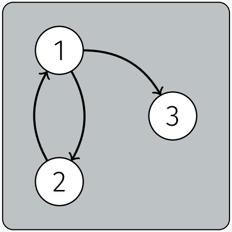
Algebraic Graphs with Class

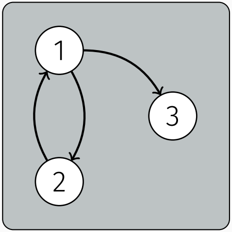
Christoph Madlener

01.06.2021

Introduction

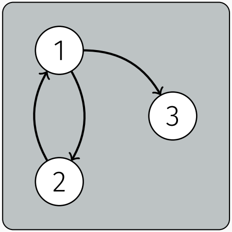


G_1



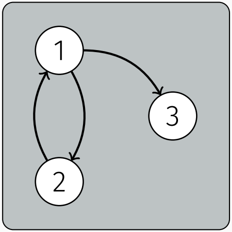
G_1

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



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

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

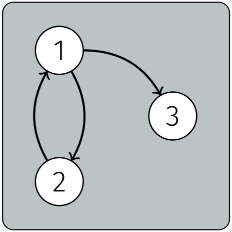


$$G = (V, E) \text{ s.t. } E \subseteq V \times V$$

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

In Haskell?

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



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

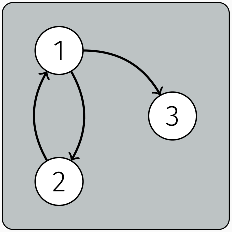
$$G_1 = \Big(\{1, 2, 3\}, \{(1, 2), (1, 3), (2, 1)\}\Big)$$

In Haskell?

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

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

Introduction



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

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

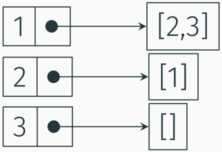
In Haskell?

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

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

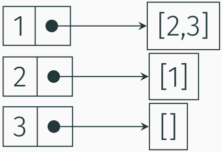
$E \not\subseteq V \times V$

containers
adjacency lists



containers & fgl

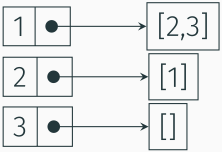
containers
adjacency lists



fgl
inductive graphs

- inductive datatype: Context of a vertex + Graph

containers
adjacency lists



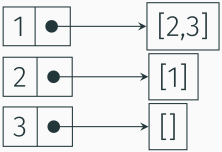
$E \subseteq V \times V$?

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

Algebraic Graphs

- complete and consistent graph representation
- simple construction primitives (*“the core”*)

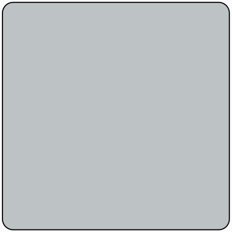
Algebraic Graphs

- complete and consistent graph representation
- simple construction primitives (*“the core”*)

Achieved by the datatype

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

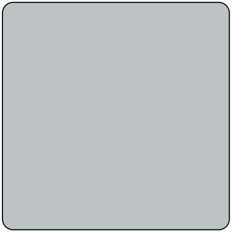
Empty - ε



$\texttt{Empty} = \varepsilon = (\emptyset, \emptyset)$

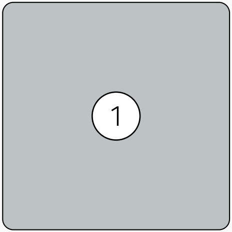
Empty (ε) & Vertex

Empty - ε



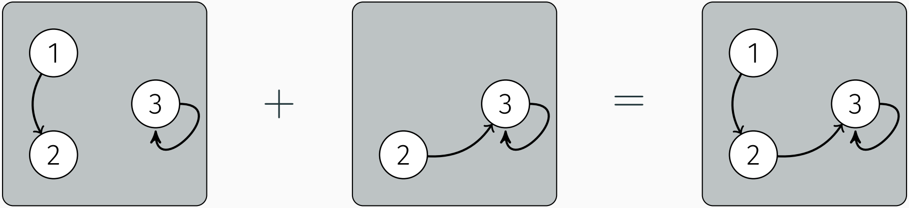
`Empty` $= \varepsilon = (\emptyset, \emptyset)$

