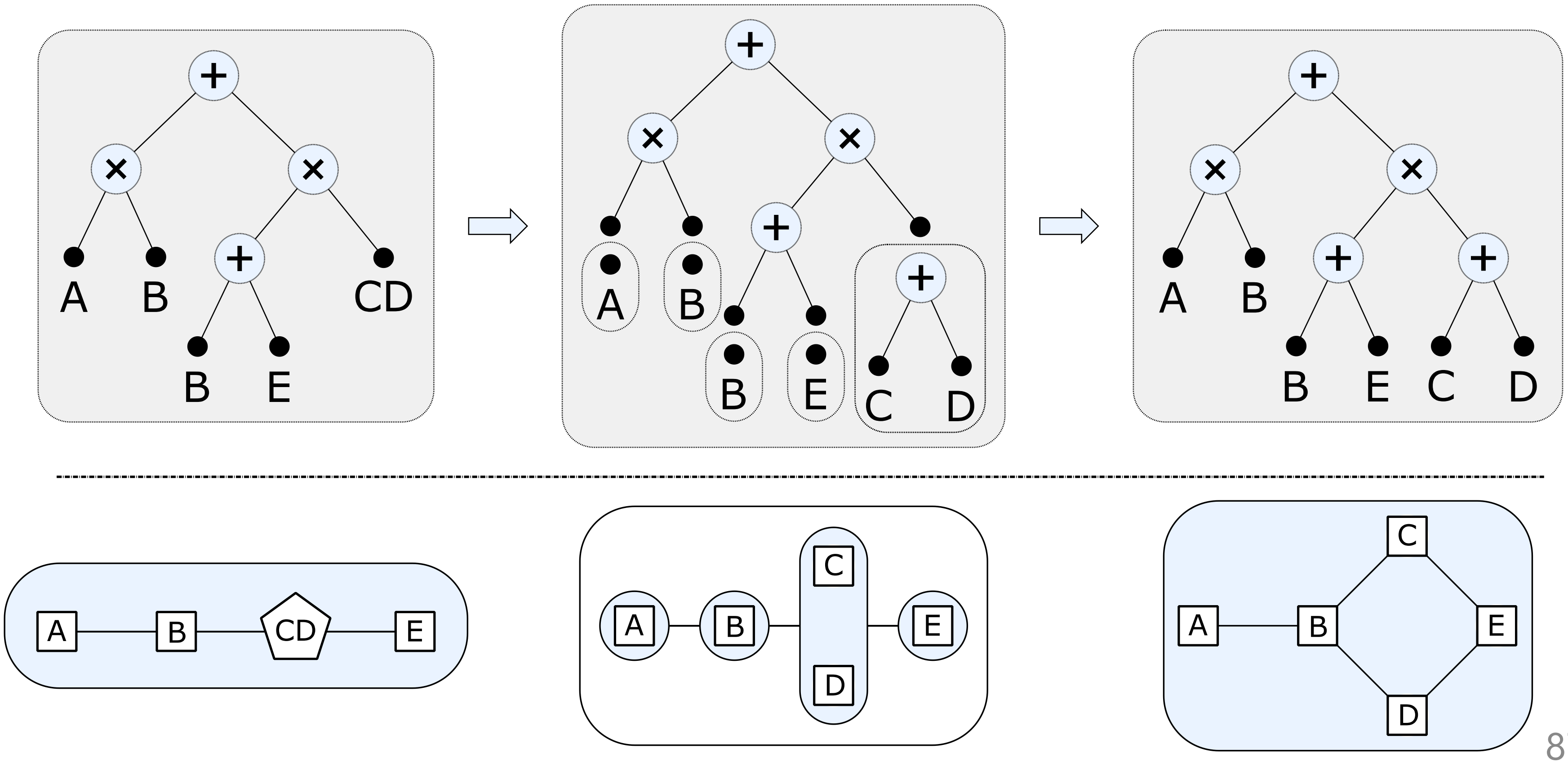


# Split vertices using Monad

```
splitCD :: Graph String
        -> Graph String
splitCD g = g >>= f
  where
    f "CD" = Vertex "C"
           + Vertex "D"
    f x    = Vertex x
```



