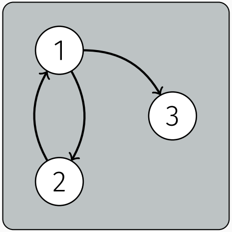
Algebraic Graphs with Class

by Andrey Mokhov

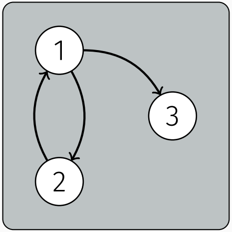
Christoph Madlener

01.06.2021



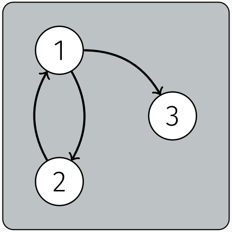
G_1

Introduction



G_1

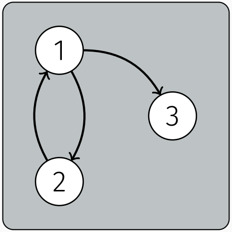
$G = (V, E)$ s.t. $E \subseteq V \times V$



$$G_1 = \Big(\{1,2,3\}, \{(1,2),(1,3),(2,1)\}\Big)$$

$$G = (V, E) \text{ s.t. } E \subseteq V \times V$$

Introduction

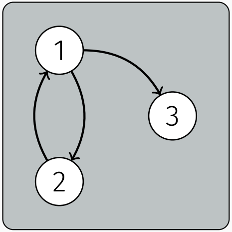


$G = (V, E)$ s.t. $E \subseteq V \times V$

$$G_1 = \Big(\{1,2,3\}, \{(1,2),(1,3),(2,1)\}\Big)$$

In Haskell?

```
data G a = G { vs :: Set a, es :: Set (a,a) }
```



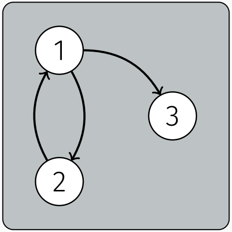
$G = (V, E)$ s.t. $E \subseteq V \times V$

$$G_1 = \Big(\{1, 2, 3\}, \{(1, 2), (1, 3), (2, 1)\}\Big)$$

In Haskell?

```
data G a = G { vs :: Set a, es :: Set (a,a) }
```

```
g1 = G [1,2,3] [(1,2), (1,3), (2,1)]
```



$G = (V, E)$ s.t. $E \subseteq V \times V$

$$G_1 = \Big(\{1, 2, 3\}, \{(1, 2), (1, 3), (2, 1)\}\Big)$$

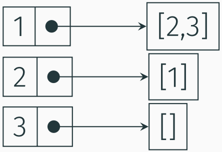
In Haskell?

```
data G a = G { vs :: Set a, es :: Set (a,a) }
```

```
g1 = G [1,2,3] [(1,2), (1,3), (2,1)]
g2 = G [1,2] [(1,3)]
```

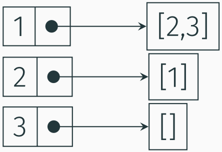
$E \not\subseteq V \times V$

containers
adjacency lists



`containers` & `fgl`

`containers`
adjacency lists

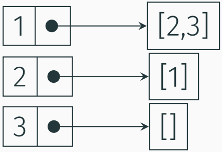


`fgl`
inductive graphs

- inductive datatype: Context of a vertex + Graph

`containers` & `fgl`

`containers`
adjacency lists



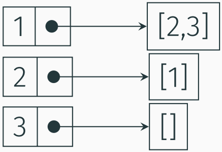
$E \subseteq V \times V$ **?**

`fgl`
inductive graphs

- inductive datatype: Context of a vertex + Graph

containers & fgl

containers
adjacency lists



$E \subseteq V \times V$ **?**

- partial functions →runtime errors

fgl
inductive graphs

- inductive datatype: Context of a vertex + Graph

- simple construction primitives (*“the core”*):

Algebraic Graphs

- simple construction primitives (*“the core”*):
    1. *empty* graph

Algebraic Graphs

- simple construction primitives (*“the core”*):
  1. *empty* graph
  2. single *vertex* graphs

- simple construction primitives (*“the core”*):
    1. *empty* graph
    2. single *vertex* graphs
    3. *overlay*ing two graphs

