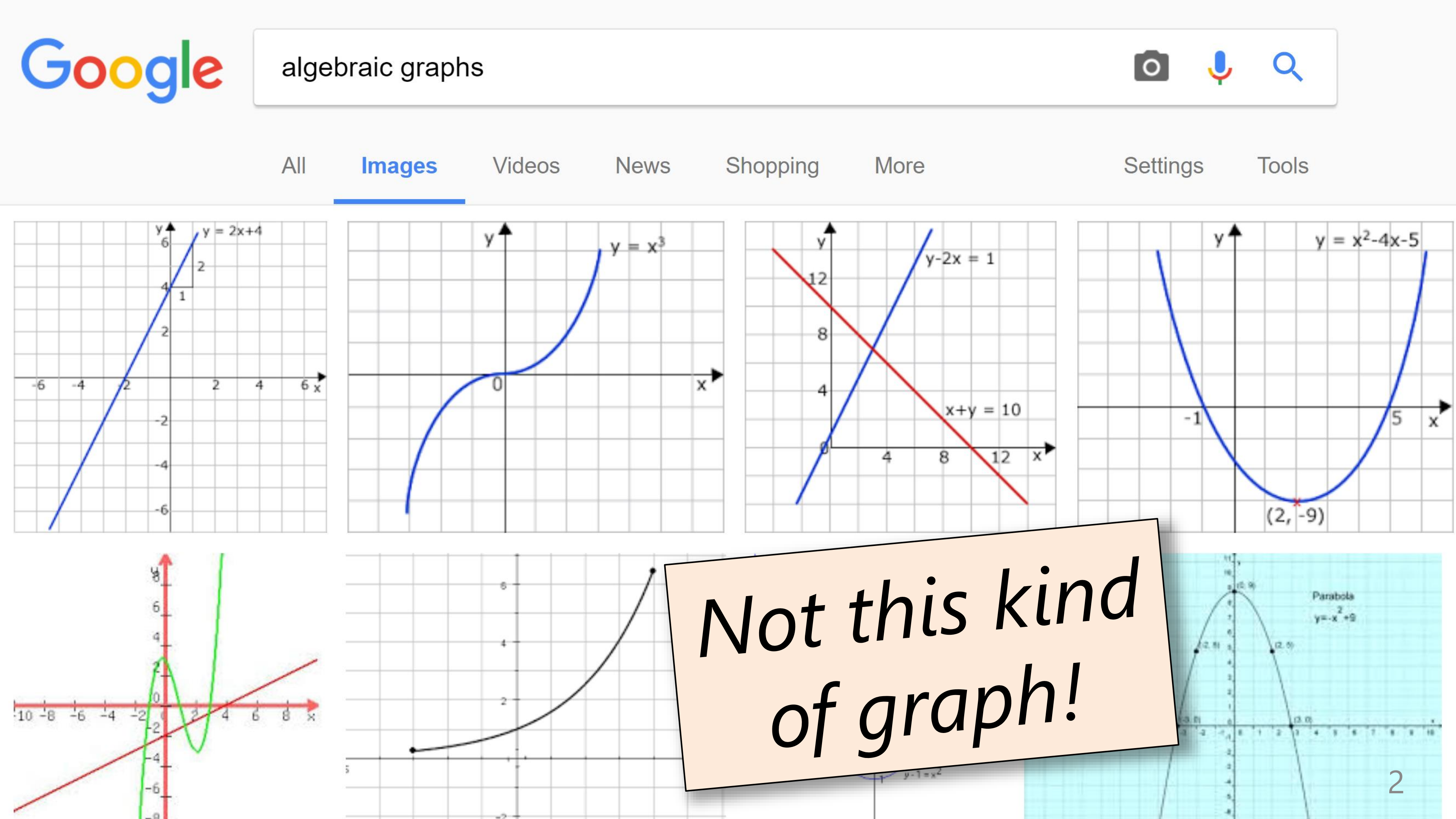
Algebraic Graphs

Andrey Mokhov
GitHub: @snowleopard, Twitter: @andreymokhov

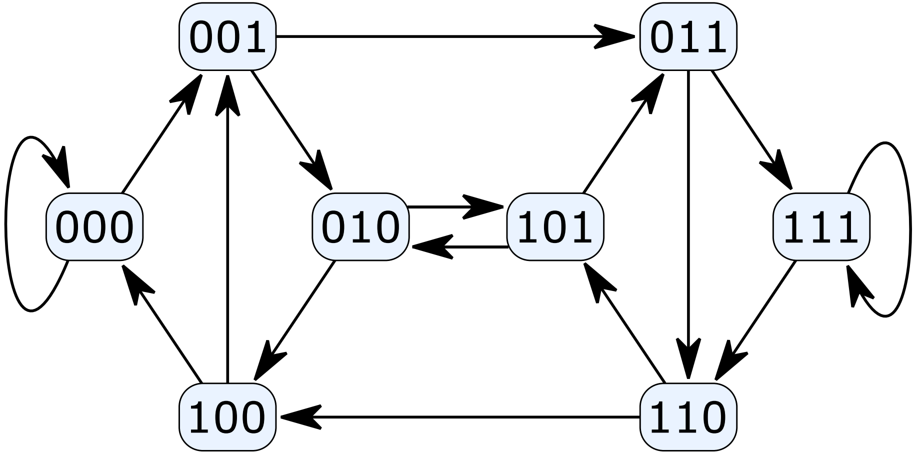
Haskell eXchange, London, October 2017

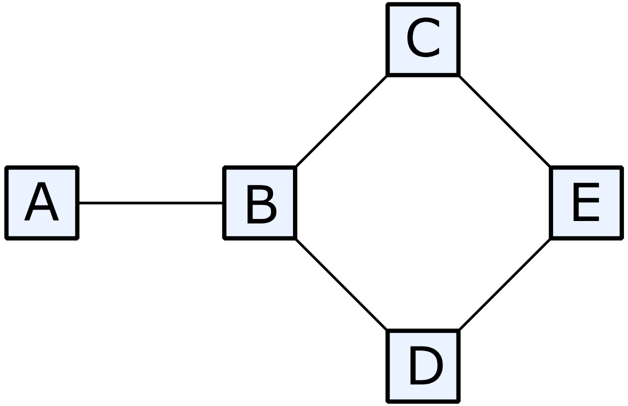

Google
algebraic graphs
All
Images
Videos
News
Shopping
More
Settings
Tools
y = 2x+4
y = x³
y-2x = 1
x+y = 10
y = x²-4x-5
(2, -9)
Parabola
Not this kind of graph!

This kind of graph:

- Labelled vertices
- Can have cycles
- Can have self-loops
- Directed or undirected

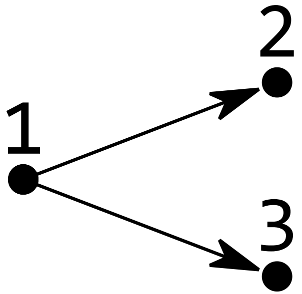
- No edge labels
- No vertex ports
- No ‘forbidden’ edges





From math to Haskell

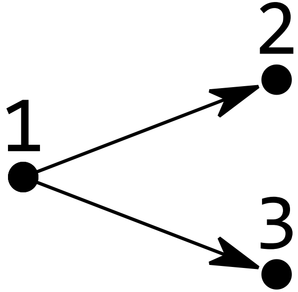
- Pair (V, E) such that $E \subseteq V \times V$
  - Example: $(\{1,2,3\}, \{(1,2), (1,3)\})$



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 :: [a]
     , edges    :: [(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 :: [a]
    , edges    :: [(a,a)] }

