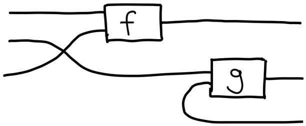
A data type of intrinsically plane graphs

Malin Altenmüller

BCTCS 2025

String diagrams (1)

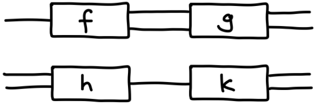
- Interested in monoidal categories with
  - sequential composition: $f \fatsemi g$
  - parallel composition: $f \otimes g$.
- Nice graphical syntax of string diagrams:



String diagrams (2)

- Properties of the category translate to its diagrams,
  e.g. symmetric vs. braided monoidal categories:

- Some equations hold automatically,
  e.g. interchange law $(f \otimes h) \fatsemi (g \otimes k) = (f \fatsemi g) \otimes (h \fatsemi k)$:



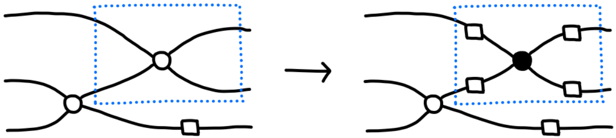
Why graphs?

- Formalise string diagrams and their rewriting theory.

Definition

A graph G is a tuple (V, E, s, t) with a set of vertices V, a set of edges E, source and target functions $s, t : E \to V$.

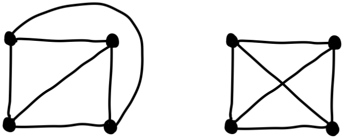
- Rewriting theory for string diagrams becomes graph rewriting:



Why plane graphs?

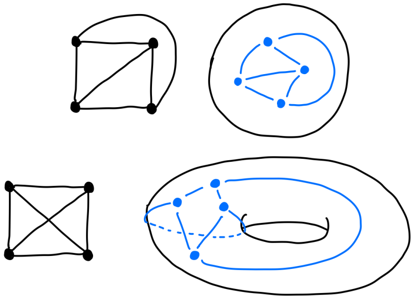
- Monoidal categories with specific *topological* properties: no crossing wires allowed!
- Generalisation of symmetric and braided monoidal categories.
- Certain theories do not come with a builtin SWAP operation.

Graphs are not suitable, we need plane graphs!



Surface-embeddings of graphs

- Drawing of a graph onto a surface (without edges crossing):



- A surface-embedding is characterised by its *faces*.

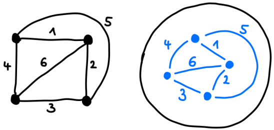
Rotation systems

= order of edges around each vertex.

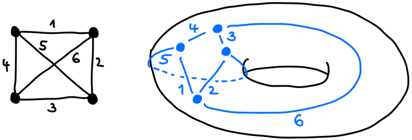
Theorem

A rotation systems determines a graph's surface-embedding.

Plane graph:



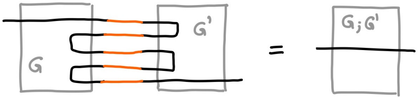
Toroidal graph:



Plane graphs as a data type?

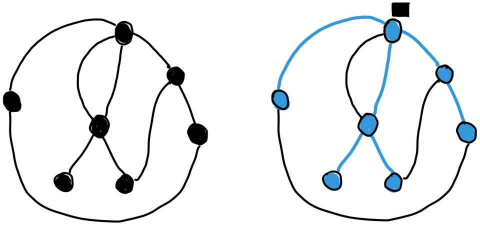
Goal: implementation of plane graphs and their rewriting theory in Agda

- Composition is really nice on paper, but not in a term based tool:



- Graphs are cyclic, but we would like an inductive type.
- How to enforce the planarity?