# Split vertices using Monad

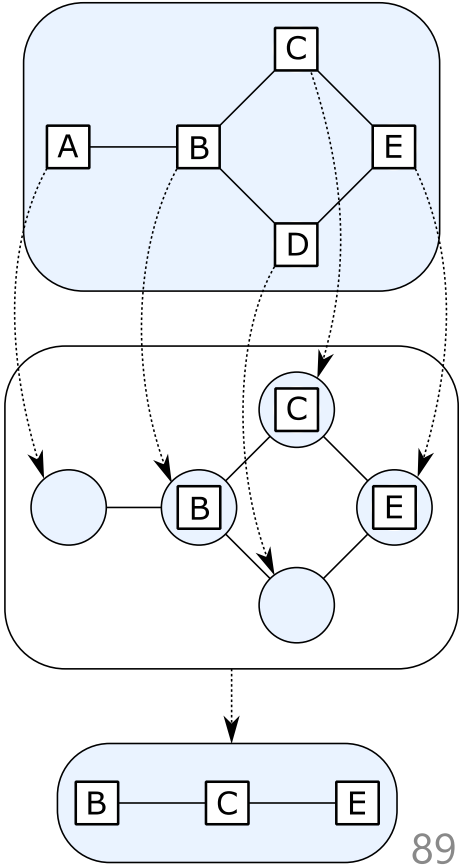


# Graph as a MonadPlus

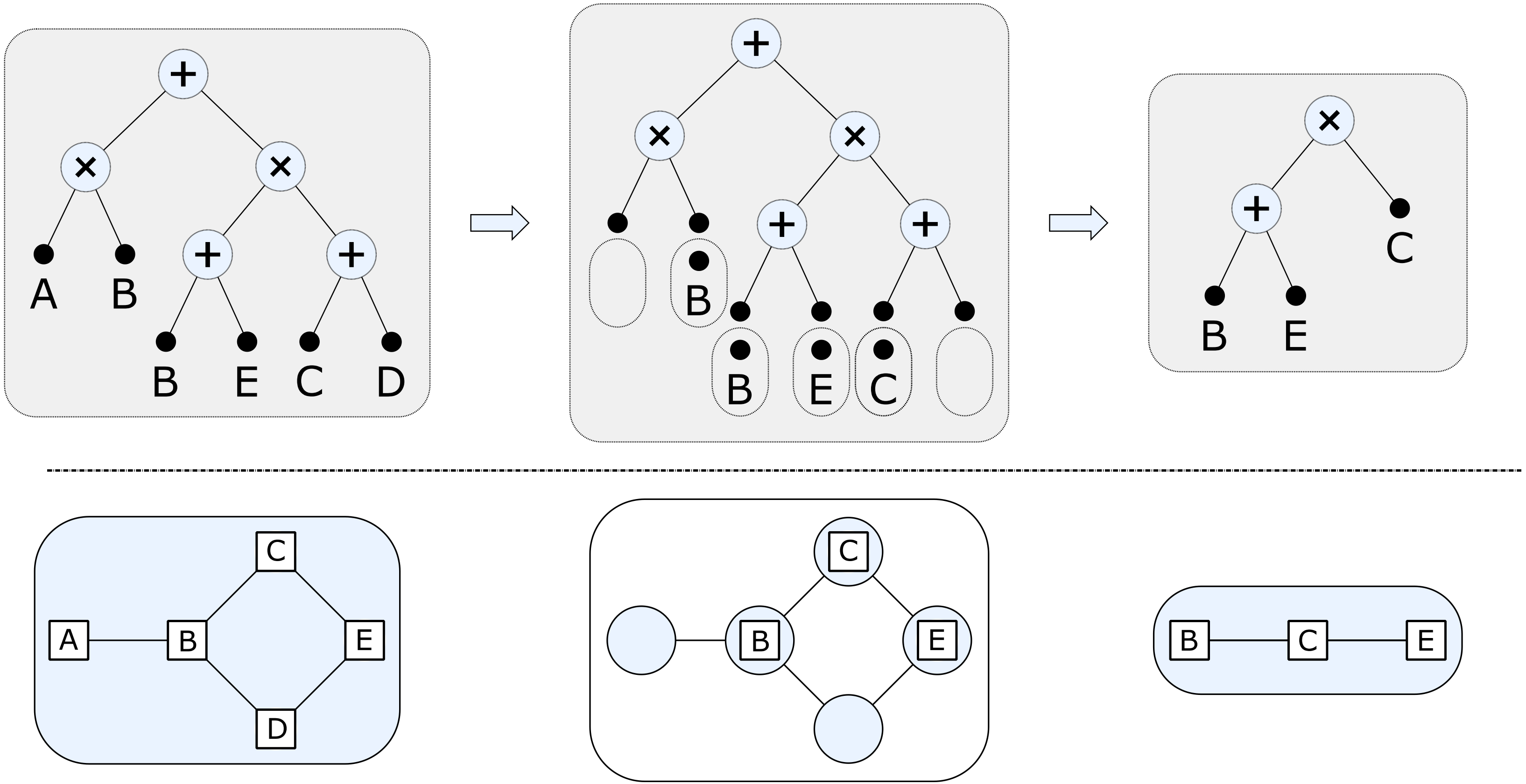
```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a

-- Lists
mzero == []
mplus [1, 2] [2, 3] == [1, 2] ++ [2, 3] == [1, 2, 2, 3]

-- Graphs
mzero == Empty
mplus (1 + 2) (2 * 3) == (1 + 2) + (2 * 3) == 1 + 2 * 3
```

