Algebraic Graphs with Class (Functional Pearl)

Andrey Mokhov
Newcastle University, United Kingdom

Abstract

The paper presents a minimalistic and elegant approach to working with graphs in Haskell. It is built on a rigorous mathematical foundation — an algebra of graphs — that allows us to apply equational reasoning for proving the correctness of graph transformation algorithms. Algebraic graphs let us avoid partial functions typically caused by 'malformed graphs' that contain an edge referring to a non-existent vertex. This helps to liberate APIs of existing graph libraries from partial functions.

The algebra of graphs can represent directed, undirected, reflexive and transitive graphs, as well as hypergraphs, by appropriately choosing the set of underlying axioms. The flexibility of the approach is demonstrated by developing a library for constructing and transforming polymorphic graphs.

CCS Concepts • **Mathematics of computing**;

Keywords Haskell, algebra, graph theory

ACM Reference Format:
Andrey Mokhov. 2017. Algebraic Graphs with Class (Functional Pearl). In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7-8, 2017 (Haskell'17),* 12 pages.
https://doi.org/10.1145/3122955.3122956

1 Introduction

Graphs are ubiquitous in computing, yet working with graphs often requires painfully low-level fiddling with sets of vertices and edges. Building high-level abstractions is difficult, because the commonly used foundation – the pair (V, E) of vertex set V and edge set $E \subseteq V \times V$ – is a source of partial functions. We can represent the pair (V, E) by the following simple data type[1]:

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

Now `G [1,2,3] [(1,2),(2,3)]` is the graph with three vertices $V = \{1, 2, 3\}$ and two edges $E = \{(1, 2), (2, 3)\}$. The consistency invariant $E \subseteq V \times V$ holds. But what is `G [1] [(1,2)]`? The edge refers to the non-existent vertex 2, breaking the invariant, and there is no easy way to reflect this in types. Perhaps, our data type is just too simplistic; let us look at state-of-the-art graph libraries instead.

The containers library is designed for performance and powers GHC itself. It represents graphs by *adjacency arrays* [King and Launchbury 1995] whose consistency invariant is not statically checked, which can lead to runtime usage errors such as 'index out of range'. Another popular library fgl uses the *inductive graph representation* [Erwig 2001], but its API also has partial functions, e.g. inserting an edge can fail with the 'edge from non-existent vertex' error. Both containers and fgl are treasure troves of graph algorithms, but it is easy to make an error when using them. Is there a safe graph construction interface we can build on top?

In this paper we present *algebraic graphs* — a new interface for constructing and transforming graphs (more precisely, graphs with labelled vertices and unlabelled edges). We abstract away from graph representation details and characterise graphs by a set of axioms, much like numbers are algebraically characterised by *rings* [Mac Lane and Birkhoff 1999]. Our approach is based on the *algebra of parameterised graphs*, a mathematical formalism used in digital circuit design [Mokhov and Khomenko 2014], which we simplify and adapt to the context of functional programming.

Algebraic graphs have a safe and minimalistic core of four graph construction primitives, as captured by the following data type:

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

Here `Empty` and `Vertex` construct the *empty* and *single-vertex* graphs, respectively; `Overlay` composes two graphs by taking the union of their vertices and edges, and `Connect` is similar to `Overlay` but also creates edges between vertices of the two graphs, see Fig. 1 for examples. The *overlay* and *connect* operations have two important properties: (i) they are closed on the set of graphs, i.e. are total functions, and (ii) they can be used to construct any graph starting from the empty and single-vertex graphs. For example, `Connect (Vertex 1) (Vertex 2)` is the graph with two vertices $\{1, 2\}$ and a single edge $(1, 2)$. Malformed graphs, such as `G [1] [(1,2)]`, cannot be expressed in this core language.

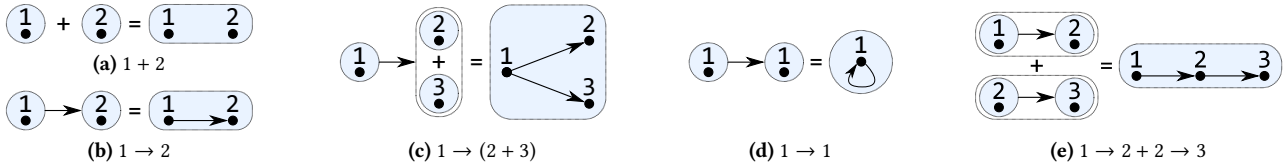
The main goal of this paper is to demonstrate that *this core is a safe, flexible and elegant foundation for working with graphs that have no edge labels*. Our specific contributions are:

- Compared to existing libraries, algebraic graphs have a smaller core (just four graph construction primitives), are more compositional (hence greater code reuse), and have no partial functions (hence fewer opportunities for usage errors). We present the core and justify these claims in §2.
- The core has a simple mathematical structure fully characterised by a set of axioms (§3). This makes the proposed interface easier for testing and formal verification. We show that the core is *complete*, i.e. any graph can be constructed, and *sound*, i.e. malformed graphs cannot be constructed.
- Under the basic set of axioms, algebraic graphs correspond to directed graphs. As we show in §4, by extending the algebra with additional axioms, we can represent undirected, reflexive, transitive graphs, their combinations, and hypergraphs. Importantly, the core remains unchanged, which allows us to define highly reusable polymorphic functions on graphs.
- We develop a library[2] for constructing and transforming algebraic graphs and demonstrate its flexibility in §5.

[1] Although in this paper we exclusively use Haskell, the problem we solve is general and the proposed approach can be readily adapted to other programming languages.

Haskell'17, September 7-8, 2017, Oxford, UK

[2] The library is on Hackage: http://hackage.haskell.org/package/algebraic-graphs.



(a) 1 + 2 (b) 1 → 2 (c) 1 → (2 + 3) (d) 1 → 1 (e) 1 → 2 + 2 → 3

Figure 1. Examples of graph construction. The overlay and connect operations are denoted by + and →, respectively.

Graphs and functional programming have a long history. We review related work in §6. Limitations of the presented approach and future research directions are discussed in §7.

2 The Core

In this section we define the *core* of algebraic graphs comprising four graph construction primitives. We describe the semantics of the primitives using the common representation of graphs by sets of vertices and edges, and then abstract away from this representation by focusing on the laws that these primitives satisfy.

Let G be the set of all directed graphs whose vertices come from a fixed universe $\mathbb{V}$. As an example, we can think of graphs whose vertices are positive integers. A graph $g \in G$ can be represented by a pair (V, E) where $V \subseteq \mathbb{V}$ is the set of its vertices and $E \subseteq V \times V$ is the set of its edges. As mentioned in §1, when $E \nsubseteq V \times V$ the pair (V, E) is inconsistent and does not correspond to a graph.

When one needs to guarantee the internal consistency of a data structure, the standard solution is to define an abstract interface that encapsulates the data structure and provides a set of safe construction primitives. This is exactly the approach we take.

2.1 Constructing Graphs

The simplest possible graph is the *empty* graph. We denote it by ε, therefore $\varepsilon = (\varnothing, \varnothing)$ and $\varepsilon \in G$. A graph with a *single vertex* $v \in \mathbb{V}$ is denoted simply by v. For example, $1 \in G$ is the graph $(1, \varnothing)$.

To construct larger graphs from the above primitives we define binary operations *overlay* and *connect*, denoted by + and →, respectively. The overlay operation + is defined as

$$(V_1, E_1) + (V_2, E_2) \stackrel{\text{def}}{=} (V_1 \cup V_2, E_1 \cup E_2).$$

That is, the overlay of two graphs comprises the union of their vertices and edges. The connect → operation is defined similarly:

$$(V_1, E_1) \rightarrow (V_2, E_2) \stackrel{\text{def}}{=} (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2).$$

The difference is that when we connect two graphs, an edge is added from each vertex of the left-hand argument to each vertex of the right-hand argument[3]. Note that the connect operation is the only source of edges when constructing graphs. As we will see in §3, overlay and connect are very similar to addition and multiplication. We therefore give connect a higher precedence, i.e. $1 + 2 \rightarrow 3$ is interpreted as $1 + (2 \rightarrow 3)$. Fig. 1 illustrates a few examples of graph construction using the defined primitives:

- $1 + 2$ is the graph with two isolated vertices 1 and 2.
- $1 \rightarrow 2$ is the graph with an edge between vertices 1 and 2.
- $1 \rightarrow (2+3)$ comprises vertices $\{1, 2, 3\}$ and edges $\{(1, 2), (1, 3)\}$.
- $1 \rightarrow 1$ is the graph with vertex 1 and the *self-loop*.
- $1 \rightarrow 2 + 2 \rightarrow 3$ is the *path graph* on vertices $\{1, 2, 3\}$.

