Algebraic Graphs with Class

Christoph Madlener
madlener@in.tum.de

May 10, 2021

Abstract

We give an overview of algebraic graphs as introduced by Mokhov [8]. Algebraic graphs have a small and safe core of construction primitives, alongside a set of axioms, characterizing an algebra of graphs. They can be used to elegantly implement a graph transformation library employing functional programming. We also review the suitability of algebraic graphs for formal verification using Isabelle/HOL.

1 Introduction

Graphs are a fundamental structure studied in depth by both mathematicians and computer scientists alike. One very common definition states that a (directed) graph is a tuple $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is the set of edges. While this is a perfectly valid and natural mathematical definition it is not necessarily suitable for implementation. This can be illustrated by trying to directly translate this to Haskell:

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

The value `G [1,2,3] [(1,2),(2,3)]` then represents the graph $G = (\{1, 2, 3\}, \{(1, 2), (2, 3)\})$, however `G [1,2] [(2,3)]` does not represent a consistent graph, as the edge refers to a non-existent node. The state-of-the-art `containers` library implements graphs with adjacency lists employing immutable arrays [6]. The consistency condition $E \subseteq V \times V$ is not checked statically though, which can lead to runtime errors. `fgl`, another popular Haskell graph library uses inductive graphs [4], also exhibiting partial functions and potential for runtime errors due to the violation of consistency.

This lead Mokhov to conceive *algebraic graphs* [8], a sound and complete representation for graphs. They abstract away from graph representation details and characterize graphs by a set of axioms. Algebraic graphs have a small safe core of graph construction primitives and are suitable for implementing graph transformations. These primitives are represented in the following datatype:

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

`Empty` constructs the empty graph, `Vertex` a graph with a single vertex and no edges. `Overlay` essentially is the union of two graphs and `Connect` additionally adds edges from all vertices from one graph to all vertices of the other.

Alongside proper definitions for these primitives we will see in section 2 that this is indeed a sound and complete graph representation. We will also cover the algebraic structure (2.2) exhibited by these graphs and elegant graph transformations (2.3) based on a type class for the core. In subsection 2.4 we explore how we can exploit the algebraic structure in verification using Isabelle/HOL.

In section 3 we will examine the deep embedding (i.e. the datatype as opposed to the type class of the preceding sections) regarding applicability in practice and in verification (subsection 3.1).

In section 4 we conclude and give an outlook on possible future work and open questions connected to algebraic graphs as presented.

2 A type class for algebraic graphs

This section will cover most parts of the original paper by Mokhov [8]. The graph construction primitives of algebraic graphs will be formally defined and their soundness and completeness proven. Afterwards we will cover the algebraic structure exhibited by these primitives when interpreted as graphs. As part of this we will introduce a type class in Haskell which further abstracts from implementation details. We will also see that different classes of graphs can be obtained by extending the set of axioms. Instances of these different classes and a graph transformation library in Haskell are provided. To conclude this section we will explore the suitability of this type class for formal verification in Isabelle/HOL.

2.1 The Core

In the introduction we already briefly saw the four graph construction primitives. Let us first define the set of (consistent) graphs $\mathcal{G}$ over a fixed universe of vertices $\mathbb{V}$. A graph $G \in \mathcal{G}$ can be represented as a tuple $G = (V, E)$ where $V \subseteq \mathbb{V}$ and $E \subseteq V \times V$. The four graph construction primitives then are defined as follows. The *empty* graph, denoted ε is simply the tuple $(\emptyset, \emptyset)$ which is clearly consistent, hence $\varepsilon \in \mathcal{G}$. Graphs with a single *vertex* $v \in \mathbb{V}$ and no edges, i.e. $(\{v\}, \emptyset)$ denoted by just v, are also trivially consistent, hence $v \in \mathcal{G}$. The binary operations *overlay* and *connect*, denoted by $\oplus$ and $\rightarrow$ respectively, allow constructing larger graphs. Their definitions are

$$\begin{aligned}(V_1, E_1) \oplus (V_2, E_2) &\coloneqq (V_1 \cup V_2, E_1 \cup E_2)\\(V_1, E_1) \rightarrow (V_2, E_2) &\coloneqq (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)\end{aligned}$$