# Find induced subgraphs using MonadPlus

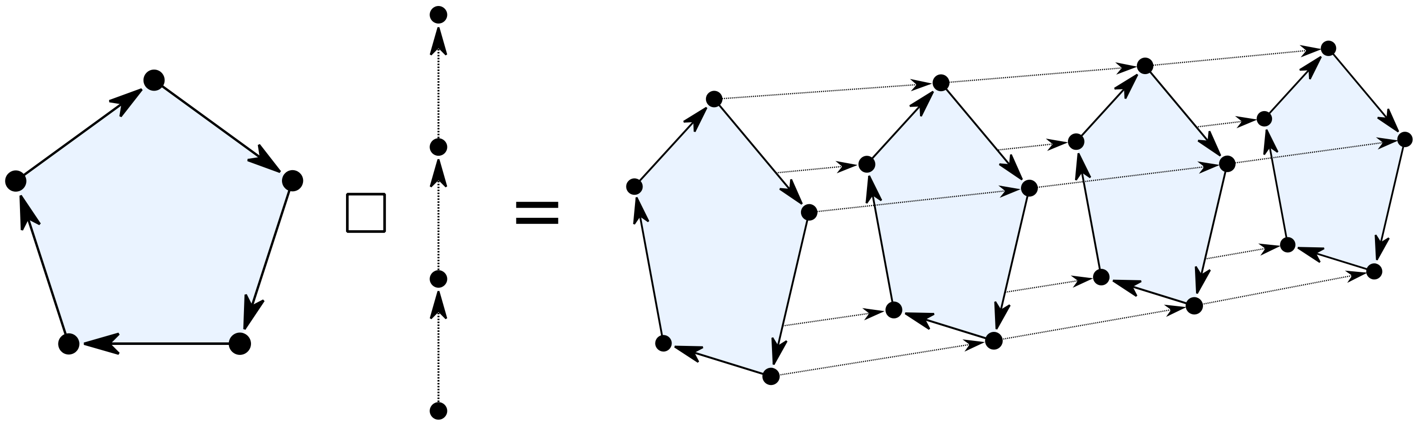
```haskell
induceBCE :: Graph String -> Graph String
induceBCE = mfilter (`elem` ["B","C","E"])

-- From Control.Monad:
mfilter :: MonadPlus m
        => (a -> Bool) -> m a -> m a
mfilter p ma = do
    a <- ma
    if p a then return a else mzero
```



# Find induced subgraphs using MonadPlus



# Cartesian graph product




```
box :: Graph a -> Graph b -> Graph (a, b)
box x y = msum $ xs ++ ys
  where
    xs = map (\b -> fmap (,b) x) $ toList y
    ys = map (\a -> fmap (a,) y) $ toList x
```

# From four primitives to a library

```haskell
-- An abstract interface or a type class
empty   :: Graph a
vertex  :: a -> Graph a
overlay :: Graph a -> Graph a -> Graph a
connect :: Graph a -> Graph a -> Graph a
```