[3]Our definitions of overlay and connect coincide with those of graph *union* and *join*, respectively, e.g see Harary [1969], however the arguments of union and join are typically assumed to have disjoint sets of vertices. We make no such assumptions, hence our definitions are total: any graphs can be composed using overlay and connect.

As shown in §1, the core can be represented by a simple data type `Graph`, parameterised by the type of vertices `a`. To make the core more reusable, the next subsection defines the core type class that has the usual inhabitants, such as the pair (V, E), data types from containers and fgl, as well as other, stranger forms of life.

2.2 Type Class

We abstract the graph construction primitives defined in §2.1 as the type class `Graph`[4]:

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

Here the associated type[5] `Vertex g` corresponds to the universe of graph vertices $\mathbb{V}$, `empty` is the empty graph ε, `vertex` constructs a graph with a single vertex, and `overlay` and `connect` compose given graphs according to the definitions in §2.1. All methods of the type class are total, i.e. are defined for all possible inputs, therefore, the presented API allows *fewer opportunities for usage errors* and *greater opportunities for reuse*.

Let us put the interface to the test and construct some graphs. A single edge is obtained by connecting two vertices:

```
edge :: Graph g => Vertex g -> Vertex g -> g
edge x y = connect (vertex x) (vertex y)
```

The graphs in Fig. 1(b,d) are `edge 1 2` and `edge 1 1`, respectively. A graph that contains a given list of isolated vertices can be constructed as follows:

```
vertices :: Graph g => [Vertex g] -> g
vertices = foldr overlay empty . map vertex
```

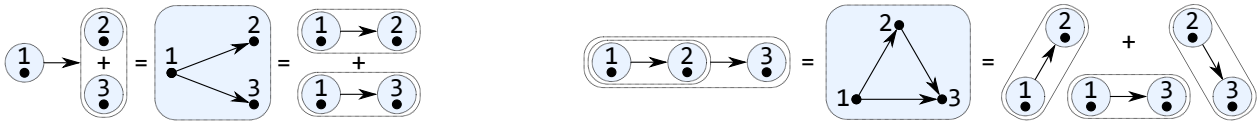
That is, we turn each vertex into a singleton graph and overlay the results. The graph in Fig. 1(a) is `vertices [1,2]`. By replacing `overlay` with `connect` in the above definition, we obtain a directed *clique* – a fully connected graph on a given list of vertices:

```
clique :: Graph g => [Vertex g] -> g
clique = foldr connect empty . map vertex
```

For example, `clique [1,2,3]` expands to $1 \rightarrow 2 \rightarrow 3 \rightarrow \varepsilon$, i.e. the graph with three vertices $\{1, 2, 3\}$ and three edges $(1, 2)$, $(1, 3)$ and $(2, 3)$. Note that it is different from the graph in Fig. 1(e).

The graph construction functions defined above are total, fully polymorphic, and elegant. Thanks to the minimalistic core type class, it is easy to wrap your favourite graph library into the described interface, and reuse the above functions, as well as many others that we define throughout this paper.

[4]The name collision (`data Graph` and `class Graph`) is not a problem in practice, because the data type and type class are not used together and live in separate modules.
[5]Associated types [Chakravarty et al. 2005] require the TypeFamilies GHC extension.



(a) Distributivity: $1 \to (2 + 3) = 1 \to 2 + 1 \to 3$

(b) Decomposition: $1 \to 2 \to 3 = 1 \to 2 + 1 \to 3 + 2 \to 3$

Figure 2. Two axioms of the algebra of graphs.

3 Algebraic Structure

The functions `edge`, `vertices` and `clique` defined in the previous section §2 satisfy a few properties that we can intuitively write down and verify at the level of sets of vertices and edges:

- `vertex x = vertices [x]` and `edge x y = clique [x,y]`.
- `vertices xs` $\subseteq$ `clique xs`, where $x \subseteq y$ means x is a *subgraph* of y, i.e. $V_x \subseteq V_y$ and $E_x \subseteq E_y$ hold.
- `clique (xs ++ ys) = connect (clique xs) (clique ys)`.

In this section we characterise the `Graph` type class with a set of axioms that reveal an algebraic structure very similar to a semiring[6]. This provides a convenient framework for proving graph properties, such as those listed above, using equational reasoning. The presented characterisation is generally useful for formal verification, as well as automated testing of graph library APIs.

3.1 Axiomatic Characterisation

The definitions of `vertices` and `clique` in §2.2 use ε as the identity for both overlay $+$ and connect $\to$ operations. This seems unusual, but we can check that $x + \varepsilon = x$ and $x \to \varepsilon = x$ for any graph $x \in G$ by plugging the empty graph into the definitions of overlay and connect, respectively. Furthermore, we can verify the following:

- $(G, +, \varepsilon)$ is an idempotent commutative monoid.
- $(G, \to, \varepsilon)$ is a monoid.
- $\to$ distributes over $+$, as illustrated in Fig. 2(a).

The above looks remarkably close to a semiring, with the only oddity being the shared identity of the two operations. The lack of the annihilating zero element (i.e. $x \to 0 = 0$) and the following *decomposition law* is what makes the algebra of graphs different:

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

Fig. 2(b) illustrates the law by showing that the triangle graph can be obtained in two different ways: by connecting the three vertices of the triangle and by constructing its edges separately and overlaying them.

Interestingly, the fact that overlay and connect share the same identity follows from the decomposition law. Indeed, let ε_+ and $\varepsilon_\to$ stand for the identities of $+$ and $\to$, respectively. Then:

$$\begin{aligned}
\varepsilon_+ &= \varepsilon_+ \to \varepsilon_\to \to \varepsilon_\to && \text{(identity of } \to\text{)} \\
&= \varepsilon_+ \to \varepsilon_\to + \varepsilon_+ \to \varepsilon_\to + \varepsilon_\to \to \varepsilon_\to && \text{(decomposition)} \\
&= \varepsilon_+ + \varepsilon_+ + \varepsilon_\to && \text{(identity of } \to\text{)} \\
&= \varepsilon_\to && \text{(identity of } +\text{)}
\end{aligned}$$

Furthermore, the identity $(x + \varepsilon = x)$ and idempotence $(x + x = x)$ can be proved from the decomposition law, which leads to the following *minimal set of axioms that characterise algebraic graphs.*

[6] See Golan [1999] for a classic overview of semiring applications, where the author hints at the existence of a non-semiring 'algebra of digraphs' whose operations coincide with overlay and connect, referring to an unpublished paper by Anthony P. Stone. Dolan [2013] uses the semiring theory to implement graph algorithms in Haskell.

Algebraic graphs are characterised by the following 8 axioms:

- $+$ is commutative and associative, i.e. $x + y = y + x$ and $x + (y + z) = (x + y) + z$.
- $(G, \to, \varepsilon)$ is a monoid, i.e. $\varepsilon \to x = x$, $x \to \varepsilon = x$ and $x \to (y \to z) = (x \to y) \to z$.
- $\to$ distributes over $+$: $x \to (y + z) = x \to y + x \to z$ and $(x + y) \to z = x \to z + y \to z$.
- Decomposition: $x \to y \to z = x \to y + x \to z + y \to z$.

Our definition of graph construction primitives in §2.1 satisfies these axioms and is therefore a valid `Graph` instance. We provide an implementation for this and other useful instances in §4. Some of them will satisfy additional axioms; for example, by making the connect operation commutative, we obtain undirected graphs.

Algebraic graphs are *complete* in the sense that it is possible to describe any graph using the core interface. Indeed, given a graph (V, E) we can construct it as `graph` V E, where the function `graph` is defined as follows.

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

Here `edges` is a generalisation of the function `edge` to a list of edges, so that `edge x y = edges [(x,y)]`:

```
edges :: Graph g => [(Vertex g, Vertex g)] -> g
edges = foldr overlay empty . map (uncurry edge)
```

The *absorption theorem* $x \to y + x + y = x \to y$, which follows from decomposition of $x \to y \to \varepsilon$, states that an edge (u, v) contains its two vertices $\{u, v\}$ and is inseparable from them. Therefore, if the pair (V, E) is inconsistent, the set of vertices of `graph` V E will be expanded to $\hat{V}$ so that $E \subseteq \hat{V} \times \hat{V}$ holds. More generally, the absorption theorem states that in addition to being complete, the algebraic graph API is also *sound* in the sense that it is impossible to construct an inconsistent pair (V, E) using the four `Graph` methods.

