A Data Type of Intrinsically Plane Graphs in Agda

Malin Altenmüller
University of Edinburgh, UK

Conor Titania Mc Bride
University of Strathclyde, Glasgow, UK
Quantinuum, Cambridge, UK

Abstract

This work develops a suitable data type for plane graph embeddings in Agda. Graphs are used as combinatorial representations for string diagrams, a graphical calculus for monoidal categories. Whenever a monoidal theory does not include any symmetry or braiding operations, it describes processes that are sensitive to their topology. To encode this information in the graphical language, we have to consider surface-embeddings of graphs. We study the simplest case, plane graphs, and present their implementation in Agda. We overcome issues like the cyclic nature of a graph by using one of its spanning trees as an underlying inductive structure. The graphs we implement are plane by construction and any operation on them is guaranteed to preserve this planarity. Additionally, we present a notion of focussing on a certain subgraph within a graph. This operation is crucial for the application of local rewrite rules which themselves are at the centre of diagrammatic reasoning in monoidal categories.

2012 ACM Subject Classification Theory of computation → Type theory; Mathematics of computing → Graphs and surfaces; Theory of computation → Rewrite systems

Keywords and phrases planar graph, spanning tree, dependent types, graph rewriting

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.13

Supplementary Material *Software*: `https://maltenmuller.github.io/thesis/src/Index.html`

Funding *Malin Altenmüller*: funded by EPSRC grant number EP/X025551/1.

Acknowledgements We would like to thank the anonymous reviewers for helpful suggestions.

1 Introduction

To model diagrammatic reasoning for string diagrams [26], a combinatorial representation is needed that can accommodate diagram rewriting. In the standard case of symmetric monoidal categories this is typically achieved by using (hyper-)graphs and their rewriting [22, 9, 7]. Here, only the *connectivity* information between edges and vertices matter. *Non-symmetric* monoidal categories have received little attention to date, even though they have important applications. Theories without symmetry are crucial for modelling of quantum circuits [10, 8] in which swapping two qubits is an expensive operation and thus cannot be represented by an implicit changing of wires. They also describe a larger class of theories as they contain the braided and symmetric cases. String diagrams for non-symmetric monoidal categories have to encode an additional *topological* component as the arrangement of edges matters. Consequently, for their combinatorial presentation we have to consider graphs embedded on surfaces. For this work, we are interested in *plane* graphs which are those that can be drawn onto the surface of a sphere without any edges crossing.

We present a library for plane graphs and their rewriting in Agda. We will use Agda's rich type system to encode the planarity property as part of a graph's type. A graph of this type is guaranteed to be plane and any operation on graphs must preserve the planarity property. We solve the challenge of graphs being a cyclic structure by using one of their

spanning trees as underlying inductive structure and store the remaining information on top. Instead of a standard definition of a surface-embeddings as additional information on top of a graph, we work with intrinsically plane graphs directly.

Surface-embedded graphs. We start by recalling some definitions and terminology on graphs and their surface-embeddings. More details can be found in the literature [16, 24].

▶ **Definition 1.** *A* directed graph *G consists of a set V of* vertices*, a set E of* edges*, and two functions $s, t : E \to V$,* source *and* target*, respectively. A graph is called* total *(or* closed*), if s and t are total functions, and* partial *(or* open*) otherwise. An edge e is called* incident *to both its source and target vertices, and vice versa.*

A *planar* graph can be embedded into the surface of a sphere (and, equivalently, into the plane), and the corresponding concrete embedding is called a *plane* graph.

▶ **Definition 2.** *The* embedding *of a graph G into a surface S is a drawing of G onto S: Vertices are mapped to points of the surface, and edges to simple arcs, connecting two vertices each. As no two arcs share any point other than their endpoints, there are no crossing edges in a graph embedding. The* faces *of a graph embedding are regions of the surface enclosed by edge cycles of the graph.*

(a) Plane embedding. **(b)** Toroidal embedding.

Figure 1 Two different surface embeddings of the same graph.