# From four primitives to a library

```
-- An abstract interface or a type class
empty   :: Graph a
vertex  :: a -> Graph a
overlay :: Graph a -> Graph a -> Graph a
connect :: Graph a -> Graph a -> Graph a
```

```
-- Combine primitives into larger graphs
vertices :: [a] -> Graph a
vertices vs = foldr overlay empty (map vertex vs)

clique :: [a] -> Graph a
clique vs = foldr connect empty (map vertex vs)
```

# From four primitives to a library

```
edge :: a -> a -> Graph a
edge u v = ???


star :: a -> [a] -> Graph a
star u vs = ???
```

# From four primitives to a library

```
edge :: a -> a -> Graph a
edge u v = connect (vertex u) (vertex v)

star :: a -> [a] -> Graph a
star u vs = ???
```

# From four primitives to a library

```
edge :: a -> a -> Graph a
edge u v = connect (vertex u) (vertex v)

star :: a -> [a] -> Graph a
star u vs = connect (vertex u) (vertices vs)
```

# From four primitives to a library

```
edge :: a -> a -> Graph a
edge u v = connect (vertex u) (vertex v)

star :: a -> [a] -> Graph a
star u vs = connect (vertex u) (vertices vs)
```

```
isSubgraphOf g h = overlay g h == h
hasEdge u v g    = ???
```

# From four primitives to a library

```
edge :: a -> a -> Graph a
edge u v = connect (vertex u) (vertex v)

star :: a -> [a] -> Graph a
star u vs = connect (vertex u) (vertices vs)
```

```
isSubgraphOf g h = overlay g h == h
hasEdge u v g    = edge u v `isSubgraphOf` g
```

# From four primitives to a library

```
edge :: a -> a -> Graph a
edge u v = connect (vertex u) (vertex v)

star :: a -> [a] -> Graph a
star u vs = connect (vertex u) (vertices vs)
```

```
isSubgraphOf g h = overlay g h == h
hasEdge u v g     = edge u v `isSubgraphOf` h
  where
    h = mfilter (`elem` [u,v]) g
```

# Folding algebraic graphs

```
-- Like foldr but for graphs
foldg :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b)
      -> Graph a -> b
foldg e v o c = go
  where
    go Empty            = e
    go (Vertex  x  )    = v x
    go (Overlay x y)    = o (go x) (go y)
    go (Connect x y)    = c (go x) (go y)
```

# Folding algebraic graphs

```haskell
-- Like foldr but for graphs
foldg :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b)
      -> Graph a -> b
foldg e v o c = go
  where
    go Empty            = e
    go (Vertex  x   ) = v x
    go (Overlay x y) = o (go x) (go y)
    go (Connect x y) = c (go x) (go y)
```

```haskell
isEmpty :: Graph a -> Bool
isEmpty = foldg True (const False) (&&) (&&)
```

# Folding algebraic graphs

```
-- Like foldr but for graphs
foldg :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b)
      -> Graph a -> b
foldg e v o c = go
  where
    go Empty          = e
    go (Vertex  x   ) = v x
    go (Overlay x y) = o (go x) (go y)
    go (Connect x y) = c (go x) (go y)
```

The arguments (**e**, **v**, **o**, **c**) must satisfy the laws of the algebra

```
isEmpty :: Graph a -> Bool
isEmpty = foldg True (const False) (&&) (&&)
```

# Folding algebraic graphs

```haskell
hasVertex :: Eq a => a -> Graph a -> Bool
hasVertex x = foldg False (==x) (||) (||)

vertexSet :: Ord a => Graph a -> Set a
vertexSet = foldg Set.empty singleton union union

transpose :: Graph a -> Graph a
transpose = foldg empty vertex overlay (flip connect)

size :: Graph a -> Int
size = foldg 1 (const 1) (+) (+)
```

# Folding algebraic graphs

```
hasVertex :: Eq a => a -> Graph a -> Bool
hasVertex x = foldg False (==x) (||) (||)

vertexSet :: Ord a => Graph a -> Set a
vertexSet = foldg Set.empty singleton union union

transpose :: Graph a -> Graph a
transpose = foldg empty vertex overlay (flip connect)

size :: Graph a -> Int
size = foldg 1 (const 1) (+) (+)
```

Breaks laws:
**size(x) ≠ size(x+ε)**