The following theorems can be proved from the axioms:

- Identity of $+$: $x + \varepsilon = x$.
- Idempotence of $+$: $x + x = x$.
- Absorption: $x \to y + x + y = x \to y$.
- Saturation: $x \to x \to x = x \to x$.

These theorems were verified in Agda by Alekseyev [2014] who studied the more general algebra of parameterised graphs.

3.2 Partial Order on Graphs

It is fairly standard to define $x \preceq y$ as $x + y = y$ for an idempotent operation $+$, since it gives a partial order on the elements of the algebra. Indeed, all partial order laws are satisfied:

- Reflexivity $x \preceq x$ follows from the idempotence $x + x = x$.
- Antisymmetry $x \preceq y \wedge y \preceq x \Rightarrow x = y$ holds since $x + y = y$ and $y + x = x$ imply $x = y$.
- Transitivity $x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$ can be proved as $z = y + z = (x + y) + z = x + (y + z) = x + z$.

```
vertices (h:t)  =  foldr overlay empty (map vertex (h:t))          (definition of vertices)
                =  foldr overlay empty (vertex h : map vertex t)   (definition of map)
                =  overlay (vertex h) (vertices t)                 (definition of foldr)
                ⊆  overlay (vertex h) (clique t)                   (monotony and the inductive hypothesis)
                ⊆  connect (vertex h) (clique t)                   (overlay-connect order)
                =  foldr connect empty (vertex h : map vertex t)   (definition of foldr)
                =  foldr connect empty (map vertex (h:t))          (definition of map)
                =  clique (h:t)                                    (definition of clique)
```

Figure 3. Equational reasoning with algebraic graphs.

It turns out that this definition corresponds to the *subgraph* relation, i.e. we can define:

$$x \subseteq y \stackrel{\text{def}}{=} x + y = y.$$

Indeed, expanding $x + y = y$ to $(V_x, E_x) + (V_y, E_y) = (V_y, E_y)$ gives us $V_x \cup V_y = V_y$ and $E_x \cup E_y = E_y$, which is equivalent to $V_x \subseteq V_y$ and $E_x \subseteq E_y$, as desired.

Therefore, we can check if a graph is a subgraph of another one if we know how to compare graphs for equality:

```
isSubgraphOf :: (Graph g, Eq g) => g -> g -> Bool
isSubgraphOf x y = overlay x y == y
```

The following theorems about the partial order on graphs can be proved:

- Least element: $\varepsilon \subseteq x$.
- Overlay order: $x \subseteq x + y$.
- Overlay-connect order: $x + y \subseteq x \rightarrow y$.
- Monotony: $x \subseteq y \Rightarrow$
  $(x + z \subseteq y + z) \wedge (x \rightarrow z \subseteq y \rightarrow z) \wedge (z \rightarrow x \subseteq z \rightarrow y)$.

3.3 Equational Reasoning

In this subsection we show how to use equational reasoning and the laws of the algebra to prove properties of functions on graphs. For example, to prove that `vertex x = vertices [x]` we rewrite the right-hand side using the function definitions and $x + \varepsilon = x$:

```
vertices [x]  =  foldr overlay empty (map vertex [x])
              =  foldr overlay empty [vertex x]
              =  overlay (vertex x) empty
              =  vertex x
```

Proving that `vertices xs ⊆ clique xs` requires more work. We start with the case when `xs` is the empty list `[]`, which is straightforward: `vertices [] = ε ⊆ ε = clique []`, as follows from the definition of `foldr`. If `xs` is non-empty, i.e. `xs = h:t`, we make the inductive hypothesis that `vertices t ⊆ clique t` and proceed as shown in Fig. 3.

We formally proved all properties and theorems discussed in this paper in Agda[7].

4 Graphs à la Carte

In this section we define several useful `Graph` instances, and show that the algebra presented in the previous section §3 is not restricted to directed graphs, but can be extended to axiomatically represent undirected (§4.3), reflexive (§4.4) and transitive (§4.5) graphs, their various combinations (§4.6), and even hypergraphs (§4.7).

[7] The proofs are available at https://github.com/snowleopard/alga-theory.

4.1 Binary Relation

We start by a direct encoding of the graph construction primitives defined in §2.1 into the abstract data type `Relation` isomorphic to a pair of sets (V, E), see Fig. 4. As we have seen, this implementation satisfies the axioms of the graph algebra. Furthermore, it is a *free graph* in the sense that it does not satisfy any other laws. This follows from the fact that any algebraic graph expression g can be rewritten in the following *canonical form*:

$$g = \Big(\sum_{v \in V_g} v \Big) + \Big(\sum_{(u,v) \in E_g} u \rightarrow v \Big),$$

where V_g is the set of vertices that appear in g, and $(u, v) \in E_g$ if vertices u and v appear in the left-hand and right-hand arguments of the connect operation $\rightarrow$ at least once (and should thus be connected by an edge). The canonical form of an expression g can be represented as `R` V_g E_g, and any additional law on `Relation` would therefore violate the canonicity property. The existence of the canonical form was proved by Mokhov and Khomenko [2014] for an extended version of the algebra. The proof fundamentally builds on the decomposition axiom: one can apply it repeatedly to an expression, breaking up connect sequences $x \rightarrow y \rightarrow z$ into pairs $x \rightarrow y$ until the decomposition can no longer be applied. We can then open parentheses, such as $x \rightarrow (y + z)$, using the distributivity axiom and rearrange terms into the canonical form by the commutativity and idempotence of overlay +.

It is convenient to make `Relation` an instance of the `Num` type class to use the standard + and ∗ operators as shortcuts for `overlay` and `connect`, respectively:

```
instance (Ord a, Num a) => Num (Relation a) where
    fromInteger = vertex . fromInteger
    (+)         = overlay
    (*)         = connect
    signum      = const empty
    abs         = id
    negate      = id
```

Note that the `Num` law `abs x * signum x == x` is satisfied by the above definition since $x \rightarrow \varepsilon = x$. Any `Graph` instance can be made a `Num` instance if need be, using a definition similar to the above.

We can now experiment with graphs and binary relations using the interactive GHC:

```
λ> 1 * (2 + 3) :: Relation Int
R {domain = fromList [1,2,3], relation = fromList [(1,2),(1,3)]}
λ> 1 * (2 + 3) + 2 * 3 == (clique [1..3] :: Relation Int)
True
λ> 1 * 2 == (2 * 1 :: Relation Int)
False
```

```
import           Data.Set (Set, singleton, union, elems, fromAscList)
import qualified Data.Set as Set (empty)

data Relation a = R { domain :: Set a, relation :: Set (a, a) } deriving Eq

instance Ord a => Graph (Relation a) where
    type Vertex (Relation a) = a
    empty       = R Set.empty Set.empty
    vertex  x   = R (singleton x) Set.empty
    overlay x y = R (domain x `union` domain y) (relation x `union` relation y)
    connect x y = R (domain x `union` domain y) (relation x `union` relation y `union`
        fromAscList [ (a, b) | a <- elems (domain x), b <- elems (domain y) ])
```

Figure 4. Implementing the `Graph` type class by a binary relation and the core graph construction primitives defined in §2.1.

```
λ> :t clique "abc"
clique "abc" :: (Graph g, Vertex g ~ Char) => g
λ> relation (clique "abc")
fromList [('a','b'),('a','c'),('b','c')]
```

The last example highlights the fact that the `Relation` a instance allows vertices of any type a that satisfies the `Ord` a constraint.

4.2 Deep Embedding

We can embed the core graph construction primitives into a simple data type (excuse and ignore the name clash with the type class):

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

The instance definition is a direct mapping from the *shallow embedding* of the core primitives, represented by the type class, into the corresponding *deep embedding*, represented by the data type. It is known, e.g. see Gibbons and Wu [2014], that by *folding* the data type one can always obtain the inverse mapping:

```
fold :: Graph g => Graph (Vertex g) -> g
fold Empty         = empty
fold (Vertex  x  ) = vertex x
fold (Overlay x y) = overlay (fold x) (fold y)
fold (Connect x y) = connect (fold x) (fold y)
```

We cannot use the derived `Eq` instance of the `Graph` data type, because it would clearly violate the axioms of the algebra, e.g. `Overlay Empty Empty` is structurally different from `Empty`, but they must be equal according to the axioms. One way to implement a custom law-abiding `Eq` instance is to *reinterpret* the graph expression as a binary relation, thereby gaining access to the canonical graph representation:

```
instance Ord a => Eq (Graph a) where
    x == y = fold x == (fold y :: Relation a)
```

An interesting feature of this graph instance is that it allows us to represent densely connected graphs more compactly. For example, `clique [1..n] :: Graph Int` has a linear-size representation in memory, while `clique [1..n] :: Relation Int` stores each edge separately and therefore requires $O(n^2)$ memory. Exploiting the compact graph representation for deriving algorithms that are asymptotically faster on dense graphs, compared to conventional algorithms operating on 'uncompressed' graph representations isomorphic to (V, E), is outside the scope of this paper, but is an interesting direction of future research.