Algebraic Graphs

- simple construction primitives (*“the core”*):
  1. *empty* graph
  2. single *vertex* graphs
  3. *overlay*ing two graphs
  4. *connect*ing two graphs

Algebraic Graphs

- simple construction primitives (*“the core”*):
  1. *empty* graph
  2. single *vertex* graphs
  3. *overlay*ing two graphs
  4. *connect*ing two graphs
- complete and consistent graph representation

Algebraic Graphs

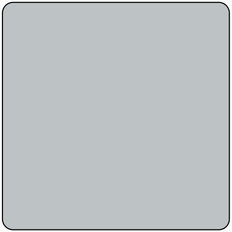
- simple construction primitives (*"the core"*):
  1. *empty* graph
  2. single *vertex* graphs
  3. *overlay*ing two graphs
  4. *connect*ing two graphs
- complete and consistent graph representation
- algebraic structure → formal verification

Algebraic Graphs

- simple construction primitives (*“the core”*):
  1. *empty* graph
  2. single *vertex* graphs
  3. *overlay*ing two graphs
  4. *connect*ing two graphs
- complete and consistent graph representation
- algebraic structure → formal verification
- compact representation for dense graphs

Empty Graph & Vertex Graphs

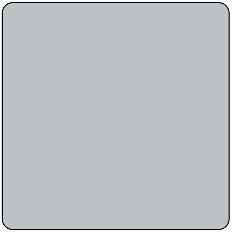
Empty - ε



$$\varepsilon = (\emptyset, \emptyset)$$

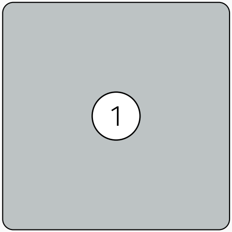
Empty Graph & Vertex Graphs

Empty - ε



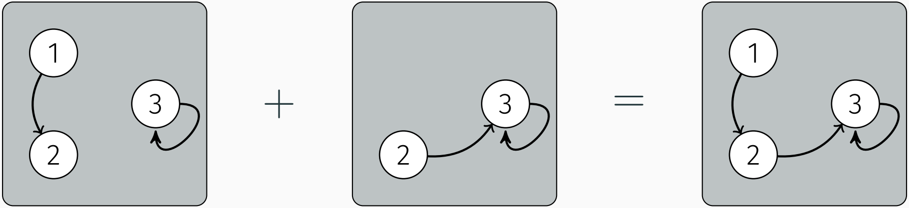
$$\varepsilon = (\emptyset, \emptyset)$$

Vertex



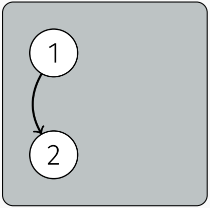
$$1 = (\{1\}, \emptyset)$$

Overlay ($+$)

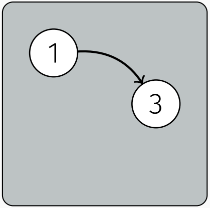


$$(V_1, E_1) + (V_2, E_2) := (V_1 \cup V_2, E_1 \cup E_2)$$

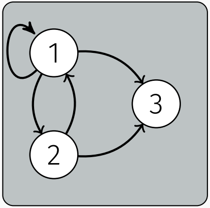
Connect ($\rightarrow$)



$\rightarrow$



$=$



$$(V_1, E_1) \rightarrow (V_2, E_2) \coloneqq (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$$

Graph Construction

- for a graph $G = (V, E)$:

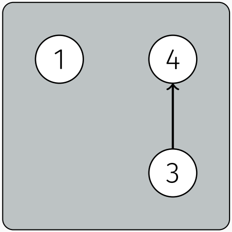
$$\sum_{v \in V} v + \sum_{(u,v) \in E} u \to v$$

Graph Construction

- for a graph $G = (V, E)$:

$$\sum_{v \in V} v + \sum_{(u,v) \in E} u \to v$$

- for $V = \{1\}$ and $E = \{(3, 4)\}$:



Overlay $(+)$ and connect $(\rightarrow)$ form an algebra abiding the following axioms

Overlay ($+$) and connect ($\rightarrow$) form an algebra abiding the following axioms

- $+$ is commutative and associative
  - $x + y = y + x$
  - $x + (y + z) = (x + y) + z$