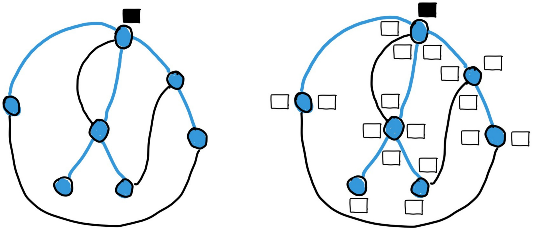
Spanning trees to the rescue



graph = spanning tree (incl. root) + non-tree edges

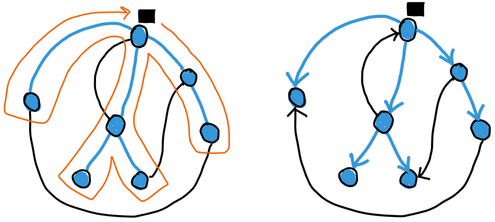
An inductive data type

graph = spanning tree (incl. root) + non-tree edges + corners



An ordered data type

- A graph is the clockwise traversal of its spanning tree:



- Edge set E is split into tree edges and non-tree edges.

Indexing type

Lemma

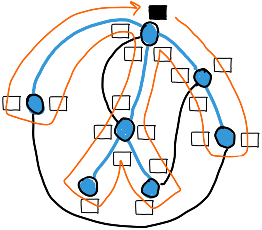
In a clockwise traversal, corners and edges always alternate.

- Store this information in a simple data type:

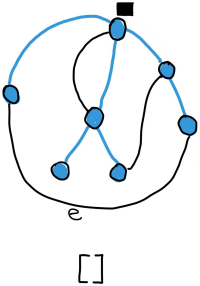
```
data Next : Set where
  edge   : Next
  corner : Next
```

- Traversal of the tree is guided by an indexing type:

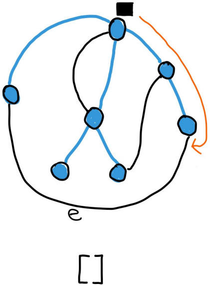
```
TravTy : Set
TravTy = List E × Next
```