Surface-embeddings of graphs are determined by the arrangement of their faces on the surface. We will use a traversal of a graph's spanning tree to encode this information, but the following alternative presentation provides useful intuition:

▶ **Definition 3.** *A* rotation *for a vertex v in a graph G is a cyclic lists* $\mathsf{inc}(v)$ *of its* incident *edges of type $E \times \{\mathsf{src}, \mathsf{tgt}\}$. (e, src) occurs in v's rotation if and only if* $\mathsf{src}(e) = v$ *(and respectively for* tgt*). A* rotation system *for G consists of a rotation for each of its vertices.*

Every rotation system for a connected graph G induces a unique (up to surface homeomorphism and orientation-preserving topological equivalence) embedding of G into an oriented surface, and vice versa [19, 11, 16].

▶ Remark 4. Throughout the formalisation we consider all graphs to be *closed*, meaning that both ends of every edge are connected to a vertex. We can represent open graphs by using *boundary vertices* which are auxiliary structures that represent the "outside" face of a graph [3, 2] and all input and output edges of an open graph are attached to its boundary vertex, see Figure 2. We can add topological information to these interface edges via the rotation at the boundary vertex.

▶ Remark 5. All graphs are considered *finite* and *connected* in this work. Additionally, we formalise *undirected* graphs only as directed edges do not affect topological considerations. Thus, instead of two different functions src and tgt as the data of a graph, we consider a function $\mathsf{end} : E \to V \times V$ which encodes the two endpoints for each edge.

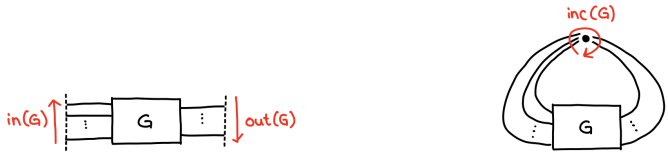


Figure 2 Introduction of an auxiliary boundary vertex to capture input and output edges of a graph G. The interface edges together form the rotation $\mathsf{inc}(G)$ at the boundary vertex.

Agda implementation. We use Agda's literate mode to display type checked code throughout this document: all code blocks are extracts from an active buffer. In addition, we include links to the corresponding blocks in the source code with the ✿ symbol, clickable in the PDF version of the document. When a bound variable is not explicitly quantified by a function definition, it assumed to be brought into scope as a module parameter or variable statement.

1.1 Related work

Graphs in functional languages. Representing cyclic and shared structures in a functional programming language has been the subject of various research. A number of projects use the technique of splitting a structure into a non-cyclic, inductive part and the remaining, potentially cyclic, information on top [6, 13, 27, 18], a similar approach to our work on plane graphs. Another approach focus on detecting cycles when defining operations on graphs, for example by marking nodes during an operation [12] or by using the features of a lazy functional language to unroll cycles a finite number of times only [21]. A different line of work suggests certain algebraic structures to represent graphs with a small set of generators and combinators [25], which are used to express properties of graphs and algorithms on them.

Implementations of plane graphs. The particular case of a representation of plane graphs is an important feature in the Rocq [5] proof of the Four Colour Theorem [15, 14]. This proof uses *hypermaps* in which the connection points between edges and vertices in the original graph are encoded as vertices and their relative position is encoded by relations on them. This presentation specifies an spatial arrangement of the faces of the graph embedding, similar to the information in a rotation system from which planarity can be checked. In our implementation we present the spatial arrangement of the graph more directly as an ordered traversal. We are therefore able to specify a data type of graphs that is intrinsically plane, thus the planarity property never has to be checked explicitly but holds by definition.

2 Plane Graphs in Agda

Typically, graph embeddings are defined as graphs together with additional structure, specifying a certain surface-embedding. This approach is unsuitable in the environment of Agda: Graphs are unordered data structures represented by sets and functions, but implementing unordered sets in Agda is a non-trivial task. It involves explicitly quotienting an inductive type, adding a layer of complexity. In contrast, surface-embeddings of graphs already come with an ordering on vertices and edges. In fact, it is exactly this order that turns a graph into a graph embedding. Therefore, instead of equipping graphs with their surface embedding, we will use graph embeddings as the first class objects straight away.