Overlay ($+$) and connect ($\rightarrow$) form an algebra abiding the following axioms

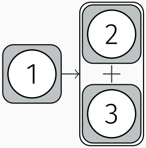
- $+$ is commutative and associative
  - $x + y = y + x$
  - $x + (y + z) = (x + y) + z$
- $(\mathcal{G}, \rightarrow, \varepsilon)$ is a monoid
  - $\varepsilon \rightarrow x = x \rightarrow \varepsilon = x$
  - $x \rightarrow (y \rightarrow z) = (x \rightarrow y) \rightarrow z$

An Algebra of Graphs - Distributivity

- $\rightarrow$ distributes over $+$
  - $x \rightarrow (y + z) = x \rightarrow y + x \rightarrow z$
  - $(x + y) \rightarrow z = x \rightarrow z + y \rightarrow z$

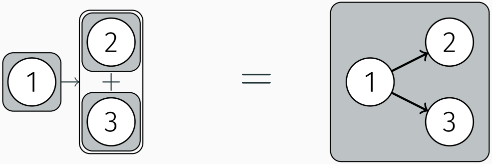
An ALGEBRA of Graphs - Distributivity

- $\rightarrow$ distributes over $+$
  - $x \rightarrow (y + z) = x \rightarrow y + x \rightarrow z$
  - $(x + y) \rightarrow z = x \rightarrow z + y \rightarrow z$



$1 \rightarrow (2 + 3)$

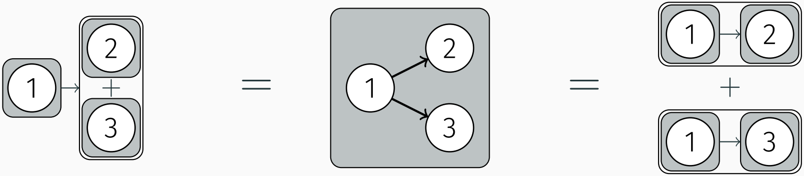
- $\rightarrow$ distributes over $+$
  - $x \rightarrow (y + z) = x \rightarrow y + x \rightarrow z$
  - $(x + y) \rightarrow z = x \rightarrow z + y \rightarrow z$



$1 \rightarrow (2 + 3)$

An ALGEBRA of Graphs - Distributivity

- $\rightarrow$ distributes over $+$
  - $x \rightarrow (y + z) = x \rightarrow y + x \rightarrow z$
  - $(x + y) \rightarrow z = x \rightarrow z + y \rightarrow z$



$1 \rightarrow (2 + 3)$

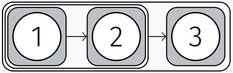
$1 \rightarrow 2 + 1 \rightarrow 3$

An ALGEBRA of Graphs - Decomposition

- $x \to y \to z = x \to y + x \to z + y \to z$

An ALGEBRA of Graphs - Decomposition

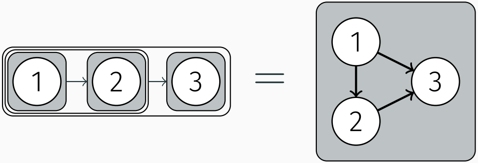
- $x \to y \to z = x \to y + x \to z + y \to z$



$1 \to 2 \to 3$

An ALGEBRA of Graphs - Decomposition

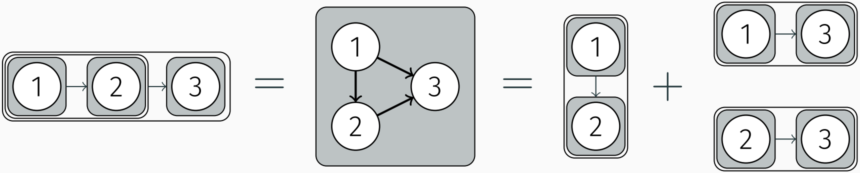
- $x \to y \to z = x \to y + x \to z + y \to z$