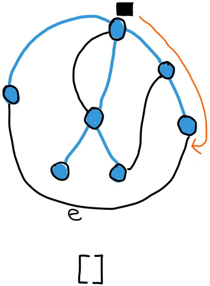
A stack of non-tree edges

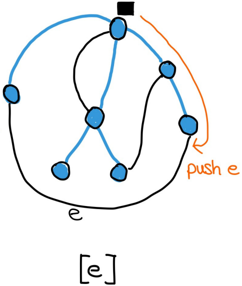


A stack of non-tree edges

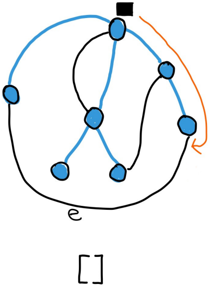


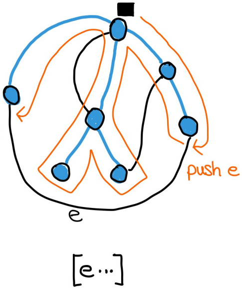
A stack of non-tree edges



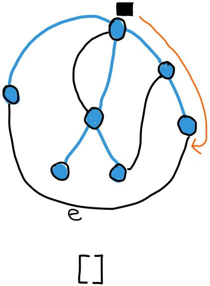


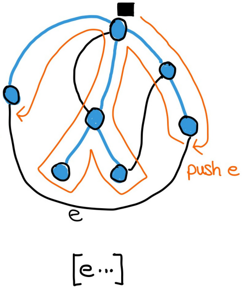
A stack of non-tree edges

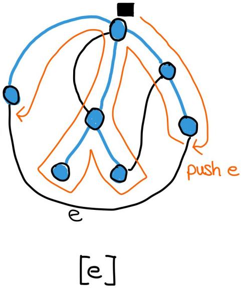




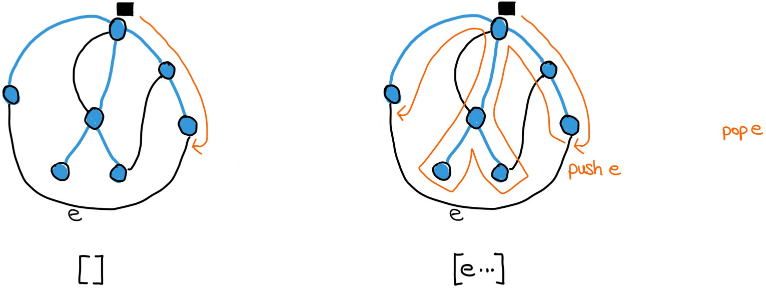
A stack of non-tree edges



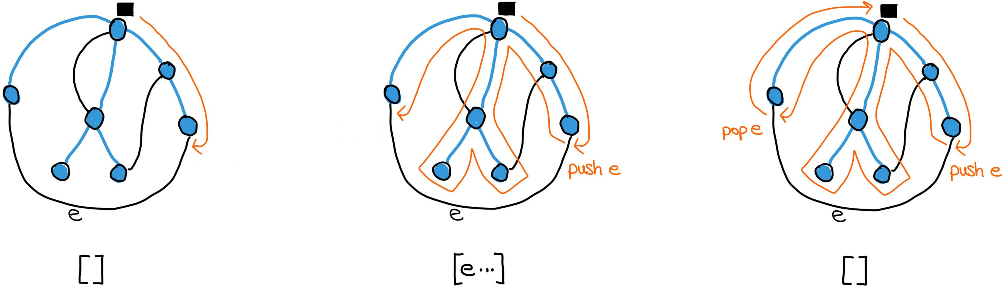




A stack of non-tree edges

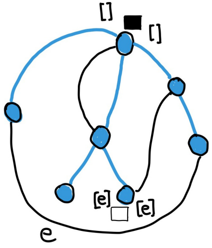


A stack of non-tree edges



Indexing type – example

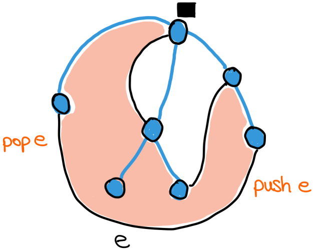
- Every corner is indexed by a stack of edges characterising its face:



- A plane graph has index ([ ] , corner) ([ ] , corner).

Stack structure determines faces

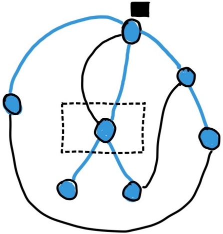
- Every non-tree edges closes a face of the graph embedding:

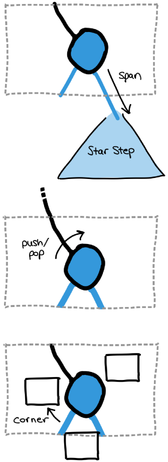


- We can calculate the faces of the embedding by observing the changes of the edge stack.

Possible steps in the traversal

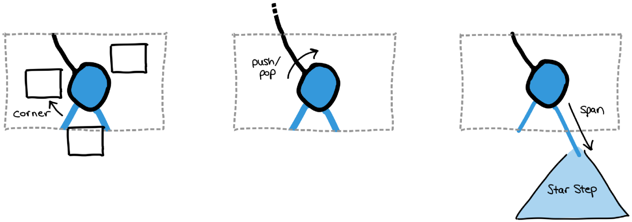
One step in the clockwise traversal of the spanning tree:





The type of steps

$$\begin{array}{l}
\textsf{data Step} : \textsf{TravTy} \to \textsf{TravTy} \to \textsf{Set where} \\
\quad \textsf{corner} : (c : C) \to \textsf{Step}\ (es\ ,\ \textsf{corner})\ (es\ ,\ \textsf{edge}) \\
\quad \textsf{push} \quad : (e : E) \to \textsf{Step}\ (es\ ,\ \textsf{edge})\ (e\ ,\text{-}\ es\ ,\ \textsf{corner}) \\
\quad \textsf{pop} \quad\ : (e : E) \to \textsf{Step}\ (e\ ,\text{-}\ es\ ,\ \textsf{edge})\ (es\ ,\ \textsf{corner}) \\
\quad \textsf{span} \quad : (e : E)\ (v : V) \to \textsf{Star Step}\ (es\ ,\ \textsf{corner})\ (es'\ ,\ \textsf{edge}) \to \textsf{Step}\ (es\ ,\ \textsf{edge})\ (es'\ ,\ \textsf{corner})
\end{array}$$



A Graph is a sequence of steps: Star Step ([ ] , corner) ([ ] , corner).