2.1 Graphs are cyclic structures

Another challenge for implementing graphs in Agda is their *cyclic* nature. Iterating a graph may unroll its cycles and continue indefinitely. In finite graphs we may encounter cycles, too, but we will eventually *revisit* nodes and edges in a traversal. These structures are on the one hand finite (as their closed representation contains finite sets only) but also infinite (as their traversal may continue forever). Infinite data types which have a finite representation have been studied as rational fixpoints of functors [23]. We would like to compare this approach to our work more formally in the future.

For our construction of a suitable representation of plane graphs, we will split a graph into two parts. The inductive part is a finite, cycle-free, connected, maximal subgraph which we can represent and manipulate in Agda easily. Alongside this structure, we store a graph's *looping* edges which form its cycles.

▶ **Observation 6.** *We observe the following properties about trees and graphs:*
1. *A tree is a connected, cycle-free graph, together with a choice of root.*
2. *Any maximal subtree of a graph G is a spanning tree of G.*

We take a graph G's spanning tree as suitable maximal, non-cyclic subgraph. Any spanning tree includes all of G's vertices, and some of its edges which we call *spanning* edges. The remaining information are all other edges of G, which we call *looping* edges. This leads to our encoding of graphs as *over-connected* trees. A tree is very straight-forward to represent as an inductive type in Agda. We then add the additional edges to the spanning tree at the relevant positions, see Figure 4a for an example.

▶ Remark 7. Throughout this work, we assume a choice of spanning tree and root for any given graph, which can be calculated for a given graph by standard graph algorithms. In traditional diagram rewriting, we have a canonical choice of root by taking the first input parameter of a system as its root. A potential extension to our implementation are equivalence classes of spanning trees for the same graph.

▶ Remark 8. The development of graphs as overconnected trees is similar to the notion of *blossoming trees* [1], which are trees equipped with additional half-edges at some vertices. Connecting these half-edges in a non-crossing manner constructs the faces of a plane graph embedding. A related structure are shuffles of parenthesis systems [4], a standard representation of plane graphs in combinatorics.

2.2 The order of edges matters

We now explain why careful inspection of the looping edges in a graph is enough to determine one of its surface-embeddings. Even though we do not implement rotation systems directly in this work, the ordering of edges is still a key characteristic of our construction. The translation from the spanning tree representation to rotation systems is straight-forward, which we demonstrate in Section 2.5. We observe the following properties about surface-embedded graphs and their plane subgraphs.

▶ **Definition 9.** *Given a graph $G = (V, E, s, t)$ and an edge $e \in E$ with $s(e) = u$ and $t(e) = v$, $v \neq u$,* edge contraction *is a function that maps G to $G' = (V', E', s', t')$ where:*
- $V' = V \setminus \{u\}$, $E' = E \setminus \{e\}$,
- $s'(e') = \begin{cases} v & \text{if } s(e') = u \\ s(e') & \text{otherwise} \end{cases}$, $t'(e') = \begin{cases} v & \text{if } t(e') = u \\ t(e') & \text{otherwise} \end{cases}$.

We write G/e for the graph obtained by contracting e; if H is a subgraph of G then G/H is the graph obtained by contracting all the edges of H.

(a) before contracting edge e. **(b)** result of edge contraction.

Figure 3 Edge contraction of edge e identifies its source and target vertices.

▶ **Example 10.** If the graph G contains another edge e' with the same source and target vertices as e, $s(e') = u$, $t(e') = v$, then e' forms a self-loop at v after the contraction of e.

▶ **Lemma 11.** *Let G be a surface-embedded graph; then:*

- *The order of repeated edge contractions does not matter: for any edges $e1, e2 \in E_G$, we have $G/e_1/e_2 \cong G/e_2/e_1$.*
- *For any edge e, the edge contraction G/e embeds in the same surface.*
- *The edge contraction of any plane subgraph of G embeds in the same surface.*

As a tree is trivially a plane graph, we can apply Lemma 11 and observe that contracting a graph's entire spanning tree does not change its genus. Correspondingly, a graph's genus is encoded by its looping edges alone.