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

Problem

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

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

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

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

Hard to express in types

Problem

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

– Non-example: $(\{1\}, \{(1,2)\})$

```
data Graph a = Graph
     { vertices :: [a]
     , edges    :: [(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 are unrepresentable.

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 are unrepresentable.

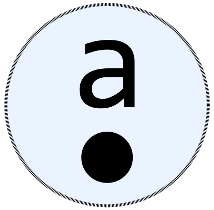
A. Mokhov, V. Khomenko. *"Algebra of Parameterised Graphs"*, ACM Transactions on Embedded Computing Systems, 2014

Empty :: Graph a

Empty :: Graph a

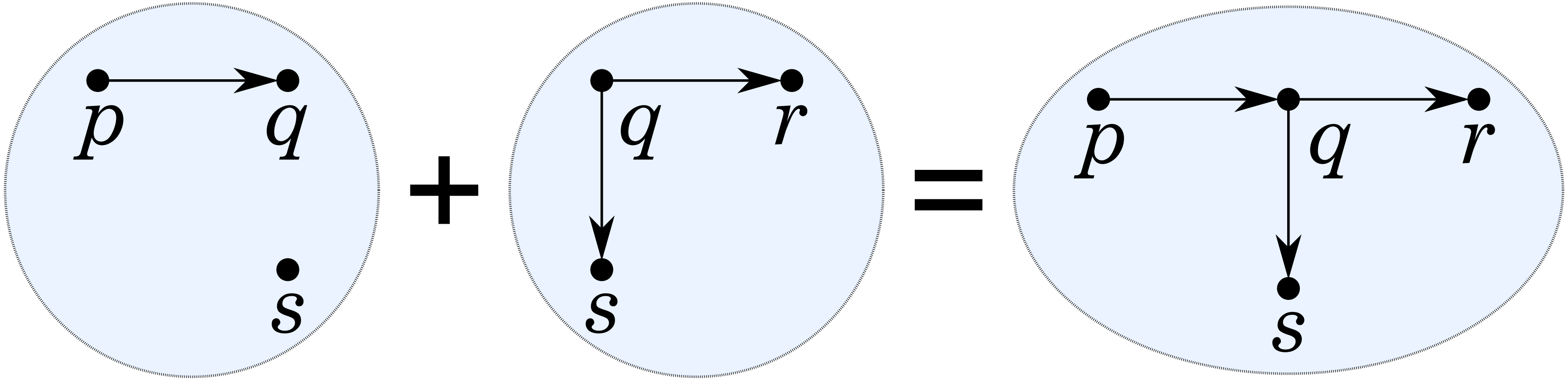
$(\emptyset, \emptyset)$

Vertex :: a -> Graph a



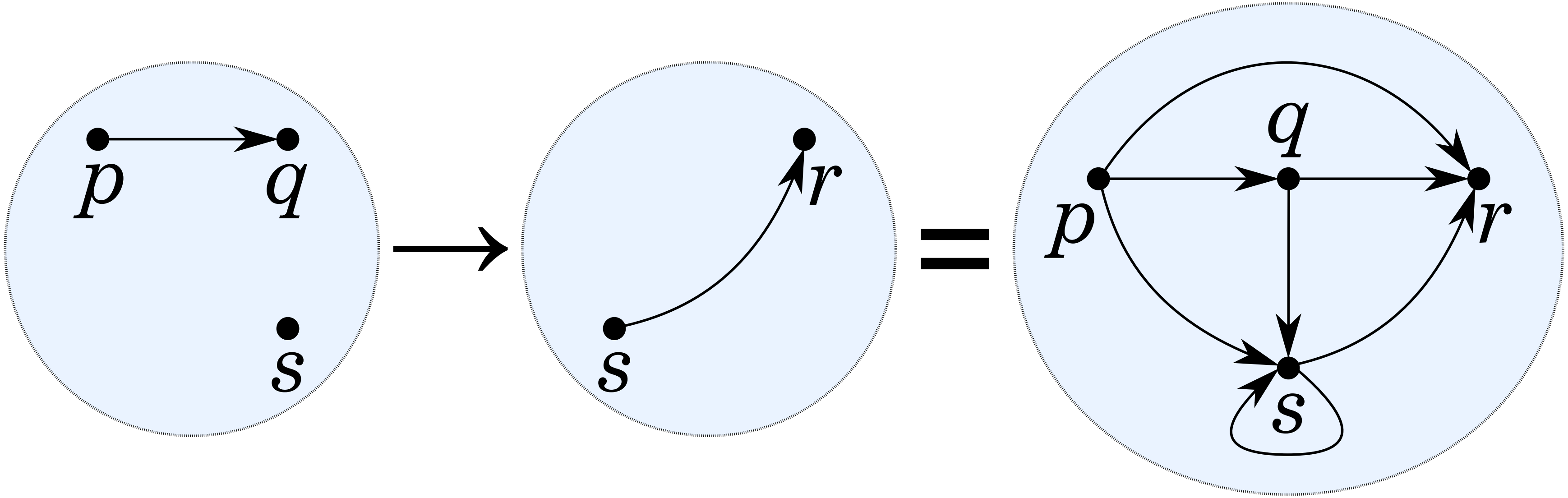
$(\{a\}, \emptyset)$

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



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

Algebraic Graphs

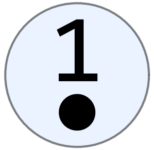
```
data Graph a = Empty
             | Vertex a
             | Overlay (Graph a) (Graph a)
             | Connect (Graph a) (Graph a)
```

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

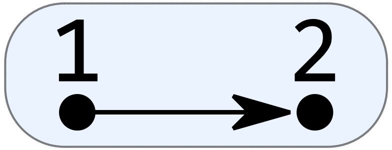