$1 \to 2 \to 3$

An ALGEBRA of Graphs - Decomposition

- $x \to y \to z = x \to y + x \to z + y \to z$



$1 \to 2 \to 3$

$1 \to 2 + 1 \to 3 + 2 \to 3$

Under these axioms $+$ and $\rightarrow$ almost form a semiring

Under these axioms $+$ and $\rightarrow$ almost form a semiring

- $(\mathcal{G}, +, \varepsilon)$ is a commutative, idempotent monoid

Under these axioms $+$ and $\rightarrow$ almost form a semiring

- $(\mathcal{G}, +, \varepsilon)$ is a commutative, idempotent monoid
- $(\mathcal{G}, \rightarrow, \varepsilon)$ is a monoid

An ALMOST Semiring on Graphs

Under these axioms $+$ and $\rightarrow$ almost form a semiring

- $(\mathcal{G}, +, \varepsilon)$ is a commutative, idempotent monoid
- $(\mathcal{G}, \rightarrow, \varepsilon)$ is a monoid
- $\rightarrow$ distributes over $+$

An Almost Semiring on Graphs

Under these axioms $+$ and $\rightarrow$ almost form a semiring

- $(\mathcal{G}, +, \varepsilon)$ is a commutative, idempotent monoid
- $(\mathcal{G}, \rightarrow, \varepsilon)$ is a monoid
- $\rightarrow$ distributes over $+$

- Missing: annihilating zero for connect ($x \rightarrow \varepsilon \neq \varepsilon$)

An ALMOST Semiring on Graphs

Under these axioms $+$ and $\rightarrow$ almost form a semiring

- $(\mathcal{G}, +, \varepsilon)$ is a commutative, idempotent monoid
- $(\mathcal{G}, \rightarrow, \varepsilon)$ is a monoid
- $\rightarrow$ distributes over $+$

- Missing: annihilating zero for connect ($x \rightarrow \varepsilon \neq \varepsilon$)
- Unusual: shared identity ε of $+$ and $\rightarrow$

Obtain other classes of graphs by modifying the set of axioms:

Obtain other classes of graphs by modifying the set of axioms:

- Undirected graphs: $\rightarrow$ is commutative $(x \rightarrow y = y \rightarrow x)$

Beyond Directed Graphs

Obtain other classes of graphs by modifying the set of axioms:

- Undirected graphs: $\rightarrow$ is commutative ($x \rightarrow y = y \rightarrow x$)
- Reflexive graphs: $v = v \rightarrow v$ for any vertex v

Beyond Directed Graphs

Obtain other classes of graphs by modifying the set of axioms:

- Undirected graphs: $\rightarrow$ is commutative ($x \rightarrow y = y \rightarrow x$)
- Reflexive graphs: $v = v \rightarrow v$ for any vertex v
- Transitive graphs:

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

Beyond Directed Graphs

Obtain other classes of graphs by modifying the set of axioms:

- Undirected graphs: $\rightarrow$ is commutative ($x \rightarrow y = y \rightarrow x$)
- Reflexive graphs: $v = v \rightarrow v$ for any vertex v
- Transitive graphs:

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

- Preorders (reflexive + transitive), equivalence relations (preorder + undirected)

Beyond Directed Graphs

Obtain other classes of graphs by modifying the set of axioms:

- Undirected graphs: $\rightarrow$ is commutative ($x \rightarrow y = y \rightarrow x$)
- Reflexive graphs: $v = v \rightarrow v$ for any vertex v
- Transitive graphs:

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

- Preorders (reflexive + transitive), equivalence relations (preorder + undirected)
- Hypergraphs

A Type Class for Algebraic Graphs

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

A Type Class for Algebraic Graphs

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

graph :: Graph g => [V g] -> [(V g, V g)] -> g
graph vs es = overlay (vertices vs) (edges es)
```

A Type Class for Algebraic Graphs

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

graph :: Graph g => [V g] -> [(V g, V g)] -> g
graph vs es = overlay (vertices vs) (edges es)
```

→ Graph construction & transformation library – independent of concrete graph representation

Using Isabelle/HOL

- (type class + axioms) $\sim$ *locale* in Isabelle

```
locale algebraic_pre_graph =
  fixes empty :: 'g ("ε")
    and vertex :: "'v ⇒ 'g"
    and overlay :: "'g ⇒ 'g ⇒ 'g" (infixl ‹+› 75)
    and connect :: "'g ⇒ 'g ⇒ 'g" (infixl ‹→› 80)
```