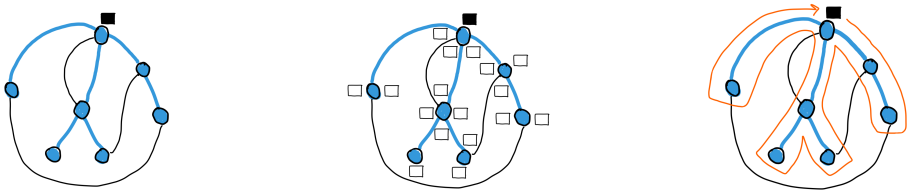
▶ **Corollary 12.** *The structure of a graph's looping edges alone determine its genus.*

As a graph's spanning tree contains all vertices of a graph, its contraction yields a graph containing a single vertex, with all the looping edges attached to it:

▶ **Definition 13.** *A* bouquet graph *is a graph consisting of one vertex v and a set of edges E with $s(e) = t(e) = v$ for all edges $e \in E$. A rotation system for a bouquet graph is a single list in which each edge appears exactly twice.*

We have observed that the structure of looping edges is key for specifying a graph's embedding. We therefore have record them in a certain order in our data type of plane graphs. This order corresponds to the rotation at the central vertex of the bouquet graph when contracting the graph's entire spanning tree.

Traversing a graph's spanning tree. We will record the occurrence of looping edges alongside a *clockwise* traversal of the graph's spanning tree. An example of this traversal operation is shown in Figure 4c. A spanning tree traversal moves through the entire graph and visits all possible *positions* inside it. Positions in a graph are assigned to its vertices and occur between any two adjacent edges. They are called *corners*, see Figure 4b.



(a) Spanning tree (thicker edges). **(b)** All corners in the graph. **(c)** Traversal of the spanning tree.

Figure 4 Spanning tree representation of an example graph.

▶ **Definition 14.** *The* corners *of a vertex $v \in V$ of graph $G = (V, E)$ are subdivisions of its neighbourhood, separated by the edges incident at v. The corners of all vertices $v \in V$ are the corners of the whole graph G. The* root *of a graph's spanning tree is a distinguished corner.*

▶ Remark 15. In case a vertex v has no incident edges, v's only corner represents its entire neighbourhood. Otherwise, a corner specifies a triple (e_1, v, e_2) of a vertex $v \in V$ and two adjacent edges $e_1, e_2 \in E$ at v.

We now specify a graph traversal operation as a traversal of the spanning tree in Algorithm 1, see Figure 4c for an illustration. The algorithm visits all the corners in the graph in a specific order, defined by the structure of the spanning tree. Note that only one of the potential steps moves the traversal along a spanning edge. The other two options of steps in the algorithm record additional information, either a corner or a looping edge.

Algorithm 1 Graph traversal.

This procedure defines the traversal of a graph by using its spanning tree. Starting from the root corner, the traversal steps through elements around vertices (edges and corners) in a clockwise order until it reaches the root again. Assume the current position of the traversal to be the corner $c1$ at vertex $v1$. The traversal can progress by one of three possible steps:

1. Following a spanning edge e: this operation progresses the traversal from vertex $v1$ to vertex $v2$ which is adjacent to $v1$ by e. The traversal then continues recursively at $v2$.
2. Passing a corner: this operation recognises a corner and progresses the traversal at the same vertex $v1$ to the next element in clockwise order.
3. Recording a looping edge: this operation passes a looping edge by storing it in a separate data structure (more details about this operation to follow). It progresses the traversal at the same vertex $v1$ to the next element in clockwise order.

We utilise the structure of this algorithm in the type of Graphs in Agda in Section 2.3. We introduce a type of Steps for the potential steps of a traversal, corresponding to the three steps in Algorithm 1. A full graph traversal is implemented as a sequence of Steps.

We make the following important observation about tree traversals. This property will serve as a guide for the implementation of graphs as traversals of their spanning tree.