So overlay is just the union of two graphs, while connect also adds an edge from each vertex of its first argument to each vertex of its second argument. It is straightforward to see that if $G_1 \in \mathcal{G}$ and $G_2 \in \mathcal{G}$ then also $G_1 \oplus G_2 \in \mathcal{G}$ and $G_1 \rightarrow G_2 \in \mathcal{G}$. In combination with the consistency of ε and of single vertex graphs this already proves that algebraic graphs are a consistent representation for graphs. The completeness of this representation, i.e. for each $G \in \mathcal{G}$ there is an algebraic graph expression representing it, is shown in the next subsection. Both of these facts were also formally proven in Isabelle/HOL (see subsection 2.4).

As we will shortly see in subsection 2.2 overlay and connect are somewhat related to addition and multiplication, hence, we give connect a higher precedence than overlay. Let us consider some examples to develop some intuition for the primitives. They are illustrated in Figure 1.

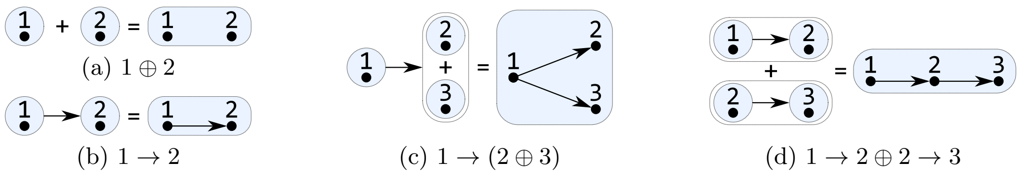

(a) $1 \oplus 2$ (b) $1 \to 2$ (c) $1 \to (2 \oplus 3)$ (d) $1 \to 2 \oplus 2 \to 3$

Figure 1: Examples of graph construction. The overlay and connect operations are denoted by $\oplus$ and $\to$, respectively. Illustrations from [8].

(a) $1 \oplus 2 = (\{1, 2\}, \emptyset)$

(b) $1 \to 2 = (\{1, 2\}, \{(1, 2)\})$

(c) $1 \to (2 \oplus 3) = (\{1, 2, 3\}, \{(1, 2), (1, 3)\})$

(d) $1 \to 2 \oplus 2 \to 3 = (\{1, 2, 3\}, \{(1, 2), (2, 3)\})$

We already saw how to deeply embed these graph construction primitives in a datatype. For greater reusability the following type class can be defined.

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

The associated type `Vertex g` represents the universe of vertices $\mathbb{V}$, the remainder of the type class corresponds to ε, single vertex graphs, $\oplus$ resp. $\to$.

At this point we will only introduce some very basic functions for constructing graphs. More advanced constructions and transformations will be given in subsection 2.3. A graph with a single edge is obtained by simply connecting two vertices:

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

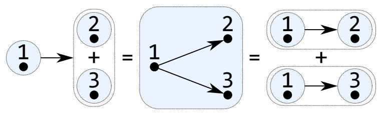
A graph with only isolated vertices can be constructed from a list of vertices as follows:

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

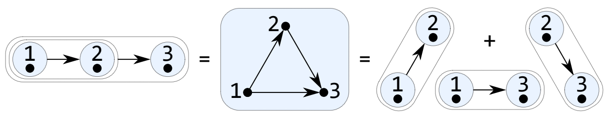
Replacing `overlay` with `connect` in `vertices` leads to a (directed) clique. Later we will also consider undirected graphs. In that context this function actually builds a fully connected graph on the given list of vertices.

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

Following and extending this approach leads to a safe and fully polymorphic graph transformation library for any concrete graph representation (e.g. graphs in `containers` or `fgl`) instantiating this class.