Using Isabelle/HOL

- (type class + axioms) $\sim$ *locale* in Isabelle

```
locale algebraic_pre_graph =
  fixes empty :: 'g ("ε")
    and vertex :: "'v ⇒ 'g"
    and overlay :: "'g ⇒ 'g ⇒ 'g" (infixl ‹+› 75)
    and connect :: "'g ⇒ 'g ⇒ 'g" (infixl ‹→› 80)
```

- instantiation with $G = (V, E)$ representation → formal proof of completeness and consistency

Challenge: graph type $'g$

Challenge: graph type $'g$

- subgraph relation $x \sqsubseteq y :\Leftrightarrow x + y = y$

Challenge: graph type $'g$

- subgraph relation $x \sqsubseteq y :\Leftrightarrow x + y = y$

definition $\mathtt{has\_vertex}$ $::$ $"'g \Rightarrow \ 'v \Rightarrow \mathtt{bool}"$ where

$"\mathtt{has\_vertex}\ g\ u \equiv \mathtt{vertex}\ u \sqsubseteq g"$

definition $\mathtt{has\_edge}$ $::$ $"'g \Rightarrow \ 'v \Rightarrow \ 'v \Rightarrow \mathtt{bool}"$ where

$"\mathtt{has\_edge}\ g\ u\ v \equiv \mathtt{edge}\ u\ v \sqsubseteq g"$

Define walks inductively

inductive *vwalk* where

```
vwalk_Nil: "vwalk g []" |
vwalk_single: "has_vertex g u ⟹ vwalk g [u]" |
vwalk_Cons: "vwalk g (v#xs) ⟹ has_edge g u v
⟹ vwalk g (u#v#xs)"
```

Formal Verification - Walks

Define walks inductively

inductive *vwalk* where

```
vwalk_Nil: "vwalk g []" |
vwalk_single: "has_vertex g u ⟹ vwalk g [u]" |
vwalk_Cons: "vwalk g (v#xs) ⟹ has_edge g u v
⟹ vwalk g (u#v#xs)"
```

- Appending/splitting walks, reachability

Define walks inductively

inductive *vwalk* where

```
  vwalk_Nil: "vwalk g []" |
  vwalk_single: "has_vertex g u ⟹ vwalk g [u]" |
  vwalk_Cons: "vwalk g (v#xs) ⟹ has_edge g u v
⟹ vwalk g (u#v#xs)"
```

- Appending/splitting walks, reachability
- → Polymorphic graph *formalization* library

Compact Representation for Dense Graphs

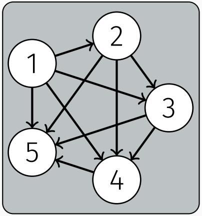
$$1 \to 2 \to \ldots \to n$$

is a linear size representation of a graph with a quadratic number of edges

Compact Representation for Dense Graphs

$$1 \to 2 \to \ldots \to n$$

is a linear size representation of a graph with a quadratic number of edges

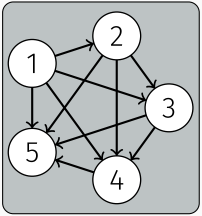


$$1 \to 2 \to 3 \to 4 \to 5$$

Compact Representation for Dense Graphs

$$1 \to 2 \to \ldots \to n$$

is a linear size representation of a graph with a quadratic number of edges



$$1 \to 2 \to 3 \to 4 \to 5$$

Open question

- Can we exploit this compact representation, i.e. find algorithms that work directly on algebraic graphs expressions?

Deep Embedding

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

Deep Embedding

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

- Custom `Eq` instance required!
  (`Empty == Overlay Empty Empty`)

Reuse well-known Haskell abstractions for Graph transformation

Reuse well-known Haskell abstractions for Graph transformation

- `instance Functor Graph` → contracting vertices

Deep Embedding - Graph Transformation

Reuse well-known Haskell abstractions for Graph transformation

- `instance Functor Graph` → contracting vertices
- `instance Monad Graph` → splitting vertices, inducing subgraphs

Conclusion

- alternative approach to graph construction/transformation
    - small core of primitives - safe and complete
    - suitable for functional programming and formal verification
    - available on `hackage`[1]