4.3 Undirected Graphs

As hinted at in §3.1, to switch from directed to undirected graphs it is sufficient to add the axiom of commutativity for the connect operation. For undirected graphs we can denote connect by $\leftrightarrow$:

- $\leftrightarrow$ is commutative: $x \leftrightarrow y = y \leftrightarrow x$.

Curiously, with the introduction of this axiom, the associativity of $\leftrightarrow$ follows from the left-associated version of the decomposition axiom and the commutativity of $+$:

$$\begin{array}{lcll} (x \leftrightarrow y) \leftrightarrow z & = & x \leftrightarrow y + x \leftrightarrow z + y \leftrightarrow z & \text{(decomposition)} \\ & = & y \leftrightarrow z + y \leftrightarrow x + z \leftrightarrow x & \text{(commutativity)} \\ & = & (y \leftrightarrow z) \leftrightarrow x & \text{(decomposition)} \\ & = & x \leftrightarrow (y \leftrightarrow z) & \text{(commutativity)} \end{array}$$

Therefore, *the minimal algebraic characterisation of undirected graphs* comprises only 6 axioms:

- $+$ is commutative and associative, i.e. $x + y = y + x$ and $x + (y + z) = (x + y) + z$.
- $\leftrightarrow$ is commutative $x \leftrightarrow y = y \leftrightarrow x$ and has ε is the identity: $x \leftrightarrow \varepsilon = x$.
- Left distributivity: $x \leftrightarrow (y + z) = x \leftrightarrow y + x \leftrightarrow z$.
- Left decomposition: $(x \leftrightarrow y) \leftrightarrow z = x \leftrightarrow y + x \leftrightarrow z + y \leftrightarrow z$.

Commutativity of the connect operator forces graph expressions that differ only in the direction of edges into the same equivalence class. One can implement this by the *symmetric closure* of the underlying binary relation:

```
newtype Symmetric a = S (Relation a) deriving (Graph, Num)

instance Ord a => Eq (Symmetric a) where
    S x == S y = symmetricClosure x == symmetricClosure y
```

Note that algebraic expressions of undirected graphs have the canonical form where all edges are directed in a canonical order, e.g. according to some total order on vertices.

Let's test that the custom equality works as desired:

```
λ> clique "abcd" == (clique "dcba" :: Relation Char)
False

λ> clique "abcd" == (clique "dcba" :: Symmetric Char)
True
```

As you can see, polymorphic graph construction functions, such as `clique`, can be reused when working with undirected graphs. We can define a subclass `class Graph g => UndirectedGraph g` and use the `UndirectedGraph g` constraint for functions that rely on the commutativity of the `connect` method.

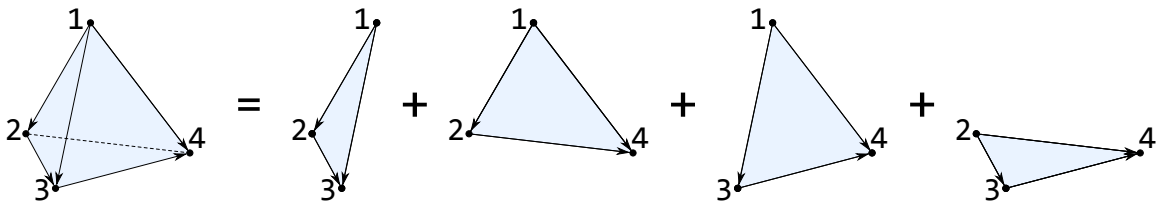


Figure 5. 3-decomposition: $1 \to 2 \to 3 \to 4 = 1 \to 2 \to 3 + 1 \to 2 \to 4 + 1 \to 3 \to 4 + 2 \to 3 \to 4$.

4.4 Reflexive Graphs

A graph is *reflexive* if every vertex of the graph is connected to itself, i.e. has a self-loop. An example of a reflexive graph is the graph corresponding to the partial order relation $\subseteq$ on graphs: indeed, $x \subseteq x$ holds for all x. To represent reflexive graphs algebraically we can introduce the following axiom:

- Self-loop: $v = v \to v$, where $v \in \mathbb{V}$ is a vertex.

The self-loop axiom corresponds to the additional `Graph` law:

- `vertex x = connect (vertex x) (vertex x)`.

One can implement the reflexive `Graph` instance analogously to the implementation of the `Symmetric` data type presented in §4.3, by wrapping the `Relation` into a `newtype` and giving it a custom `Eq` instance based on the `reflexiveClosure`.

We can define `class Graph g => ReflexiveGraph g` to increase the type safety of functions that rely on the self-loop axiom.

4.5 Transitive Graphs

In many applications graphs satisfy the *transitivity* property: if a vertex x is connected to y, and y is connected to z, then the edge between x and z can be added or removed without changing the semantics of the graph. A common example is *dependency graphs* or *partial orders* — the semantics of such graphs is typically their *transitive closure*. To describe this class of graphs algebraically we add the following *closure* axiom:

- Closure: $y \neq \varepsilon \Rightarrow x \to y + y \to z + x \to z = x \to y + y \to z$.

By using the axiom one can rewrite a graph expression into its transitive closure or, alternatively, into its *transitive reduction*, hence all graphs that differ only in the existence of some transitive edges are forced into the same equivalence class. Note that the precondition $y \neq \varepsilon$ is necessary as otherwise $+$ and $\to$ can no longer be distinguished, which is clearly undesirable:

$$x \to z = x \to \varepsilon \to z = x \to \varepsilon + \varepsilon \to z + x \to z = x \to \varepsilon + \varepsilon \to z = x + z.$$

It is interesting to note that $+$ and $\to$ have simple meanings for transitive graphs: they correspond to the *parallel* and *sequential composition*, respectively. This allows us to algebraically describe concurrent systems, which was the original motivation behind the research on algebraic graphs [Mokhov and Khomenko 2014].

We can implement transitive graphs by wrapping `Relation` in a `newtype Transitive` with a custom equality test that compares the transitive closures of the underlying relations. A subclass `class Graph g => TransitiveGraph g` can be defined to distinguish algebraic graphs with the closure axiom from others.

4.6 Preorders and Equivalence Relations

By combining reflexive and transitive graphs, one can obtain *preorders*. For example, $(1 + 2 + 3) \to (2 + 3 + 4)$ is a preorder with vertices 2 and 3 forming a *strongly-connected component*. By finding all strongly-connected components in the graph (e.g. by utilising the function `scc` from the containers library) we can derive the following *graph condensation*: $\{1\} \to \{2, 3\} \to \{4\}$. One way to interpret this preorder as a dependency graph is that tasks 2 and 3 are executed as a step, simultaneously, and that they both depend on task 1, and are prerequisite for task 4. Note that having sets as the type of graph vertices is perfectly legal: the type of the above graph condensation is `(Graph g, Vertex g ~ Set Int) => g`.

One can further combine preorders and undirected graphs, obtaining *equivalence relations*, which can be equipped with an efficient instance based on the *disjoint set* data structure [Tarjan and Van Leeuwen 1984]. One interesting application of the resulting algebra is modelling connectivity in circuits [Mokhov 2015].

4.7 Hypergraphs

As described in §4.1, the decomposition axiom collapses an algebraic graph expression into a collection of vertices and pairs of vertices (i.e. graphs). By replacing the decomposition axiom with *3-decomposition*, we obtain *hypergraphs* comprising vertices, edges and *3-edges* (triples of vertices):

- 3-decomposition: $w \to x \to y \to z =$
  $w \to x \to y + w \to x \to z + w \to y \to z + x \to y \to z$.