▶ **Proposition 16.** *In a graph traversal, edges and corners always alternate.*

Proof. In a graph with a single vertex and no edges, there is only one corner and the property is immediately true. Otherwise, a corner is defined by a vertex and two neighbouring edges at that vertex. We analyse the three possible steps in a graph traversal:

- The only way progress a traversal after a corner has been visited is to inspect one of its two defining edges.
- When recording a looping edge e at vertex v, the traversal moves past this edge at the same vertex. Every vertex in the graph is connected to at least one spanning edge (by the definition of spanning tree). Therefore, the vertex v has at least one more edge attached to it in addition to e. This means that the edge e specifies two different corners (one to its left and one to its right) and as every two neighbouring edges define a corner in between them, the next element in the traversal has to be a corner.
- When following a spanning edge t from v, we arrive at a different vertex v'. This vertex v' contains at least one corner, located next to (and characterised by) the spanning edge t. According to the clockwise graph traversal, this corner is visited next. ◀

2.3 The graph data type

To explain the data type of plane graphs we will first define one Step of a graph traversal. We then show how to combine multiple steps into a sequence to form a full traversal, and in the end consider some degenerate cases of graphs. We assume a type V of a graph's *vertices*, a type E of its *edges*, and a type C of its *corners*. We have observed in Proposition 16 that the traversal of a graph follows a strict pattern of alternating edges and corners. We use this information to guide the implementation of a Step, together with the state of looping edges at that position in the traversal.

Indexing type. A traversal type ⚙ contains two components:

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

The first set in TravTy is a stack of edges E, represented as a List (the type of lists is defined below). This list records the looping edges throughout the traversal. Whenever we encounter one of these edges, the elements on the stack change: the first time we visit a looping edge we add it to the top of the stack, and the second time we pop it from the stack again. The stack-discipline ensures that edges which have been added most recently have to be removed first, which encodes precisely the planarity property of the graph embedding, see Section 2.4. Every step in the traversal is indexed by its effect on the stack of looping edges, with a full traversal starting and finishing on an empty stack.

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

after : Next → Next
after edge   = corner
after corner = edge

What : Next → Set
What edge   = E
What corner = C
```

Figure 5 A traversal is guided by alternating edges and corners. ⚙.

The second set in a TravTy is a marker type Next. This is a two-element type with constructors corner and edge, see Figure 5. Throughout the traversal we use elements of Next as tags, specifying the kind of data we are currently visiting. The lifting of a tag to the data it labels is implemented by the function What (see Figure 5), mapping an edge label to the type of edges E and a corner label to the type of corners C. We use the tag system to specify how data in the traversal *changes* with every step. Recall Proposition 16: in a graph traversal, edges and corner always occur alternately. We enforce this alternation with the function after (see Figure 5) which changes the tag from an edge to a corner, and vice versa. This indexing structure ensures that graph data is well formed at fits together in the right way, by construction.

Sequencing relations with Stars. Before we specify the data type of steps in a traversal, we define a generic type of *sequences*, shown in Figure 6a ⚙.

Given a relation R, this type builds sequences of elements which are pairwise related via R. In addition to the relation parameter, this type abstract over the Direction of the sequence. The direction indicates to which end of a sequence new elements are added: the head element of a cons structure is located on the left, and that of a snoc structure on the right. We define graphically corresponding infix operators, _,-_ for cons and _-,_ for snoc. Instantiating FCat with the two directions leads to the notions of Star (in analogy with the

```
data FCat (d : Dir) { X : Set } (R : X → X → Set) (x : X) : X → Set where
  nil' : { y : X } → x ≡ y → FCat d R x y
  cons : d ≡ cons → { y z : X } → R x y → FCat d R y z → FCat d R x z
  snoc : d ≡ snoc → { y z : X } → FCat d R x y → R y z → FCat d R x z
```

(a) Definition of relation composition.

```
Star = FCat cons
Rats = FCat snoc
```

(b) Instantiation with directions.

```
List A = Star { One } (\ _ _ → A) ⟨⟩ ⟨⟩
Tsil A = Rats { One } (\ _ _ → A) ⟨⟩ ⟨⟩
```

(c) Instantiation with the unit type.

Kleene star) and Rats, as shown in Figure 6b. We will use sequences in both directions later in this work. The simplest case of a forwards sequence is a List, see Figure 6c. Lists are implemented by instantiating Star with the trivial relation on the unit type, indicating that no particular property is enforced on neighbouring elements. Backwards "snoc"-lists, or Tsils, are the corresponding instance of Rats.

Steps of the graph traversal. The traversal type is used to index each Step in a traversal of a graph's spanning tree. The different constructors distinguish different positions inside a tree, and the different data stored in a graph ⚙. See Figure 7 for an illustration.