[1] http://hackage.haskell.org/package/algebraic-graphs

Conclusion

- alternative approach to graph construction/transformation
  - small core of primitives - safe and complete
  - suitable for functional programming and formal verification
  - available on `hackage`[1]
- two main directions for future work
  1. exploit compact representation/other applications of algebraic graph expressions
  2. exploit "polymorphism" of type class/locale

[1] http://hackage.haskell.org/package/algebraic-graphs

Graph Construction - Vertices

- Graph with only vertices:

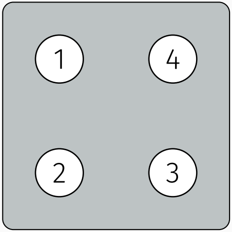
$$\sum_{v \in V} v$$

Graph Construction - Vertices

- Graph with only vertices:

$$\sum_{v \in V} v$$

- for $V = \{1, 2, 3, 4\}$:



Graph Construction - Edges

- Graph from a set of edges:

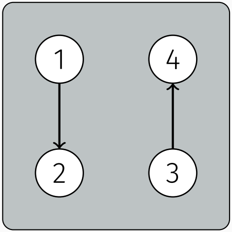
$$\sum_{(u,v) \in E} u \to v$$

Graph Construction - Edges

- Graph from a set of edges:

$$\sum_{(u,v)\in E} u \to v$$

- for $E = \{(1,2),(3,4)\}$:



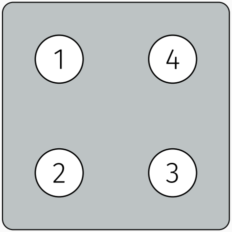
Graph Construction

```
vertices :: [a] -> Graph a
vertices = foldr Overlay Empty . map Vertex
```

Graph Construction

```
vertices :: [a] -> Graph a
vertices = foldr Overlay Empty . map Vertex
```

`vertices [1,2,3,4]` =



Graph Construction

```haskell
edge :: a -> a -> Graph a
edge u v = Connect (Vertex u) (Vertex v)
```

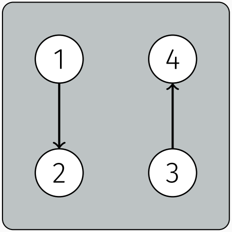
Graph Construction

```
edge :: a -> a -> Graph a
edge u v = Connect (Vertex u) (Vertex v)

edges :: [(a,a)] -> Graph a
edges = foldr Overlay Empty . map (uncurry edge)
```

Graph Construction

```
edge :: a -> a -> Graph a
edge u v = Connect (Vertex u) (Vertex v)

edges :: [(a,a)] -> Graph a
edges = foldr Overlay Empty . map (uncurry edge)
```

`edges [(1,2),(3,4)]` $=$



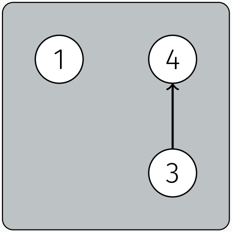
Graph Construction

```
graph :: [a] -> [(a,a)] -> Graph a
graph vs es = Overlay (vertices vs) (edges es)
```

Graph Construction

```
graph :: [a] -> [(a,a)] -> Graph a
graph vs es = Overlay (vertices vs) (edges es)
```

`graph [1] [(3,4)]` =



Graph Transformation - Functor

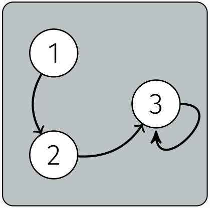
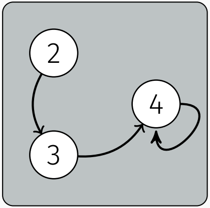
Functor

```
instance Functor Graph where
  ...
  fmap f (Vertex u) = Vertex (f u)
  ...
```

Graph Transformation - Functor

Functor

```
instance Functor Graph where
  ...
  fmap f (Vertex u) = Vertex (f u)
  ...
```

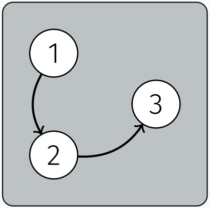
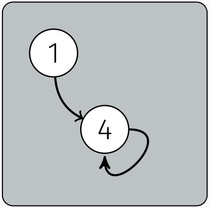
`fmap (+1)`  $=$ 

Graph Transformation - Merging Vertices

```haskell
mergeVs :: (a -> Bool) -> a -> Graph a -> Graph a
mergeVs p v = fmap $ \u -> if p u then v else u
```