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



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

Figure 2: Two axioms of the algebra of graphs. Illustrations from [8].

2.2 Algebraic Structure

Before turning to more advanced graph constructions we will observe the algebraic structure of the primitives. It turns out that overlay and connect almost form a semiring over graphs:

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

The shared identity and the missing annihilating zero for $\to$ (i.e. $x \to 0 = 0$) are the only difference. There is also the following *decomposition law*, as illustrated in Figure 2(b).

$$x \to y \to z = x \to y \oplus x \to z \oplus y \to z$$

Using decomposition we can actually prove some of the above properties, leading to the following minimal set of axioms characterizing directed graphs:

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

One can easily check that the definitions of the primitives given in subsection 2.1 satisfy these axioms. At this point let us consider the completeness of algebraic graphs. For any graph $G = (V, E)$ the following function builds it using the four construction primitives:

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

The `edges` function generalizes `edge` to a list of edges:

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

They are a direct translation of the fact, that any algebraic graph expression g for a graph $G = (V, E) \in \mathcal{G}$ can be rewritten in the *canonical form*:

$$g = \left(\bigoplus_{v \in V} v\right) \oplus \left(\bigoplus_{(u,v) \in E} u \to v\right)$$

The completeness of algebraic graphs was also formally proven in Isabelle/HOL (see subsection 2.4).

Mokhov also follows a standard approach of defining a partial order ($\preceq$) for the idempotent overlay operation as $x \preceq y \iff x \oplus y = y$. This is indeed a partial order under the graph axioms (i.e. it is reflexive, antisymmetric and transitive). In fact this partial order can actually be used as a definition for the subgraph relation, denoted by $\subseteq$:

$$x \subseteq y \coloneqq x \preceq y \iff x \oplus y = y$$

So far we only considered directed graphs. Undirected graphs can be obtained from the very same construction primitives as directed graphs. It suffices to modify the underlying axioms. In the case of undirected graphs this can be achieved by making connect commutative (then denoted by $\leftrightarrow$), i.e. adding the axiom $x \leftrightarrow y = y \leftrightarrow x$. By introducing this axiom we can prove some of the other axioms of directed graphs. Hence, undirected graphs are characterized by the following minimal set of axioms:

- $(\mathcal{G}, \oplus)$ is a commutative semigroup
- $\leftrightarrow$ is commutative and has ε as the identity
- left distributivity: $x \leftrightarrow (y \oplus z) = x \leftrightarrow y \oplus x \leftrightarrow z$
- left decomposition: $(x \leftrightarrow y) \leftrightarrow z = x \leftrightarrow y \oplus x \leftrightarrow z \oplus y \leftrightarrow z$

By choosing different additional axioms, other classes of graphs can be characterized. Some examples are reflexive graphs (i.e. each vertex has a self-loop), transitive graphs (as in dependency graphs), and the combination thereof; preorders. Combining undirected graphs and reflexive graphs yields equivalence relations. It is even possible to define hypergraphs of arbitrary (but fixed) order by replacing the decomposition law with another suitable axiom [8].

In summary the algebraic approach to graph representation has a small and safe core of construction primitives. Pairing these primitives with suitable sets of axioms leads to a highly flexible representation for graphs.

2.3 Graph Transformation Library

Based on the core type class, Mokhov implements what he refers to as a graph transformation library[1]. In order to understand the following constructions and transformations better, we follow his approach and first define instances for the type class `Graph`. Recall the attempt from the introduction to directly translate the graph representation $G = (V, E)$ into the Haskell datatype `G`. We will now define a `Graph` instance for that datatype, while realizing that more generally it can be used to represent relations on some domain (hence the renaming to `Relation`).

```haskell
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
```

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

```
  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) ])
```

Mokhov also defines a Num instance for Relation which enables the use of + and ∗ as shortcuts for overlay resp. connect.