Vertex



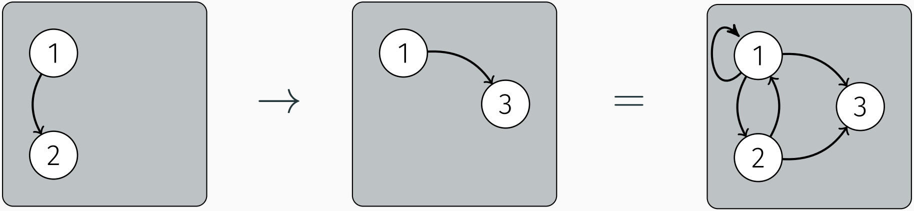
`Vertex 1` $= (\{1\}, \emptyset)$

Overlay ($+$)



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

Connect ($\rightarrow$ / $*$)



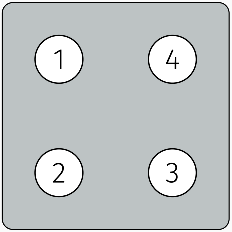
$$(V_1, E_1) \rightarrow (V_2, E_2) := (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$$

Graph Construction

```haskell
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)
```

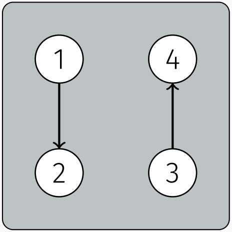
Graph Construction

```
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)]` $=$

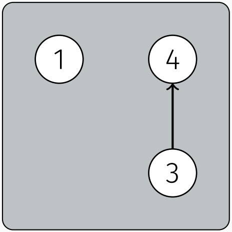


Graph Construction

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

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

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



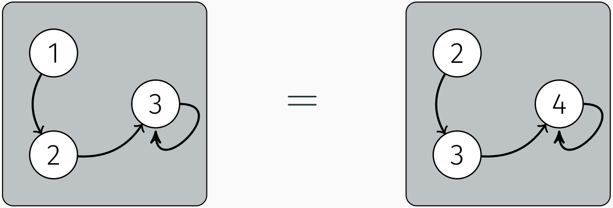
Graph Transformation - Functor

Functor

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

Functor

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

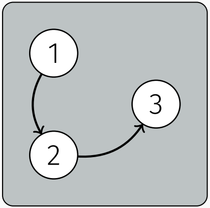
`fmap (+1)` 

Graph Transformation - Merging Vertices

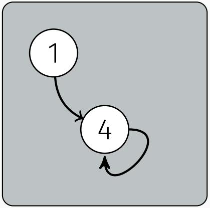
```
mergeVs :: (a -> Bool) -> a -> Graph a -> Graph a
mergeVs p v = fmap $ \u -> if p u then v else u
```

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

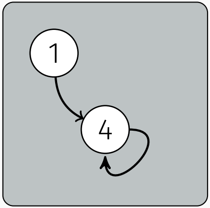
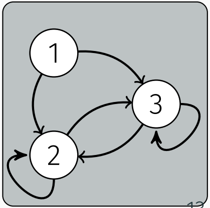
```haskell
instance Monad Graph where
  ...
  (Vertex u) >>= f = f u
  ...
```

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
```