```
data Step : TravTy → TravTy → Set where
  corner : (c : C)                                                  → Step (es , corner) (es , edge)
  push   : (e : E)                                                  → Step (es , edge) (e ,- es , corner)
  pop    : (e : E)                                                  → Step (e ,- es , edge) (es , corner)
  span   : (e : E) (v : V) → Star Step (es , corner) (es′ , edge) → Step (es , edge) (es′ , corner)
```

The first constructor defines visiting a corner c in a graph. This Step does not change the stack of edges es, but it enforces the next element in the traversal to be an edge. Another potential traversal Step visits a looping edge e. The first time we do so, we push this edges onto the stack es, and the second time it is popped from the stack. We will see shortly (in Theorem 20) that we can assume e to always be at the *top* of the stack whenever we encounter it for the second time in the graph traversal. Both push and pop change the traversal type such that the next step is required to be a corner. The final constructor describes the traversal of a spanning edge e. In this case, the algorithm follows e to reach a new subtree, rooted at vertex v. This subtree is represented by a *sequence* of steps around the child node v, defined as a Star Step. Throughout this Star Step, the indices of any two neighbouring Steps have to match, according to the indexing structure. In the constructor span we ask for an explicit

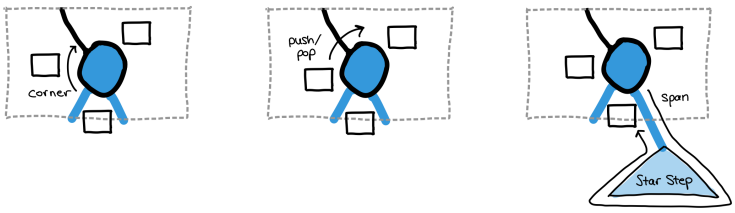


Figure 7 The different cases of constructing a Step of the traversal.

edge e connecting to the root vertex v of the subtree. The traversal of this subtree is passed as a Star Step, starting with the first corner at v and finishing at the edge e through which we have entered the subtree. The subtree must contain at least a corner, thus the Star Step is non-empty (encoded by the change in indexing type from corner to edge). Overall, span describes a Step, progressing from the edge e to expecting a corner as the next element in the traversal. As the traversal of a subtree may add or remove edge from the stack, the span constructor passes the stack es onto the subtree and returns the result es' to its parent.

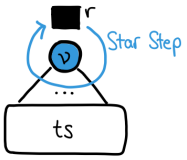
The traversal of an entire graph is implemented as a traversal of all its subtrees, therefore it will itself be represented as an instance of a Star Step.

The data type of graphs. We have specified all the relevant operations to define graphs as traversals of their spanning trees. The data type Graph accommodates degenerate cases of graphs as well as the general case of a tree traversal.

▶ **Definition 17.** *The type* Graph *is defined as the clockwise traversal of a graph's spanning tree, together with the option to add additional edges at any position* ⚙*:*

```
data Graph : Set where
  empty : C → Graph
  vertex : C → V → Graph
  tree   : C → V → Star Step ([] , edge) ([] , corner) → Graph
```

(a) Definition of the data type of Graphs.

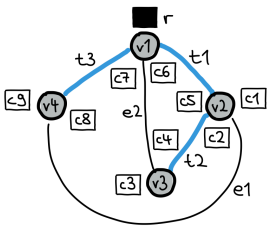


(b) Illustration of tree r v ts.

A graph may be empty in which case it contains neither a vertex nor an edge, but a root corner. The other degenerate case is a graph containing a single vertex r v with no edges attached. This graph contains one corner only: its root r. In the most general case, a graph is specified as a tree traversal. Together with a root corner r located at a vertex v, this constructor expects the traversal of all subgraphs, encoded as a Star Step (see Figure 8b for an illustration). For an entire Graph we ask for the stack of looping edges to be empty at the beginning of its traversal (in the first index of the Star Step), and empty again at the very end of it (in the second index). This reflects the fact that we consider closed graphs only in the implementation (cf. Remark 4). The tree constructor specifies a graph with a non-empty set of subgraphs, as the base cases are already covered by the other constructors. We achieve this by separating the root corner from the Star Step of subgraphs and indexing the Star Step by two different elements of Next (and thus requiring the sequence to contain at least one element). Therefore, starting from the root corner, the traversal starts with an edge and, at the other end, expects a corner to finish (which is the root corner again).

▶ **Example 18.** Assume variables $(v1, \ldots, v4 : V)$ for vertices, $(t1, t2, t3 : E)$ for spanning edges, $(e1, e2 : E)$ for looping edges, and $(r, c1, \ldots, c9 : C)$ for corners. Figure 9 shows an example graph, together with its implementation ⚙. We observe in particular the alternation of edge operations (span,push, and pop) and corners (as discussed in Proposition 16). Figure 9 additionally shows the example graph's rotation system.

▶ **Example 19.** The bouquet graph with one vertex $v1$, two self-loops $e1$ and $e2$, a root corner r and three further corners $c1$, $c2$, and $c3$ is illustrated and implemented in Agda as shown in Figure 10.