The given Graph instance for Relation satisfies the axioms of directed graphs, as was argued in subsection 2.1. Obtaining Graph instances which satisfy the axioms for undirected graphs (or other classes of graphs) is also straightforward: One can simply wrap Relation in a newtype and define a custom Eq instance. An example for undirected graphs (in this case the connection to symmetric directed graphs is very prominent) is given.

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

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

It is also sensible to introduce subclasses like class Graph g => UndirectedGraph g to reflect the class of graph we are working with in the type system, and thus, increase type safety.

Equipped with these two instances let us now actually consider graph constructions and transformations made possible with the core type class. Simple examples include constructing a *path* from a list of vertices, which then can be easily extended to a *circuit*:

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

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

Mokhov also gives functions for constructing *star* graphs (i.e. a center vertex connected to a set of leaves) and for building graphs from Trees and Forests from the containers library.

More interesting applications include the implementation of a zero time graph transposition using yet another newtype wrapper [8]:

```
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)
```

Although for certain representations it is possible to get an $\mathcal{O}(1)$ transpose operation (e.g. by setting a transposed flag), for many typical representations the whole graph has to be traversed, resulting in $\mathcal{O}(|V| + |E|)$ time.

More sophisticated transformations can be implemented using the following `Functor`-like `newtype` wrapper. The function `gmap` takes a function `a -> b` and a graph with vertices of type `a` and produces a graph with vertices of type `b`.

```
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
```

Interestingly this can be used to merge vertices fulfilling some predicate `p` by mapping them to a single vertex:

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

In the original paper `gmap` is then used further to construct rather sophisticated graphs fully polymorphically [8].

Mokhov also goes on to introduce a `Monad`-like `newtype` wrapper. The `bind` function in the context of graphs allows to replace each vertex with a (possibly empty) subgraph. This can be used to both remove and split vertices.

```
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)
```

Removing a vertex with `bind` can be generalized to a function `induce`, which only keeps vertices satisfying some predicate `p`.

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

Splitting a vertex works by replacing that vertex with a graph containing the desired 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
```

Mokhov then defines yet another `newtype` wrapper utilizing `bind` to remove edges. He also showcases the construction capabilities of the presented library by constructing de Bruijn graphs (a rather sophisticated type of graph occurring frequently in computer engineering and bioinformatics) completely polymorphically. These definitions are omitted here for brevity.

The presented material on the given type class gives an overview of an elegant, reusable and polymorphic graph construction and transformation library. Note that in the available `hackage` library for algebraic graphs the `newtype` wrappers for graph transformation are not implemented as such. Instead there is a higher-kinded type class which is based on `MonadPlus`, but has fewer instances [8].

2.4 Verification

We have now seen how to implement a polymorphic graph transformation library in Haskell using the core type class. Mokhov proposes that algebraic graphs and their axiomatic characterization are suitable for formal verification as they enable equational reasoning. Some simple statements (like `vertices xs` $\subseteq$ `clique xs`) are given accompanied by an Agda formalization[2]. For the examples in that formalization the claim obviously holds.

To experiment further we first formalized the available material in Isabelle/HOL [10]. This formalization[3] involves the definition of a *locale*, which for the purposes of this paper can be thought of as a type class which can be equipped with axioms. Proving the statements from the paper in Isabelle/HOL works without problems.

Noschinski formalized fundamental aspects of graph theory in Isabelle/HOL [11]. Part of that is essentially the standard representation of a graph as a tuple $G = (V, E)$ (i.e. a locale with one single axiom ensuring consistency). Instantiating a locale in Isabelle/HOL (which can be likened to instantiating a type class in Haskell) requires to prove the axioms attached to it. Using the definitions from subsection 2.1 for the primitives it is straightforward to instantiate the algebraic graph locale for the tuple representation. We also formally prove that the graph construction primitives only produce consistent graphs. A proof for the completeness of the primitives is given as well (in the sense that every graph represented by a tuple can be expressed as an algebraic graph).

As the statements in the original paper were of very limited scope, we define and reason about more *advanced* graph theoretic concepts: We introduce (vertex) walks and the notion of reachability on algebraic graphs. First impressions suggest that it is feasible to base more sophisticated formalizations of graph theory on algebraic graphs. Although we suspect that some additional axiom(s) may be required to make for *interesting* graph theory; as it stands the statement $\varepsilon \neq$ `vertex u` for some $u \in \mathbb{V}$ is unprovable. This is further illustrated by the fact that we can instantiate the algebraic graph locale with the unit type.

This is an interesting direction for future work for a related reason that the type class is interesting in Haskell: The type class forms the basis for a polymorphic graph transformation library, whereas the locale can be seen as a *polymorphic* graph formalization library. Any lemma we can prove using the locale axioms becomes available to any graph representation instantiating the locale. This obviously becomes more interesting the more material is available in the locale. Haslbeck et al. chose a similar approach to implement and verify Kruskal's algorithm independent of the concrete graph representation [5].

[2]available at `https://github.com/snowleopard/alga-theory`
[3]available at `https://github.com/cmadlener/algebraic_graphs`

3 Deep Embedding

Up until this point we were mainly concerned with exploiting the algebraic structure of graphs for polymorphic graph constructions and transformations, and for formal verification. However, the deep embedding of the construction primitives in a datatype is interesting in itself. Recall the definition from the introduction:

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

An instance for the type class can be defined easily, however, we cannot use the derived Eq instance for the datatype, as it would violate the graph axioms. One solution is to fold the deep embedding into a concrete graph representation to obtain a law-abiding Eq instance:

```
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)

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

This approach is also currently implemented in the library available on `hackage` (although an adjacency map is used instead of a relation). However, this is not ideal for multiple reasons. Coupling the datatype to some other graph representation merely for equality checks is inconvenient at best. More interestingly algebraic graphs can compactly represent dense graphs. Consider the graph `clique [1..n]`, which has a linear size representation in the deep embedding, while it is quadratic when stored as a `Relation`. Requiring to fully expand graphs to check equality voids this advantage. A promising approach to alleviate this issue is *modular graph decomposition* [7], as it allows to minimize algebraic graph expressions. It remains an open question if this decomposition can be computed efficiently directly on the deep embedding.

More interesting questions for future work come up in connection with the deep embedding. Currently only the transformations presented with the type class, i.e. transpose, as well as anything that can be done using the corresponding versions of `gmap` and `bind`, are implemented directly on the deep embedding. No known efficient implementations of fundamental graph algorithms - like depth-first-search - that work directly on the deep embedding are known [8]. The current library implementation therefore resorts to converting algebraic graph expressions to an adjacency map and reuses existing algorithms on this representation.

An immediate application which can be more efficient in space while also being competitive regarding time complexity using algebraic graphs that comes to mind is search. However, this only applies in best case examples, i.e. where only a small part of the graph has to be expanded. This is due to the fact that the neighborhood of a vertex in an algebraic graph expression can be computed in linear (in the size of the graph expression) time and space. We already saw that the algebraic graph representation can be significantly smaller than conventional representations.

Benchmarks (on an earlier version of the library) were performed by Moine to compare its performance to `containers` and `fgl` (and one other representation). The results are mixed; depending on the area of application, algebraic graphs offer a competitive alternative, while in other cases, they perform orders of magnitudes worse. For the full results refer to the repository containing the benchmark suite[4].

3.1 Verification

We were also interested in using the deep embedding in formal verification using Isabelle/HOL. Hence, we once more formalized large parts of the material on the deep embedding from the Haskell library. This feat turned out to be very convenient, as the inductive datatype definition of the deep embedding enables proofs by induction. Many of the resulting subgoals were then handled automatically by Isabelle.

In the Haskell implementation of the deep embedding we already saw that structural equality does not satisfy the axioms. There we can define a custom `Eq` instance to obtain a law-abiding datatype. In Isabelle/HOL we can follow a similar approach, which results in a `quotient_type`, which can be thought of as an equivalence class construction on the raw datatype. For the moment the equivalence is based on conversion of algebraic graph expressions to the tuple representation by Noschinski mentioned in subsection 2.4. Again, the minimization of algebraic graph expressions to obtain an equivalence relation on algebraic graphs, decoupled from any other graph representation, is very much desirable.