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
```

splitVertex 4 [2,3]  = 

Graph Transformation - MonadPlus

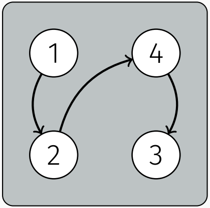
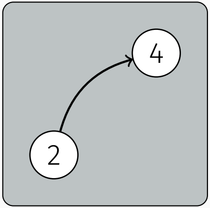
```
instance Graph MonadPlus where
  mzero = Empty
  mplus = Overlay
```

Graph Transformation - MonadPlus

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

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

Graph Transformation - MonadPlus

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

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

`induce even`  = 

Overlay $(+)$ and connect $(\rightarrow)$ almost form a semiring over graphs

Overlay ($+$) and connect ($\rightarrow$) almost form a semiring over graphs

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

Overlay ($+$) and connect ($\rightarrow$) almost form a semiring over graphs

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

Overlay ($+$) and connect ($\rightarrow$) almost form a semiring over graphs

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

Overlay ($+$) and connect ($\rightarrow$) almost form a semiring over graphs

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

- "shared" identity ε

Overlay $(+)$ and connect $(\rightarrow)$ almost form a semiring over graphs

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

- "shared" identity ε
- no "zero" for $\rightarrow$ (i.e. $g \rightarrow 0 = 0$)

Decomposition

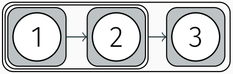
Additional axiom for algebra of graphs

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

Decomposition

Additional axiom for algebra of graphs

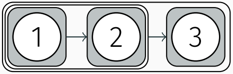
$$x \rightarrow y \rightarrow z = x \rightarrow y + x \rightarrow z + y \rightarrow z$$



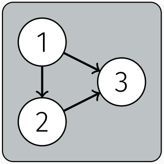
Decomposition

Additional axiom for algebra of graphs

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



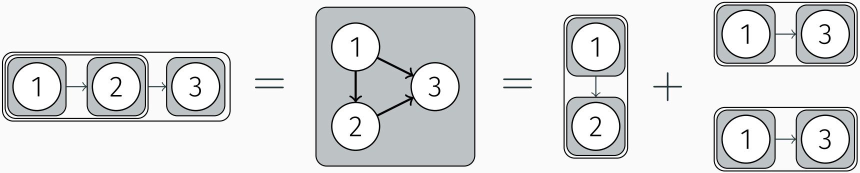
$=$



Decomposition

Additional axiom for algebra of graphs

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



Subgraph Relation

Define subgraphs in terms of $(+)$ and equality:

$$x \sqsubseteq y \iff x + y = y$$

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

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

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

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

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

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

- Undirected graphs: $\leftrightarrow$ is commutative ($x \leftrightarrow y = y \leftrightarrow x$)
- Reflexive graphs: $x = x \rightarrow x$
- 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: $\leftrightarrow$ is commutative ($x \leftrightarrow y = y \leftrightarrow x$)
- Reflexive graphs: $x = x \rightarrow x$
- 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: $\leftrightarrow$ is commutative ($x \leftrightarrow y = y \leftrightarrow x$)
- Reflexive graphs: $x = x \to x$
- Transitive graphs:

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

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

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

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

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

→Polymorphic graph construction/transformation library

Formal Verification

- define `locale` in Isabelle/HOL (think Haskell class + axioms)

Formal Verification

- define `locale` in Isabelle/HOL (think Haskell class + axioms)
- equational reasoning for simple statements

$$\texttt{vertices xs} \sqsubseteq \texttt{clique xs}$$

Formal Verification

- define `locale` in Isabelle/HOL (think Haskell class + axioms)
- equational reasoning for simple statements

$$\texttt{vertices xs} \sqsubseteq \texttt{clique xs}$$

- formal proof of completeness and consistency

Formal Verification

- formalized material on vertex walks, reachability

Formal Verification

- formalized material on vertex walks, reachability
- probably additional axioms required ($\varepsilon \neq \texttt{vertex u}$)

Formal Verification

- formalized material on vertex walks, reachability
- probably additional axioms required ($\varepsilon \neq$ `vertex u`)

Future work

- more graph theory (e.g. strongly connected components)
- algorithms

Formal Verification

- formalized material on vertex walks, reachability
- probably additional axioms required ($\varepsilon \neq$ `vertex u`)

Future work

- more graph theory (e.g. strongly connected components)
- algorithms

→ *Verified* polymorphic graph library

Conclusion

- alternative approach to graph construction/transformation
    - small core of primitives - safe and complete

Conclusion

- alternative approach to graph construction/transformation
  - small core of primitives - safe and complete
- two main directions for future work
  1. exploit compact representation/other applications of data type
  2. exploit polymorphism of type class/locale