Planarity

Theorem

A stack of non-tree edges ensures planarity of a graph.

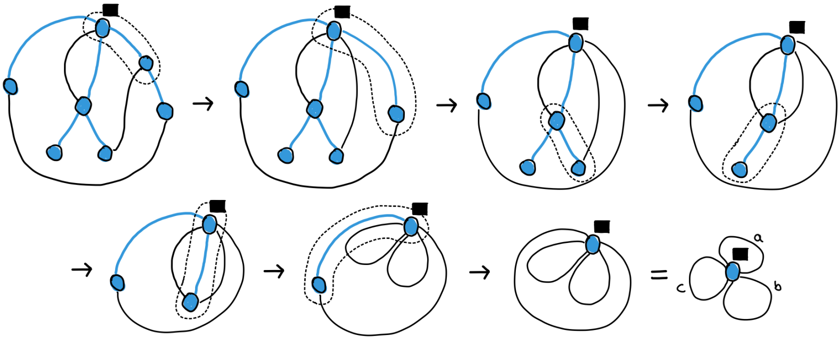
To prove this, we use the following fact:

Lemma

Contracting a plane subgraph does not change the genus of a graph's embedding.

- Plan: contract the entire spanning tree of a graph.
- All the surface information is stored in the non-tree edges of a graph.

Contracting the spanning tree



Non-tree edges form a well bracketed word `abbcca`.
(cf. context-free grammars, Dyck language,. . .)

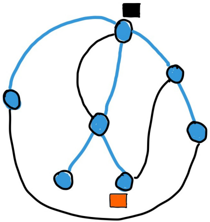
Zippers[1] for graphs

- Structure to focus on a sector in the graph.
- Useful to to highlight a certain subgraph (and rewrite it).
- Zipper = path to the focus + sibling structures alongside it.
- Store the path bottom-up: fast access to nearby elements.
- Mimic a cursor structure: forwards/backwards lists everywhere.

[1] Huet, "The Zipper".

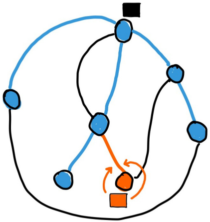
Zipper example

- Start at the focus:



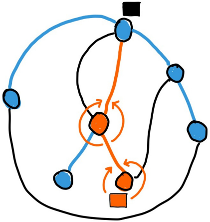
Zipper example

- Move up along the path one step at a time:



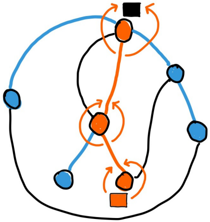
Zipper example

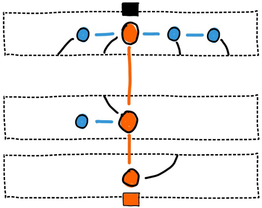
- Move up along the path one step at a time:



Zipper example

- Full path defines a *layer* structure:



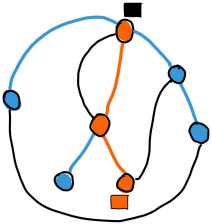


- Continue using the stack structure to ensure planarity:

```
record ZipTy : Set where
  field ahead : List E
        here : Next
        behind : List E
```

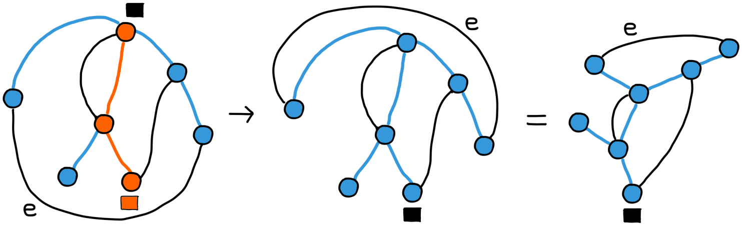
Re-rooting the tree

- Start from a zipper of a graph.
- Idea: move the spanning tree's root to the sector in focus:



- This changes the order of traversal of the spanning tree.

