Algebraic Graphs with Class

Christoph Madlener
madlener@in.tum.de

May 4, 2021

Abstract

[9]

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 [7]. 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 [5], also exhibiting partial functions and potential for runtime errors due to the violation of consistency.

This lead Mokhov to conceive *algebraic graphs* [9], 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. In subsection 3.1 we define a quotient type in Isabelle/HOL which in essence is a deep embedding which satisfies the axioms.

2 Type class/locale

2.1 The Core

- graph construction primitives
- abstract from datatype → type class
- fundamental functions (vertex, edge, etc.)

2.2 Algebraic Structure

- axioms (for digraphs)
- subgraph - partial ordering
- axioms for undirected graphs, other classes
- instances in Haskell (directed, undirected, etc. via Eq instance)

2.3 Graph Transformation Library

- graph families (path, circuit, etc.)
- transpose
- functor
- monad
- removing edges

2.4 Verification

- equational reasoning
- reasoning in Isabelle/HOL $\rightarrow$ locale
  - "advanced" graph concepts (walks, reachability, SCCs, etc.)
    - more experimentation needed for actual feasibility (additional axioms required for e.g. $Vertex\ u \neq \epsilon$?)
    - first impression: similar as with other representations; abstraction to more advanced concepts quickly hides representation
  - "polymorphic" lemmas
  - compare to Kruskal AFP entry [6] (possibility to get algorithms for any graph representation which can instantiate locale)
  - instantiation with pair_digraph [11] - soundness and completeness proof

3 Deep Embedding

- compact representation
- compare to different concrete/executable representations wrt. efficiency in time and space
  - sparse graphs (edge list)
  - adjacency map
- performance (haskell-perf)
- reasoning in Isabelle/HOL $\rightarrow$ induction is great
- algorithms directly on deep embedding? $\rightarrow$ future work/open question (Haskell implementation converts to adjacency map for algorithms)

3.1 Quotient Type

- deep embedding which satisfies axioms
- additional lemmas
- minimization [8]
  - interesting for quotient type, disconnect from Noschinski/any other representation for equality, same applies in Haskell for Eq instance)
  - interesting for algorithms (assuming we can exploit compact representation)

4 Conclusion

- summarize findings
- restate open questions, future work
- labeled graphs (outlook)
- related work: other algebraic approaches (semiring on matrices [4], relational algebra [2])
- usage examples [10, 1]

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] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[4] 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.

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

[6] 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.

[7] 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`.

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

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

[10] 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.

[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.