Fig. 5 illustrates the axiom by decomposing a tetrahedron into four 3-edges corresponding to its *faces*. To better understand the difference between the (2-)decomposition and 3-decomposition axioms, let us substitute ε for w in the 3-decomposition and simplify:

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

This is almost the 2-decomposition axiom, yet there is no way to get rid of the term $x \to y \to z$ on the right-hand side: indeed, a triple is unbreakable in this algebra, and one can only extract the pairs (edges) that are embedded in it. In fact, we can take this further and rewrite the above expression to also expose the embedded vertices:

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

Note that with 2-decomposition we can achieve something similar via the absorption theorem:

$$x \to y = x + y + x \to y.$$

This can be taken further by defining 4-decomposition and so forth, creating a hierarchy of algebraic structures corresponding to hypergraphs of different ranks.

Since every graph is also a hypergraph, we can define a superclass `class HyperGraph g => Graph g`, moving all `Graph` methods to the superclass, and leaving only the decomposition axiom in `Graph`, as the law that distinguishes it from `HyperGraph`.

```
vertices     :: Graph g => [Vertex g] -> g
clique       :: Graph g => [Vertex g] -> g
edge         :: Graph g => Vertex g -> Vertex g -> g
edges        :: Graph g => [(Vertex g, Vertex g)] -> g
graph        :: Graph g => [Vertex g] -> [(Vertex g, Vertex g)] -> g
isSubgraphOf :: (Graph g, Eq g) => g -> g -> Bool
```

(a) Derived graph construction primitives and the subgraph relation

```
path    :: Graph g => [Vertex g]             -> g
circuit :: Graph g => [Vertex g]             -> g
star    :: Graph g => Vertex g -> [Vertex g] -> g
tree    :: Graph g => Tree   (Vertex g)      -> g
forest  :: Graph g => Forest (Vertex g)      -> g
fold    :: Graph g => Graph  (Vertex g)      -> g
```

(b) Standard families of graphs and graph folding

```
transpose     :: Transpose g -> g
toList        :: ToList a -> [a]
gmap          :: Graph g => (a -> Vertex g) -> GraphFunctor a -> g
mergeVertices :: Graph g => (Vertex g -> Bool) -> Vertex g -> GraphFunctor (Vertex g) -> g
bind          :: Graph g => GraphMonad a -> (a -> g) -> g
induce        :: Graph g => (Vertex g -> Bool) -> GraphMonad (Vertex g) -> g
removeVertex  :: (Graph g, Eq (Vertex g)) => Vertex g -> GraphMonad (Vertex g) -> g
splitVertex   :: (Graph g, Eq (Vertex g)) => Vertex g -> [Vertex g] -> GraphMonad (Vertex g) -> g
removeEdge    :: (Graph g, Eq (Vertex g)) => Vertex g -> Vertex g -> RemoveEdge (Vertex g) -> g
box           :: (Graph g, Vertex g ~ (a, b)) => GraphFunctor a -> GraphFunctor b -> g
deBruijn      :: (Graph g, Vertex g ~ [a]) => Int -> [a] -> g
```

(c) Polymorphic graph manipulation

Figure 6. API of the graph construction and transformation library.

5 Graph Transformation Library

As shown in the previous section §4, the world of `Graph` instances has many inhabitants sharing a part of their 'algebraic DNA'. They all can benefit from a library of polymorphic graph construction and transformation, which we develop in this section. The API of the library is summarised in Fig. 6. The part shown in Fig. 6(a) has been defined in §3.

5.1 Standard Families of Graphs

This subsection defines a few simple functions for constructing graphs from standard graph families. See Fig. 6(b) for the list of all functions we define.

A *path* on a list of vertices can be constructed from the `edges` formed by the path neighbours:

```
path :: Graph g => [Vertex g] -> g
path []  = empty
path [x] = vertex x
path xs  = edges $ zip xs (tail xs)
```

Note that the case with a single vertex on the path requires a special treatment.

If we connect the last vertex of a path to the first one, we get a *circuit* graph, or a *cycle*. Let us express this in terms of the `path` function:

```
circuit :: Graph g => [Vertex g] -> g
circuit [] = empty
circuit xs = path (xs ++ [head xs])
```

A *star* graph can be obtained by connecting a centre vertex to a given list of leaves:

```
star :: Graph g => Vertex g -> [Vertex g] -> g
star x ys = connect (vertex x) (vertices ys)
```

Finally, *trees* and *forests* can be constructed by the following pair of mutually recursive functions:

```
tree :: Graph g => Tree (Vertex g) -> g
tree (Node r f) = star r (map rootLabel f) `overlay` forest f

forest :: Graph g => Forest (Vertex g) -> g
forest = foldr overlay empty . map tree
```

That is, a tree is represented by the root star overlaid with the forest of subtrees of the root's descendants. We remind the reader the definitions of the data types `Tree` and `Forest` from the containers library for completeness:

```
data Tree a = Node { rootLabel :: a
                   , subForest :: Forest a }

type Forest a = [Tree a]
```

Below we experiment with these functions and their properties, and define graphs `pentagon` and `p4` that will be used in subsection §5.3 and in particular will feature in Fig. 7. The helper function `edgeList` is defined as `edgeList = Set.toList . relation`.

```
λ> pentagon = circuit [1..5]
λ> p4 = path "abcd"

λ> :t pentagon
pentagon :: (Graph g, Num (Vertex g), Enum (Vertex g)) => g

λ> isSubgraphOf (path [1..5]) (pentagon :: Relation Int)
True

λ> edgeList p4
[('a','b'),('b','c'),('c','d')]

λ> t = Node 1 [Node 2 [], Node 3 [Node 4 [], Node 5 []]]
λ> edgeList (tree t)
[(1,2),(1,3),(3,4),(3,5)]

λ> p4 == (clique "abcd" :: Transitive Char)
True
```

The last property deserves a remark: the transitive closure of a path graph is the directed clique on the same set of vertices, therefore they are considered equal by the `Transitive` graph instance.

5.2 Graph Transpose

In the rest of this section we present a toolbox for transforming polymorphic graph expressions. The functions in the presented toolbox are listed in Fig. 6(c).

One of the simplest transformations one can apply to a graph is to flip the direction of all of its edges. Transpose is usually straightforward to implement but whichever data structure you use to represent graphs, you will spend at least $O(1)$ time to modify it (say, by flipping the `treatAsTransposed` flag); much more often you will have to traverse the data structure and flip every edge, resulting in $O(|V| + |E|)$ time complexity. However, by working with polymorphic graphs, i.e. graphs of type `forall g. Graph g => g`, and using Haskell's zero-cost `newtype` wrappers, we can implement transpose that takes zero time.

Consider the following `Graph` instance:

```
newtype Transpose g = T { transpose :: g } deriving Eq

instance Graph g => Graph (Transpose g) where
    type Vertex (Transpose g) = Vertex g
    empty       = T empty
    vertex      = T . vertex
    overlay x y = T $ overlay (transpose x) (transpose y)
    connect x y = T $ connect (transpose y) (transpose x)
```

That is, we wrap a graph in a `newtype` flipping the order of `connect` arguments. Let us check if this works:

```
λ> edgeList $ 1 * (2 + 3) * 4
[(1,2),(1,3),(1,4),(2,4),(3,4)]

λ> edgeList $ transpose $ 1 * (2 + 3) * 4
[(2,1),(3,1),(4,1),(4,2),(4,3)]
```

The `transpose` has zero runtime cost, because all we do is wrapping and unwrapping the `newtype`, which is guaranteed to be free or, to be more precise, is handled by GHC at compile time.

To make sure `transpose` is only applied to polymorphic graphs, we do not export the constructor `T`, therefore the only way to call `transpose` is to give it a polymorphic argument and let the type inference interpret it as a value of type `Transpose`.

5.3 Graph Functor

We now implement a function `gmap` that given a function `a -> b` and a polymorphic graph whose vertices are of type `a` will produce a polymorphic graph with vertices of type `b` by applying the function to each vertex. This is almost a `Functor` but it does not have the usual type signature, because `Graph` is not a higher-kinded type[8]:

```
newtype GraphFunctor a =
    F { gfor :: forall g. Graph g => (a -> Vertex g) -> g }

instance Graph (GraphFunctor a) where
    type Vertex (GraphFunctor a) = a
    empty       = F $ \_ -> empty
    vertex  x   = F $ \f -> vertex (f x)
    overlay x y = F $ \f -> overlay (gmap f x) (gmap f y)
    connect x y = F $ \f -> connect (gmap f x) (gmap f y)

gmap :: Graph g => (a -> Vertex g) -> GraphFunctor a -> g
gmap = flip gfor
```

[8]It is possible to define a higher-kinded version of `Graph`, but it has fewer instances.