```
Vertex 1
```


```
Vertex 2
```

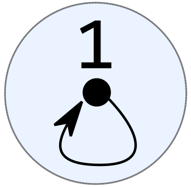

```
Overlay (Vertex 1) (Vertex 2)
```

Or simply $1 + 2$



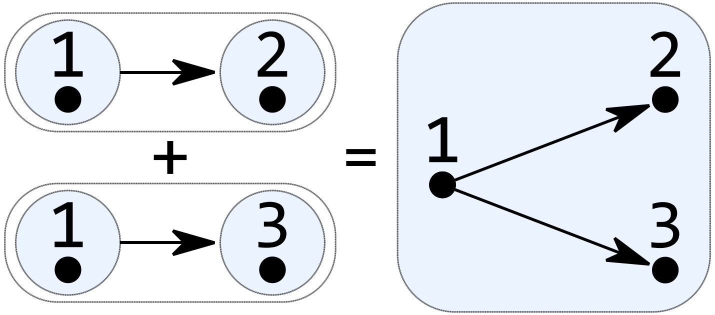

```
Connect (Vertex 1) (Vertex 2)
```

Or simply $1 \rightarrow 2$



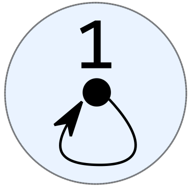
$1 \to 1$

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



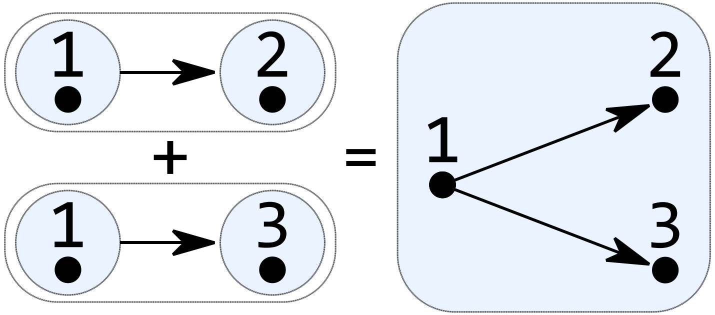
$1 \to 2 + 1 \to 3$

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



$1 \to 1$

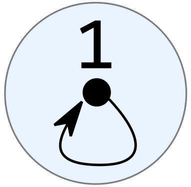
```
Connect (Vertex 1) (Vertex 1)
```



Can we factor out 1?

$1 \to 2 + 1 \to 3$

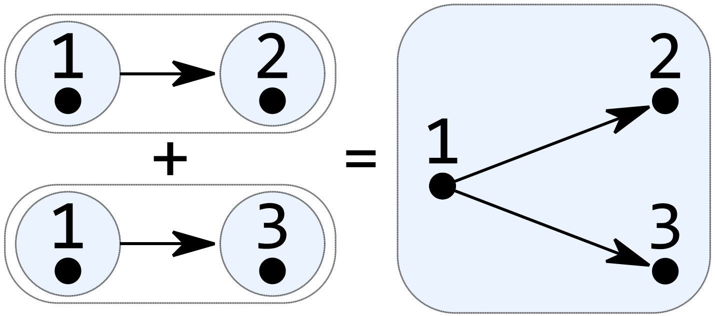
```
Overlay (Connect (Vertex 1) (Vertex 2))
        (Connect (Vertex 1) (Vertex 3))
```



$1 \rightarrow 1$

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