The quotient type also allows us to formally prove that it satisfies the axioms, by instantiating the locale corresponding to the core type class. It also paves the way for further formalization efforts on the deep embedding. However, this also requires more work to become truly interesting, i.e. if algorithms that work directly on the deep embedding or other advanced usages of it can be found (or put rather bluntly, if there is more material to formalize).

4 Conclusion

We saw a novel approach to representing graphs, which is especially suitable for functional programming. It allows for safe, polymorphic and elegant graph constructions and transformations. Algebraic graphs also lend themselves nicely to formal verification.

The alert reader might have noticed that the given primitives can only be used to represent unlabeled graphs. Mokhov extended the algebraic graph library to also represent edge-labeled graphs. There is no published material on this yet (besides the source code), for more information refer to the talk by Mokhov during Haskell eXchange 2018[5].

Algebraic graphs as presented have been successfully used in practice, e.g. for asynchronous circuit design [1] and acceleration of graph processing using FPGAs [9]. In these examples the available transformations are sufficient. An important direction for future research is going beyond those (relatively simple) transformations. That means to find and formulate algorithms either inside the core type class or directly on the deep embedding. Berghammer et. al [2] successfully employ algebraic approaches to formally verify graph algorithms. However, they are using an algebra of relations as opposed to the one here defined on graph construction primitives. Dolan [3] uses an algebra on adjacency matrices (specifically a semiring) to find paths in graphs. Mokhov himself postulates that the algebraic approach allows to formulate graph algorithms as systems of equations with unknowns. This may also be an approach for finding novel graph algorithms.

[4]available at `https://github.com/haskell-perf/graphs`

[5]available at `https://skillsmatter.com/skillscasts/12361-labelled-algebraic-graphs`

References

[1] Jonathan Beaumont et al. “High-level asynchronous concepts at the interface between analog and digital worlds”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (2017), pp. 61–74.

[2] Rudolf Berghammer et al. “Relational characterisations of paths”. In: *Journal of Logical and Algebraic Methods in Programming* 117 (2020), p. 100590.

[3] Stephen Dolan. “Fun with semirings: a functional pearl on the abuse of linear algebra”. In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming.* 2013, pp. 101–110.

[4] Martin Erwig. “Inductive graphs and functional graph algorithms”. In: *Journal of Functional Programming* 11.5 (2001), pp. 467–492.

[5] Maximilian P.L. Haslbeck, Peter Lammich, and Julian Biendarra. “Kruskal’s Algorithm for Minimum Spanning Forest”. In: *Archive of Formal Proofs* (Feb. 2019). `https://isa-afp.org/entries/Kruskal.html`, Formal proof development. ISSN: 2150-914x.

[6] David J. King and John Launchbury. “Structuring Depth-First Search Algorithms in Haskell”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 344–354. ISBN: 0897916921. DOI: `10.1145/199448.199530`. URL: `https://doi.org/10.1145/199448.199530`.

[7] Ross M McConnell and Fabien De Montgolfier. “Linear-time modular decomposition of directed graphs”. In: *Discrete Applied Mathematics* 145.2 (2005), pp. 198–209.

[8] Andrey Mokhov. “Algebraic graphs with class (functional pearl)”. In: *ACM SIGPLAN Notices* 52.10 (2017), pp. 2–13.

[9] Andrey Mokhov et al. “Language and hardware acceleration backend for graph processing”. In: *Languages, Design Methods, and Tools for Electronic System Design.* Springer, 2019, pp. 71–88.

[10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Vol. 2283. LNCS. Springer, 2002.

[11] Lars Noschinski. “Graph Theory”. In: *Archive of Formal Proofs* (Apr. 2013). `https://isa-afp.org/entries/Graph_Theory.html`, Formal proof development. ISSN: 2150-914x.