Essentially, we are defining another `newtype` wrapper, which pushes the given function all the way towards the vertices of a given graph expression. This has no runtime cost, just as before, although the actual evaluation of the given function at each vertex will not be free, of course. Here is `gmap` in action:

```
λ> edgeList $ 1 * 2 * 3 + 4 * 5
[(1,2),(1,3),(2,3),(4,5)]

λ> edgeList $ gmap (+1) $ 1 * 2 * 3 + 4 * 5
[(2,3),(2,4),(3,4),(5,6)]
```

As you can see, we can increment the value of each vertex by mapping the function `(+1)` over the graph. The resulting expression is a polymorphic graph, as desired. Note that `gmap` satisfies the functor laws `gmap id = id` and `gmap f . gmap g = gmap (f . g)`, because it does not change the structure of the given expression and only pushes the given function down to its leaves – the vertices.

An alert reader might wonder: what happens if the function maps two different vertices into the same one? They will be merged. Merging graph vertices is a useful graph transformation, so let us define it in terms of `gmap`:

```
mergeVertices :: Graph g => (Vertex g -> Bool)
    -> Vertex g -> GraphFunctor (Vertex g) -> g
mergeVertices p v = gmap $ \u -> if p u then v else u
```

```
λ> edgeList $ mergeVertices odd 3 $ 1 * 2 * 3 + 4 * 5
[(2,3),(3,2),(3,3),(4,3)]
```

The function takes a predicate on graph vertices and a target vertex and maps all vertices satisfying the predicate into the target, thereby merging them. In our example the `odd` vertices $\{1, 3, 5\}$ are merged into 3, in particular creating the self-loop $3 \to 3$. Note: it takes linear time $O(|g|)$ for `mergeVertices` to traverse the graph and apply the predicate to each vertex (where $|g|$ is the size of the graph expression g), which may be much more efficient than merging vertices in a concrete data structure. For example, if the graph is represented by an adjacency matrix, it will likely be necessary to rebuild the resulting matrix from scratch, which takes $O(|V|^2)$ time. Since for many graphs we have $|g| = O(|V|)$, our `mergeVertices` may be quadratically faster than the matrix-based one.

As another application of `gmap`, we implement the *Cartesian graph product* operation `box`, or $G \,\square\, H$, where the resulting vertex set is $V_G \times V_H$ and vertex (x, y) is connected to vertex (x', y') if either $x = x'$ and $(y, y') \in E_H$, or $y = y'$ and $(x, x') \in E_G$. An example of the Cartesian product of graphs `pentagon` and `p4` is shown in Fig. 7.

```
box :: (Graph g, Vertex g ~ (a, b))
    => GraphFunctor a -> GraphFunctor b -> g
box x y = foldr overlay empty $ xs ++ ys
  where
    xs = map (\b -> gmap (,b) x) . toList $ gmap id y
    ys = map (\a -> gmap (a,) y) . toList $ gmap id x
```

The Cartesian product $G \,\square\, H$ is assembled by creating $|V_H|$ copies of graph G and overlaying them with $|V_G|$ copies of graph H. We get access to the list of graph vertices using `toList` and turn vertices of original graphs into pairs of vertices by `gmap`. Note that we need to *reinterpret* the input of type `GraphFunctor` as a polymorphic graph by `gmap id` before passing it to the `toList` function, which expects inputs of type `ToList`. As you can see,

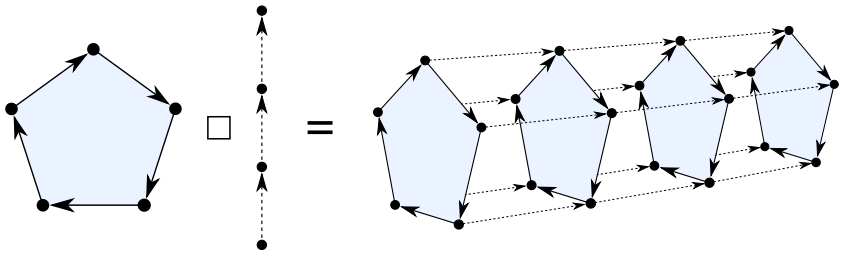


Figure 7. The Cartesian graph product of `pentagon` and `p4`.

we managed to implement quite a sophisticated graph transformation function `box` fully polymorphically. One can go further up in layers of abstraction and use `box` to construct *mesh* and *torus* graphs as `mesh xs ys = box (path xs) (path ys)` and `torus xs ys = box (circuit xs) (circuit ys)`, respectively.

The `toList` function is implemented as follows:

```
newtype ToList a = L { toList :: [a] }

instance Graph (ToList a) where
    type Vertex (ToList a) = a
    empty       = L $ []
    vertex  x   = L $ [x]
    overlay x y = L $ toList x ++ toList y
    connect x y = L $ toList x ++ toList y
```

Note that we do not provide the `Eq` instance for `ToList`, because it is impossible to make it law-abiding without requiring `Eq` for vertices, and we would like to avoid this in order to keep the `box` type signature fully parametric. As a consequence, `toList (1 + 1)` produces the list `[1,1]`.

5.4 Graph Monad

What do the operations of removing a vertex and splitting a vertex have in common? They both can be implemented by replacing each vertex of a graph with a (possibly empty) subgraph and flattening the result. You may recognise this as the `bind` operation of a monad. We implement `bind` by wrapping it into yet another `newtype`:

```
newtype GraphMonad a =
    M { bind :: forall g. Graph g => (a -> g) -> g }

instance Graph (GraphMonad a) where
    type Vertex (GraphMonad a) = a
    empty       = M $ \_ -> empty
    vertex  x   = M $ \f -> f x
    overlay x y = M $ \f -> overlay (bind x f) (bind y f)
    connect x y = M $ \f -> connect (bind x f) (bind y f)
```

The implementation is almost identical to `gmap`: instead of wrapping the value `f x` into a `vertex`, we should just leave it as is. Let us see how we can make use of this new type in our toolbox.

Firstly, we are going to implement a `filter`-like function `induce` that, given a vertex predicate and a graph, will compute the *induced subgraph* on the set of vertices that satisfy the predicate by turning all other vertices into empty subgraphs and flattening the result.

```
induce :: Graph g
    => (Vertex g -> Bool) -> GraphMonad (Vertex g) -> g
induce p g = bind g $
    \v -> if p v then vertex v else empty
```

```
λ> edgeList $ induce (<3) $ clique [0..10]
[(0,1),(0,2),(1,2)]
```

As you can see, by inducing a clique on a subset of the vertices that we like (`<3`), we get a smaller clique, as expected. The cost of `induce` for a given expression g is $O(|g|)$.

Now we can implement the `removeVertex` function:

```
removeVertex :: (Graph g, Eq (Vertex g))
    => Vertex g -> GraphMonad (Vertex g) -> g
removeVertex v = induce (/= v)
```

```
λ> edgeList $ removeVertex 2 $ 1 * 2 + 3 * 1
[(3,1)]
```

The polymorphic implementation of `removeVertex` presented above takes $O(|g|)$ to remove a vertex from a graph expression g, which is slower than some concrete graph data structures.

We can also use the `bind` function to split a vertex into a list of given vertices:

```
splitVertex :: (Graph g, Eq (Vertex g)) => Vertex g
    -> [Vertex g] -> GraphMonad (Vertex g) -> g
splitVertex v vs g = bind g $
    \u -> if u == v then vertices vs else vertex u
```

```
λ> edgeList $ splitVertex 1 [0, 1] $ 1 * (2 + 3)
[(0,2),(0,3),(1,2),(1,3)]
```

Here vertex 1 is split into a pair of vertices $\{0, 1\}$ that have the same connectivity.

5.5 Beyond Homomorphisms

Most of the `newtype` wrappers defined in this section are *homomorphisms*, that is, they preserve the structure of the original graph expression. The two exceptions are: `Transpose`, which is an *anti-homomorphism*, and `ToList` which collapses the structure of the original expression into a list.

Below we derive an implementation for `removeEdge`, which is another example of a useful function that is not a homomorphism. Removing an edge sounds easy, but the result is the most complicated `newtype` in this paper.