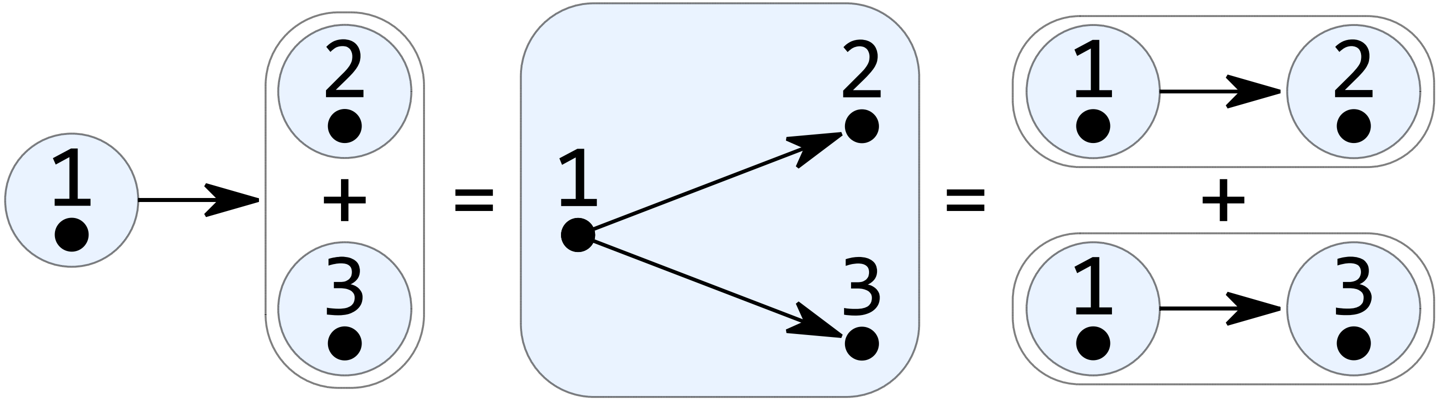
Can we factor out 1? Yes!



$1 \rightarrow 2 + 1 \rightarrow 3$

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

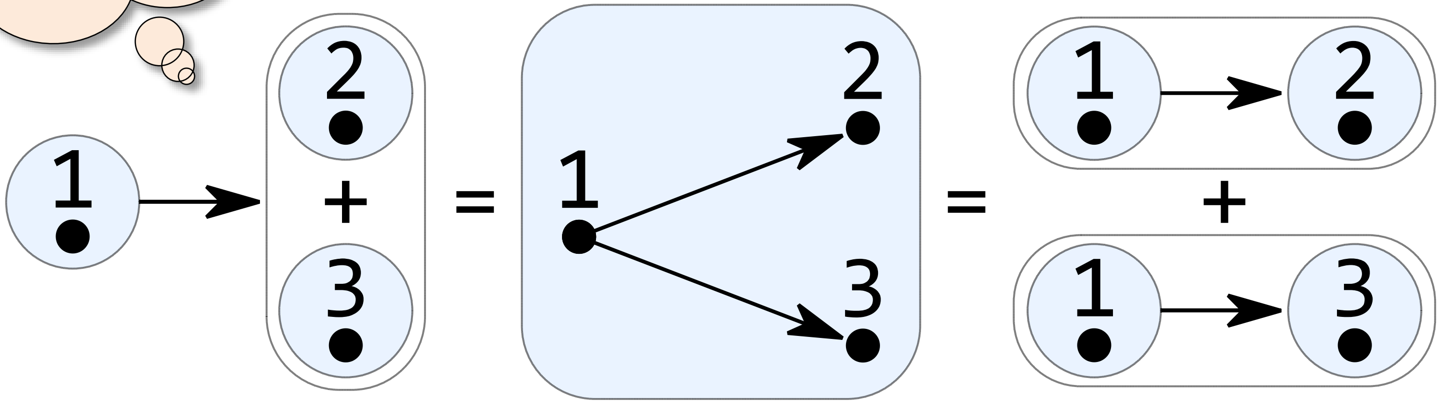
Distributivity



$$x \to (y + z) = x \to y + x \to z$$
$$(x + y) \to z = x \to z + y \to z$$

Distributivity

I bet it's just a semiring…

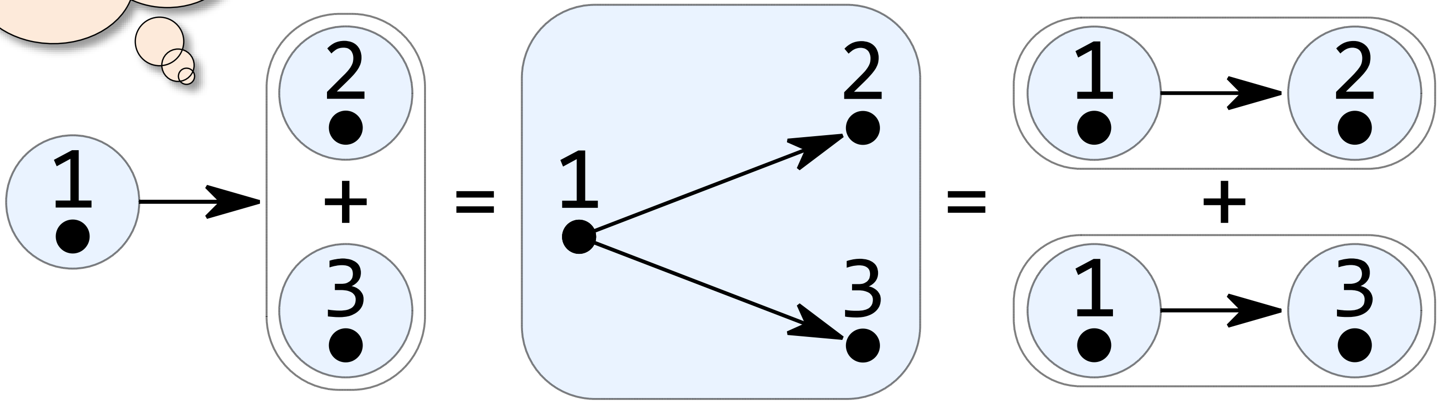


$$x \to (y + z) = x \to y + x \to z$$

$$(x + y) \to z = x \to z + y \to z$$

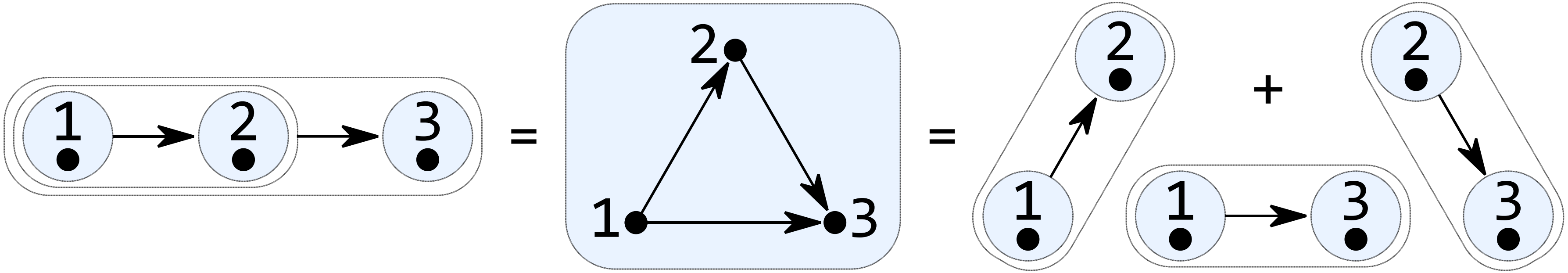
Distributivity

I bet it's just a semiring... **No!**



$$x \to (y + z) = x \to y + x \to z$$
$$(x + y) \to z = x \to z + y \to z$$

Decomposition



$$x \rightarrow y \rightarrow z = x \rightarrow y + x \rightarrow z + y \rightarrow z$$

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

Algebraic structure

Axioms:

Overlay $+$ is commutative and associative

Connect $\to$ is associative

The empty graph ε is the identity of connect $\to$

Connect $\to$ distributes over overlay $+$

Decomposition: $x \to y \to z = x \to y + x \to z + y \to z$

Theorems:

Overlay $+$ is idempotent and has ε as the identity

Other flavours of the algebra

Undirected graphs:

- $x \leftrightarrow y = y \leftrightarrow x$

Reflexive graphs:

- $\text{Vertex } x = \text{Vertex } x \rightarrow \text{Vertex } x$

Transitive graphs:

- $(y \neq \varepsilon) \Longrightarrow x \rightarrow y \rightarrow z = x \rightarrow y + y \rightarrow z$

Various combinations:

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

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     -- pure = Vertex
instance Monad        Graph
instance MonadPlus    Graph     -- mzero = ε, mplus = +
...
```

