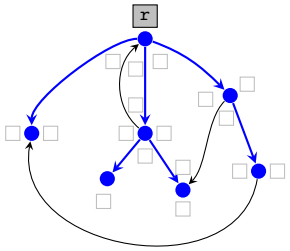
A Datatype of Planar Graphs

Malin Altenmüller and Conor McBride

University of Strathclyde, Glasgow, United Kingdom

Introduction Planar graphs are subject of interest not only in graph theory [7, 6], but also in defining syntaxes for monoidal categories [10] with specific topological properties, for example diagrams for quantum circuits where crossing wires is a non-trivial operation [4]. We present work on an intrinsically typed data structure of planar graphs, implemented in Agda.
A graph is planar if it can be drawn on a sphere without any edges crossing. We are working at the level of the drawing, so graphs are actually plane graph embeddings. A graph embedding is uniquely defined by an edge ordering around each of its vertices, called a rotation system [5].

Graphs are Decorated Trees The graphs we describe are connected, and may contain self-loops at a vertex as well as multiple edges in parallel. We define graphs inductively by using one of their spanning trees as a skeleton and storing the remaining edges alongside [2]. Our graphs have labelled edges, but also contain data in *sectors*, which are regions near vertices, subdivided by their incident edges. Sectors are also the places within a graph we can point at, or move to. The example on the right shows the spanning tree of the graph in blue, and its sectors as boxes, with a specially marked *root* sector. The data structure represents the traversal of a graph in order, starting from the root, working clockwise round the spanning tree, following tree edges and passing non-tree edges on the way. Arrowheads indicate the order of traversal (the graph itself is undirected).



At each `Step` in the traversal we either encounter the next `sector` or an edge; these two always alternate. When visiting a tree edge we ask for the subtree attached to it. The first time we meet each non-tree edge, we push its label onto a *stack*, popping when we find its other end. A `Step` is indexed by the stack before and after, as well as an indicator of whether we expect a `sector` (of type `S`) or an `edge` (of type `E`) at our next encounter. The stack is a list (with `,-` being cons), and `span`ning subtrees are the reflexive transitive closure `Star` of the `Step` relation.

```
data Step : (List E × SE) → (List E × SE) → Set where
  sector : S → Step (es , sec)(es , edg)
  push   : (e : E) → Step (     es , edg)(e ,- es , sec)
  pop    : (e : E) → Step (e ,- es , edg)(     es , sec)
  span   : E → V → Star Step (as , sec)(bs , edg)
                 →      Step (as , edg)(bs , sec)
```

All together, a plane graph consists of a vertex and a `Star Step ([] , sec)([] , sec)`: a clockwise traversal around that vertex, starting from an empty stack and expecting the root sector, visiting the rest of the tree, then finishing with an empty stack back at the root sector.

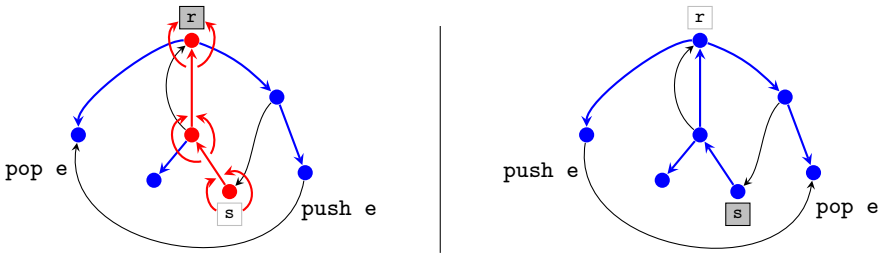
Proposition 1. *A spanning tree together with a stack of additional edges defines a plane graph.*

Proof Sketch. A spanning tree is plane by definition, thus the topological property is established by the treatment of non-tree edges. By maintaining the stack in order of traversal, edges are

popped in reverse order to being pushed, so no two edges cross. Effectively, we contract the spanning tree to a single vertex with only self-loops, i.e. the non-tree edges. The graph is planar when these self-loops form a well bracketed word [2]. □

There is more information yet in the stack indices: each edge e is boundary to a region of the graph. Anything on top of e on the stack is *local* to the region, and once we have popped e we know we have left the region. Observing the stack alone gives clues about where we currently are in the graph, and which edges are local to the current subgraph.

Zippers The first graph operation we want is focussing to a specific position [1], then moving the graph's root to this position. We use zippers [8] for graphs in a bifunctor representation [9], allowing us to transform sector data as we traverse clockwise.
Standard zippers for trees construct a path through the structure by successively choosing one branch to move along, while storing its siblings alongside. In the case of graphs, siblings could be trees but also stack operations. Each layer in the path stores a forwards list of steps ahead in the tree and a backwards list of steps behind (i.e. between the arrival point and the current position). The zipper itself is a sequence of layers surrounding the sector in focus. The left example shows in red the path from sector `s` back to the root. At each vertex along the path the curved arrows depict the backwards and forwards sibling lists:



Re-rooting a graph to a zipper's sector-in-focus involves rotating the traversal order of the spanning tree while keeping track of the direction of the non-tree edges.

Proposition 2. *Re-rooting a planar graph returns a planar graph.*

Proof Sketch. The spanning tree is traversed in a different order, but structurally unchanged. Some of the additional edges have to be turned in the process of moving the root. In the original graph, these edges were pushed before we arrive at the new root, and popped after, they were exactly the index of the new root. When re-rooting, each of these edges will change its direction and the order of edges on the stack will be reversed, thus planarity is maintained. □

In the left example, the stack operations for edge `e` are explicitly marked. The stack at segment `s` is `(e ,- [])`. The right example show the result of the re-rooting operation, with `s` now being the root (with index `[]`), and the stack operations of `e` interchanged.

A Context Comonad We define graphs as containers. Sectors are places for data, as well as places to view the graph from. We obtain a context comonad [11] whose counit projects the root sector data and whose comultiplication *decorates* each sector with the graph re-rooted to that sector. The graph stays the same, with the order of edges around vertices fixed, but we redirect its spanning tree, depending on which sector is root. In future work we intend to decorate each push and pop with the graph obtained by following their non-tree edge, allowing convenient but read-only graph traversal [3].

References

[1] Michael Gordon Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. ∂ for data: Differentiating data structures. *Fundam. Informaticae*, 65(1-2):1–28, 2005.

[2] Olivier Bernardi. Bijective counting of tree-rooted maps and shuffles of parenthesis systems, 2006.

[3] Richard S. Bird. On building cyclic and shared structures in haskell. *Formal Aspects Comput.*, 24(4-6):609–621, 2012.

[4] Bob Coecke, Ross Duncan, Aleks Kissinger, and Quanlong Wang. *Generalised Compositional Theories and Diagrammatic Reasoning*, pages 309–366. Springer Netherlands, Dordrecht, 2016.

[5] John Robert Edmonds Jr. *A combinatorial representation for oriented polyhedral surfaces.* PhD thesis, 1960.

[6] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[7] Jonathan L Gross and Thomas W Tucker. *Topological graph theory.* Courier Corporation, 2001.

[8] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[9] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 287–295. ACM, 2008.

[10] Peter Selinger. A survey of graphical languages for monoidal categories. *Lecture Notes in Physics*, page 289–355, 2010.

[11] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. In Jirí Adámek and Clemens Kupke, editors, *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008, Budapest, Hungary, April 4-6, 2008*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 263–284. Elsevier, 2008.