Here is how it works. Removing an edge (u, v) from the `empty` graph or a `vertex` is easy: nothing needs to be done, because there are no edges. To remove the edge from an `overlay`, we simply recurse to both subexpressions, because the overlay does not create any edges. The `connect` case $x \to y$ is handled by overlaying two graphs: $x_u \to y_{uv}$ and $x_{uv} \to y_v$, where:

```
newtype RemoveEdge a = RE { re :: forall g. (Vertex g ~ a, Graph g) => a -> a -> g }

instance Eq a => Graph (RemoveEdge a) where
    type Vertex (RemoveEdge a) = a
    empty       = RE $ \_ _ -> empty
    vertex  x   = RE $ \_ _ -> vertex x
    overlay x y = RE $ \u v -> overlay (re x u v) (re y u v)
    connect x y = RE $ \u v -> connect (removeVertex u $ re x u u) (re y u v) `overlay`
                               connect (re x u v) (removeVertex v $ re y v v)

removeEdge :: (Eq (Vertex g), Graph g) => Vertex g -> Vertex g -> RemoveEdge (Vertex g) -> g
removeEdge u v g = re g u v
```

Figure 8. Removing an edge from a polymorphic graph.

- x_u = `removeVertex` u x and y_{uv} = `removeEdge` u v y, thus $x_u \to y_{uv}$ definitely does not contain the edge (u, v) at the cost of losing the vertex u in the left-hand side x_u.
- y_v = `removeVertex` v y and x_{uv} = `removeEdge` u v x, thus $x_{uv} \to y_v$ definitely does not contain the edge (u, v) at the cost of losing the vertex v in the right-hand side y_v.

The overlay $x_u \to y_{uv} + x_{uv} \to y_v$ contains the vertices u and v, because at least one copy of each vertex has been preserved, but the edge (u, v) is removed in both subexpressions as intended.

We demonstrate `removeEdge` on two simple examples:

```
λ> edgeList $ path "Hello"
[('H','e'),('e','l'),('l','l'),('l','o')]

λ> edgeList $ removeEdge 'H' 'e' $ path "Hello"
[('e','l'),('l','l'),('l','o')]

λ> edgeList $ removeEdge 'l' 'l' $ path "Hello"
[('H','e'),('e','l'),('l','o')]
```

The `removeEdge` function is expensive: given an expression of size $|g|$ it may produce a transformed expression of the quadratic size $O(|g|^2)$. Many concrete **Graph** instances provide much faster equivalents of `removeEdge`.

5.6 De Bruijn Graphs

To demonstrate that one can easily construct sophisticated graphs using the presented library, let us try it on *De Bruijn graphs*, an interesting combinatorial object that frequently shows up in computer engineering and bioinformatics. The implementation is very short, but requires some explanation:

```
deBruijn :: (Graph g, Vertex g ~ [a]) => Int -> [a] -> g
deBruijn len alphabet = bind skeleton expand
  where
    overlaps = mapM (const alphabet) [2..len]
    skeleton = edges [ (Left s, Right s) | s <- overlaps ]
    expand v = vertices
        [ either ([a]++) (++[a]) v | a <- alphabet ]
```

The function builds a De Bruijn graph of dimension `len` from symbols of the given `alphabet`. The vertices of the graph are all possible words of length `len` containing symbols of the `alphabet`, and two words are connected $x \to y$ whenever x and y match after we remove the first symbol of x and the last symbol of y (equivalently, when $x = az$ and $y = zb$ for some symbols a and b). The process of construction of a 3-dimensional De Bruijn graph on the alphabet $\{0, 1\}$ is illustrated in Fig. 9. Here are all the ingredients of the solution:

- `overlaps` contains all possible words of length `len-1` that correspond to overlaps of connected vertices.
- `skeleton` contains one edge per overlap, with **Left** and **Right** vertices acting as temporary placeholders.
- We replace a vertex **Left** s with a subgraph of two vertices $\{0s, 1s\}$, i.e. the vertices whose suffix is s. Symmetrically, **Right** s is replaced by vertices $\{s0, s1\}$. This is captured by the function `expand`.
- The result is obtained by computing `bind skeleton expand`.

Below we construct the De Bruijn graph shown in Fig. 9.

```
λ> edgeList $ deBruijn 3 "01"
[("000","000"),("000","001"),("001","010"),("001","011")
,("010","100"),("010","101"),("011","110"),("011","111")
,("100","000"),("100","001"),("101","010"),("101","011")
,("110","100"),("110","101"),("111","110"),("111","111")]

λ> g = deBruijn 9 "abc"
λ> all (\(x,y) -> drop 1 x == dropEnd 1 y) $ edgeList g
True

λ> Set.size $ domain g
19683 -- i.e. 3^9

λ> Set.size $ relation g
59049 -- i.e. 3^10
```

Note that a De Bruijn graph of dimension `len` on the `alphabet` has $|\texttt{alphabet}|^{\texttt{len}}$ vertices and $|\texttt{alphabet}|^{\texttt{len}+1}$ edges.

5.7 Summary

We have presented a library of polymorphic graph construction and transformation functions that provide a flexible and elegant way to manipulate graph expressions polymorphically. Polymorphic graphs are highly reusable and composable, and can be interpreted using any of the **Graph** instances defined in §4, as well as other instances provided by the algebraic-graphs library that is available on Hackage. The library is written in the vanilla functional programming style and has no dependencies apart from core GHC libraries. Many of the presented graph transformation algorithms are expressed using familiar functional programming abstractions, such as functors and monads.

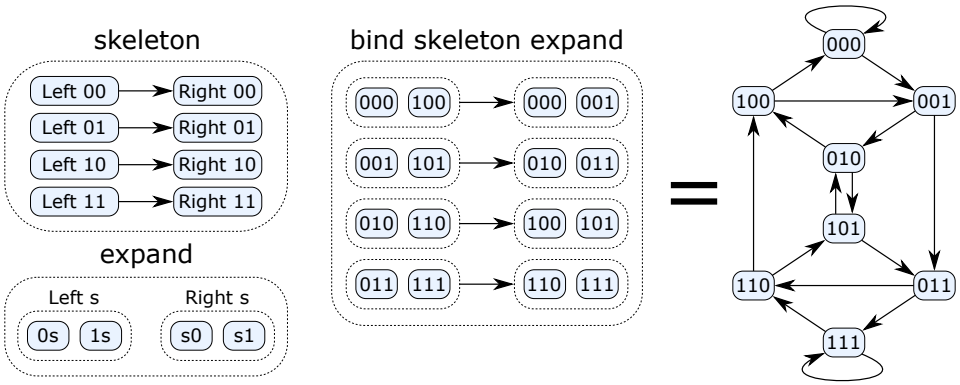


Figure 9. Constructing De Bruijn graphs using the graph monad.

6 Related Work

Historically, first approaches to graph representation in functional programming used edge lists, adjacency lists, as well as mutually recursive data structures representing cyclic graphs by the so-called 'tying the knot' approach. The former were generally slower than their imperative counterparts, while the latter were very difficult to work with. An asymptotically optimal implementation of the depth-first search algorithm developed by King and Launchbury [1995] used arrays to represent graphs and *state-transformer monads* [Launchbury and Peyton Jones 1994] to mimic imperative array updates in pure functional programming. The developed algorithms are still in use today and are available from the containers library shipped with GHC. The API of the library contains partial functions.

A fundamentally different approach by Erwig [2001] is based on *inductive graphs*, whereby a graph can be decomposed into a *context* (a node with its neighbourhood) and the rest of the graph. This inductive definition makes it possible to share common subgraphs and provides a way to implement graph algorithms in a more functional style compared to the previous approaches based on array representations. Inductive graphs are implemented in the fgl library that contains implementations of many standard graph algorithms, from depth-first search to maximum flow on weighted graphs. The library defines type classes `Graph` and `DynGraph` for working with static (unchangeable) and dynamic (changeable) graphs, comprising 10 class methods in total. Compared to algebraic graphs proposed in this paper, fgl has a larger core of graph construction primitives (10 vs 4), some of which are partial. An important advantage of fgl is the support of edge-labelled graphs.

Several other authors investigated ways to define graphs compositionally, e.g. Gibbons [1995] proposed an algebraic framework for modelling *directed acyclic graphs* comprising 6 core graph construction primitives, but the approach was not general enough to handle other practically useful classes of graphs.

Gibbon's algebra is an example of a large body of research on *categorical graph algebras*, e.g. see a survey by Selinger [2010]. These algebras are typically much more complex than the one presented in this paper[9], because they can represent graphs with *heterogeneous vertices and edges*, where not all vertices and edges are *compatible*. Graphs in this paper are *homogeneous*, i.e. an edge is allowed between any pair of vertices. This is a limitation for some applications, but it allows us to have a much simpler theory and implementation. *Petri nets* [Murata 1989] is an example of graphs where not all edges are allowed[10]. Algebraic graphs proposed in this paper cannot represent Petri nets in a safe way.