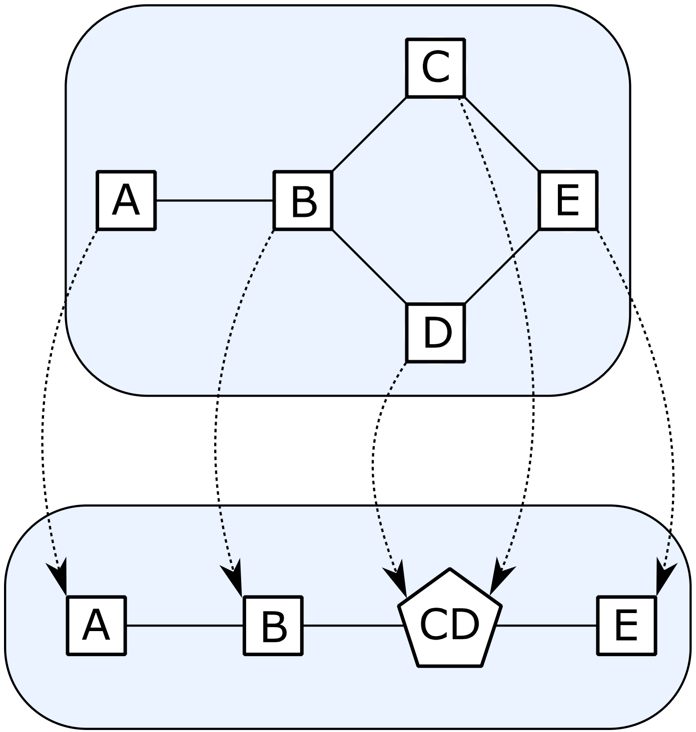
Terse graph notation using Num

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

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

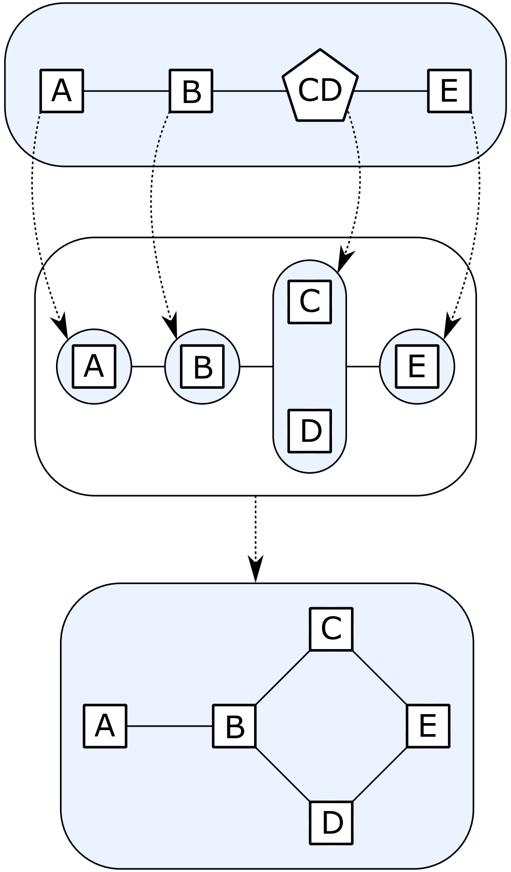
Merge vertices using Functor

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



Split vertices using Monad

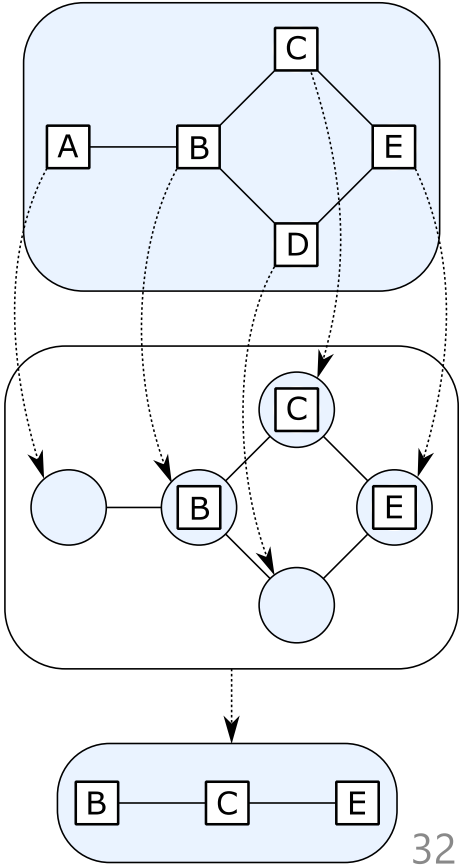
```
splitCD :: Graph String
        -> Graph String
splitCD g = g >>= f
  where
    f "CD" = Vertex "C"
           + Vertex "D"
    f x    = Vertex x
```



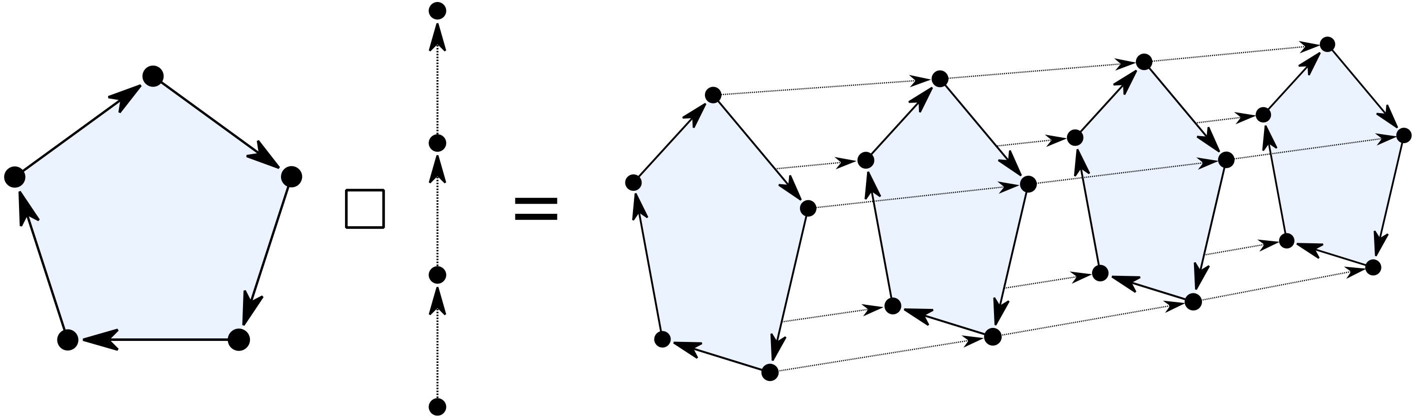
Find induced subgraphs using MonadPlus

```
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
```



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
```

Algebraic graphs with class

```
class Graph g where
    type Vertex g
    empty   :: g
    vertex  :: Vertex g -> g
    overlay :: g -> g -> g
    connect :: g -> g -> g
```

Write code once, reuse for different graph data structures

```
vertices vs = foldr overlay empty (map vertex vs)
clique   vs = foldr connect empty (map vertex vs)
```

Algebraic graphs with class

```
vertices vs = foldr overlay empty (map vertex vs)
clique   vs = foldr connect empty (map vertex vs)
```

```
edge     u  v  = connect (vertex   u ) (vertex   v )
star     u  vs = connect (vertex   u ) (vertices vs)
biclique us vs = connect (vertices us) (vertices vs)
```

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

Algebraic graphs with class

```
vertices vs = foldr overlay empty (map vertex vs)
clique   vs = foldr connect empty (map vertex vs)
```

```
edge     u  v  = connect (vertex   u ) (vertex   v )
star     u  vs = connect (vertex   u ) (vertices vs)
biclique us vs = connect (vertices us) (vertices vs)
```

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

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/snowleopard/alga-theory

Used in industry!

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: edge labels, graph algorithms, compact graph representation, links to topology, etc. Help me!