Graph Transformation - Merging Vertices

```haskell
mergeVs :: (a -> Bool) -> a -> Graph a -> Graph a
mergeVs p v = fmap $ \u -> if p u then v else u
```

`mergeVs (>1) 4`  = 

Graph Transformation - Monad

```
instance Monad Graph where
  ...
  (Vertex u) >>= f = f u
  ...
```

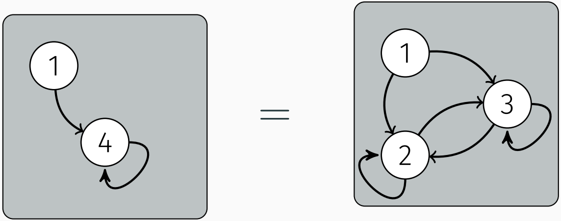
Graph Transformation - Monad

```haskell
instance Monad Graph where
  ...
  (Vertex u) >>= f = f u
  ...

splitVertex :: (a -> Bool) -> [a] -> Graph a
splitVertex p vs = g >>= \u -> if p u
                               then vertices vs
                               else Vertex u
```

Graph Transformation - Monad

```
instance Monad Graph where
  ...
  (Vertex u) >>= f = f u
  ...

splitVertex :: (a -> Bool) -> [a] -> Graph a
splitVertex p vs = g >>= \u -> if p u
                               then vertices vs
                               else Vertex u
```

`splitVertex 4 [2,3]`



Graph Transformation - MonadPlus

```haskell
instance Graph MonadPlus where
  mzero = Empty
  mplus = Overlay
```

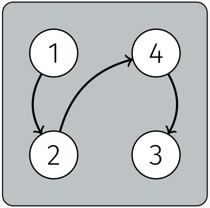
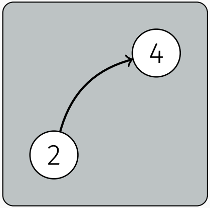
Graph Transformation - MonadPlus

```
instance Graph MonadPlus where
  mzero = Empty
  mplus = Overlay

induce :: (a -> Bool) -> Graph a -> Graph a
induce = mfilter
```

Graph Transformation - MonadPlus

```
instance Graph MonadPlus where
  mzero = Empty
  mplus = Overlay
```

```
induce :: (a -> Bool) -> Graph a -> Graph a
induce = mfilter
```

`induce even`  = 

Graph Equality

Structural equality is not suitable:

```
Overlay (Vertex 1) Empty == Vertex 1
```

Graph Equality

Structural equality is not suitable:

`Overlay (Vertex 1) Empty == Vertex 1`

`Eq` instance for Algebraic Graphs

- Current implementation: build adjacency map

Graph Equality

Structural equality is not suitable:

`Overlay (Vertex 1) Empty == Vertex 1`

`Eq` instance for Algebraic Graphs

- Current implementation: build adjacency map
- Possible approach: *minimize* graph expressions →*Modular Graph Decomposition*

Compact Representation for Dense Graphs

Fully connected (undirected) graph

```
clique :: [a] -> Graph a
clique = foldr Connect Empty . map Vertex
```

Compact Representation for Dense Graphs

Fully connected (undirected) graph

```
clique :: [a] -> Graph a
clique = foldr Connect Empty . map Vertex
```

- Linear size representation
- quadratic in size when using e.g. adjacency map

Compact Representation for Dense Graphs

Fully connected (undirected) graph

```
clique :: [a] -> Graph a
clique = foldr Connect Empty . map Vertex
```

- Linear size representation
- quadratic in size when using e.g. adjacency map

Open question

- Can we exploit this compact representation, i.e. find algorithms that work directly on algebraic graphs?

Formal Verification

- using Isabelle/HOL

Formal Verification

- using Isabelle/HOL
- inductive datatype → proofs by induction, `auto`

Formal Verification

- using Isabelle/HOL
- inductive datatype → proofs by induction, `auto`
- `quotient_type` (think `Eq` instance) based on tuple representation

Formal Verification

- using Isabelle/HOL
- inductive datatype → proofs by induction, `auto`
- `quotient_type` (think `Eq` instance) based on tuple representation

Future work

- Minimization
- Algorithms, applications beyond graph construction/transformation