From a very different angle, simple algebraic structures, such as *semirings*, have been successfully applied to solving various path problems on graphs using functional programming, e.g. see Dolan [2013]. These approaches typically use matrix-based data structures for manipulating connectivity and distance information with the goal of solving optimisation problems on graphs, and are not suitable as an abstract interface for graph representation.

Simple graph construction cores are known for special families of graphs. For example, non-empty *series-parallel graphs* require only three primitives: a single vertex, and series and parallel composition operations. A classical result [Valdes et al. 1979] states that only *N-free graphs* can be constructed using these primitives. Similarly, the family of *cographs* corresponds to P_4*-free graphs*, which also require only three graph construction primitives: a single vertex, graph complement, and disjoint graph union [Corneil et al. 1981]. Interestingly, there is an alternative core for cographs: a single vertex, disjoint graph union, and disjoint graph join. The only difference from the core used in this paper is the disjointness requirement. By dropping this requirement, we can construct arbitrary graphs. In particular, both $N = 1 \rightarrow 2 + 3 \rightarrow (2 + 4)$ and $P_4 = 1 \rightarrow 2 + 2 \rightarrow 3 + 3 \rightarrow 4$ can be easily constructed.

This paper builds on the work by Mokhov and Khomenko [2014], where the *algebra of parameterised graphs*, a mathematical structure very similar to a semiring, was proposed as a complete and sound formalism for graph representation in the context of digital circuit design. In that paper the authors did not investigate applications of the algebra in functional programming but proved many important results that are essential for this work. Alekseyev [2014] derived a formalisation of the algebra of parameterised graphs in Agda, using an encoding similar to the core type class that we define.

7 Discussion and Future Research Opportunities

The paper presented a new algebraic foundation for working with graphs. It is particularly well-suited for functional programming

[9] As an example, *Signal Flow Graphs* [Bonchi et al. 2015] have 17 primitives and a few dozens of laws. Smaller characterisations of Signal Flow Graphs exist, however minimising the number of graph construction primitives has not (so far) been a priority for the authors (private communication with Pawel Sobocinski).

[10] Petri nets have vertices of two types, called *places* and *transitions*, and edges are only allowed between vertices of different types.

languages and benefits from functional programming abstractions, such as functors and monads. Compared to the state-of-the-art, algebraic graphs are easier to use and reuse, more compositional, and have a smaller core of only four graph construction primitives, fully characterised by an elegant algebra of graphs.

We demonstrated the flexibility of algebraic graphs by several examples and developed a Haskell library for constructing and transforming polymorphic graphs.

The presented approach has a few important limitations:

- This paper has not addressed edge-labelled graphs. In particular, there is no known extension of the presented algebra characterising graphs with arbitrary vertex and edge labels. However, Mokhov and Khomenko [2014] give an algebraic characterisation for graphs labelled with Boolean functions, which can be generalised to labels that form a semiring.
  We found that one can represent edge-labelled graphs by functions from labels to graphs. For example, a *finite automaton* can be thought of as a collection of graphs, one for each symbol of the alphabet:

  ```
  type Automaton a s = a -> Relation s
  ```

  Here `a` and `s` stand for the alphabet and the set of states of the automaton, respectively. This representation of labelled graphs is supported by the following graph instance:

  ```
  instance Graph g => Graph (a -> g) where
      type Vertex (a -> g) = Vertex g
      empty       = pure empty
      vertex      = pure . vertex
      overlay x y = overlay <$> x <*> y
      connect x y = connect <$> x <*> y
  ```

  Therefore, `Automaton a s` is a valid `Graph` instance.
- As mentioned in §6, the presented approach is designed for homogeneous graphs, where an edge is allowed between any pair of vertices. It is an open research question whether it is possible to extend algebraic graphs for modelling heterogeneous graphs, such as Petri nets, without sacrificing the simplicity of the algebraic core.
- Many graph instances, e.g. `Relation`, incur a logarithmic overhead during graph construction, and may therefore be unsuitable for high-performance applications. One possible solution is to operate on deeply-embedded algebraic graphs (such as `data Graph`), and perform conversions to more conventional representations only when necessary.
- There are no known efficient implementations of fundamental graph algorithms, such as depth-first search, that work directly on the algebraic core. Therefore, we need to translate core expressions to conventional graph representations, such as adjacency lists, and utilise existing graph libraries, which may be suboptimal for certain algorithmic problems.

Despite these limitations, algebraic graphs have been successfully used in the design of processor microcontrollers [Mokhov and Khomenko 2014] and asynchronous circuits [Beaumont et al. 2015].

Our future research will focus on addressing the above limitations, and on the exploration of the following topics:

- Algebraic graph expressions can be minimised via the *modular decomposition* of graphs [McConnell and De Montgolfier 2005], thereby reducing their memory footprint, as well as speeding up their processing. Modular decomposition is a canonical graph representation, which can therefore be used to efficiently compare algebraic graph expressions for equality. Exploiting the compactness of algebraic graphs in algorithms is a promising research direction.
- By using the algebraic approach to graph representation one can formulate graph algorithms in the form of solving systems of algebraic equations with unknowns. This may potentially open way to the discovery of novel graph algorithms.

Acknowledgments

I would like to thank Arseniy Alekseyev and Neil Mitchell for numerous discussions on algebraic graphs, their advice, criticism and encouragement. Without their help this work would have likely remained an unfinished toy project. Simon Peyton Jones, Brent Yorgey, Danil Sokolov, Ben Lippmeier, Ulan Degenbaev and several anonymous reviewers provided constructive feedback on an earlier draft of this paper, helping to substantially improve it. Last but not least, I'm very grateful to Victor Khomenko for his contribution to the algebra of parameterised graphs that forms the mathematical foundation of this work.

References

Arseniy Alekseyev. 2014. *Compositional approach to design of digital circuits.* Ph.D. Dissertation. Newcastle University.

Jonathan Beaumont, Andrey Mokhov, Danil Sokolov, and Alex Yakovlev. 2015. Compositional design of asynchronous circuits from behavioural concepts. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 118–127.

Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. 2015. Full abstraction for signal flow graphs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 515–526.

Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type synonyms. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 241–253.

Derek G Corneil, H Lerchs, and L Stewart Burlingham. 1981. Complement reducible graphs. *Discrete Applied Mathematics* 3, 3 (1981), 163–174.

Stephen Dolan. 2013. Fun with semirings: a functional pearl on the abuse of linear algebra. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 101–110.

Martin Erwig. 2001. Inductive graphs and functional graph algorithms. *Journal of Functional Programming* 11, 05 (2001), 467–492.

Jeremy Gibbons. 1995. An initial-algebra approach to directed acyclic graphs. In *International Conference on Mathematics of Program Construction*. Springer, 282–303.

Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *ACM SIGPLAN Notices*, Vol. 49. ACM, 339–347.

Jonathan S Golan. 1999. *Semirings and their Applications.* Springer Science & Business Media.

Frank Harary. 1969. *Graph theory.* Addison-Wesley.

David J King and John Launchbury. 1995. Structuring depth-first search algorithms in Haskell. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 344–354.

John Launchbury and Simon L Peyton Jones. 1994. Lazy functional state threads. In *ACM SIGPLAN Notices*, Vol. 29. ACM, 24–35.

Saunders Mac Lane and Garrett Birkhoff. 1999. *Algebra.* Chelsea Publishing Company.

Ross M McConnell and Fabien De Montgolfier. 2005. Linear-time modular decomposition of directed graphs. *Discrete Applied Mathematics* 145, 2 (2005), 198–209.

Andrey Mokhov. 2015. Algebra of switching networks. *IET Computers & Digital Techniques* (2015).

Andrey Mokhov and Victor Khomenko. 2014. Algebra of Parameterised Graphs. *ACM Transactions on Embedded Computing Systems* 13, 4s (2014), 1–22.

Tadao Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580.

Peter Selinger. 2010. A survey of graphical languages for monoidal categories. In *New structures for physics*. Springer, 289–355.

Robert E Tarjan and Jan Van Leeuwen. 1984. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)* 31, 2 (1984), 245–281.

Jacobo Valdes, Robert E Tarjan, and Eugene L Lawler. 1979. The recognition of series parallel digraphs. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*. ACM, 1–12.