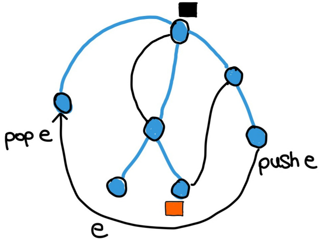
Goal: turn the tree upside down

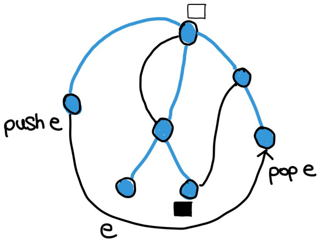


- Compute the new traversal order: edge stack structure has to change.

Turn non-tree edges

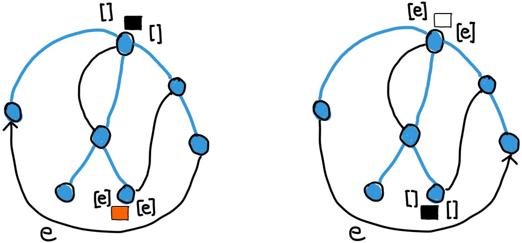
Edge e has to be turned around in the re-rooting operation,...





Turn non-tree edges

…therefore the indices at the root and focus are exchanged:



Theorem

Re-rooting preserves planarity.

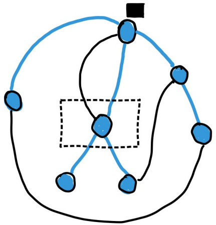
Proof: by very careful turning of non-tree edges during the operation.

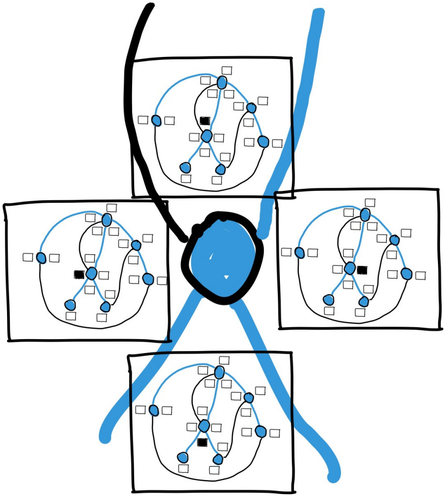
Making planarity intrinsic

- Planarity is part of the data type of graphs.
- Any element of this type is by definition plane.
- Any operation defined on this type preserves planarity by definition.
- Use it to implement rewriting of subgraphs (planarity preserving).

More ideas (1)

Equip corners with data: the graph re-rooted to here.
This gives a context comonad[2].

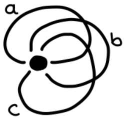




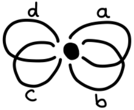
[2] Uustalu and Vene, "Comonadic Notions of Computation".

More ideas (2)

What about different surfaces from the plane?
Higher genus surfaces?
Non-orientable surfaces?
What to use instead of a stack?



(valid and non-valid embedding on the torus →)



Thank you for your attention!

A DATA TYPE OF INTRINSICALLY PLANE GRAPHS

Malin Altenmüller

malin.altenmuller@ed.ac.uk

Huet, Gérard P. "The Zipper". In: *J. Funct. Program.* 7.5 (1997), pp. 549–554. URL: http://journals.cambridge.org/action/displayAbstract?aid=44121.

Uustalu, Tarmo and Varmo Vene. "Comonadic Notions of Computation". In: *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008, Budapest, Hungary, April 4-6, 2008*. Ed. by Jirí Adámek and Clemens Kupke. Vol. 203. Electronic Notes in Theoretical Computer Science 5. Elsevier, 2008, pp. 263–284. DOI: 10.1016/j.entcs.2008.05.029. URL: https://doi.org/10.1016/j.entcs.2008.05.029.