```
exGraph : Graph
exGraph
  = tree r v1
      (span t1 v2
               (corner c1 ,- push e1 ,- corner c2
               ,- span t2 v3 (corner c3 ,- push e2 ,- corner c4 ,- [])
               ,- corner c5 ,- [])
      ,- corner c6 ,- pop e2 ,- corner c7
      ,- span t3 v4 (corner c8 ,- pop e1 ,- corner c9 ,- []) ,- [])

exGraphR : rotations exGraph
         ≡ (t1 ,- e2 ,- t3 ,- []) ,- (e1 ,- t2 ,- t1 ,- []) ,- (e2 ,- t2 ,- []) ,- (e1 ,- t3 ,- []) ,- []
```

Figure 9 Graph with vertices $v\cdot$, root r, corners $c\cdot$, spanning edges $t\cdot$, and looping edges $e\cdot$.

```
exBouquet : Graph
exBouquet = tree r v1 (push e1 ,- corner c1 ,- push e2 ,- corner c2
                      ,- pop e2 ,- corner c3 ,- pop e1 ,- [])

exBouquetR : rotations exBouquet ≡ (e1 ,- e2 ,- e2 ,- e1 ,- []) ,- []
```

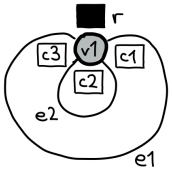


Figure 10 Bouquet graph with one vertex $v1$, root r, corners $c\cdot$, and looping edges $e\cdot$.

2.4 Planarity

In Lemma 11 we have observed that the looping edges alone determine the surface a graph is embedded in. Therefore, the data structure we use to organise looping edges in the tree traversal is crucial for defining a certain graph embedding. Recall that in a plane graph no two edges are allowed to cross each other. Therefore, the rotation at the central vertex of a bouquet graph amounts to a well bracketed word. The stack discipline we use for organising the additional edges ensures that an edge can only be popped if it is positioned on the top of the stack. Therefore, the order of popping two edges from the stack has to be the reverse of pushing them onto the stack. This property expresses precisely the data in a plane graph.

▶ **Theorem 20.** *Terms of type Graph correspond to plane graphs with a spanning tree.*

Proof. "⇒" We show that any term of the type Graph is a plane graph: The empty graph and the graph with one vertex are trivially plane graphs as they do not contain any edges. Consider the constructor tree, in particular the element Star Step: encountering a corner or pushing an edge e does not create any edge crossings (as the other end of the e is not yet attached to a vertex). When spanning a subtree, no edge crossing is introduced as the spanning tree of a graph is always plane. When popping an edge e from the stack, no crossing is introduced because of the stack discipline, meaning that for all other edges e_i in the graph, e_i is fully contained in one of the two regions that e defines (compare also Remark 22). Therefore, the constructor tree implements a plane graph.

"⇐" We show that each plane graph is determined by a spanning tree traversal as defined by the data type Graph. Assume a plane graph G with a spanning tree. The proof consists of two steps. First we contract the graph's spanning tree to a bouquet graph and then show that using a stack ensures planarity of the bouquet graph:

1. Assume a spanning edge e of the graph G and its two ends $s(e)$ and $t(e)$. We have that $s(e) \neq t(e)$, because by definition a spanning tree does not contain self-loops. Contracting e constructs a new vertex v whose rotation is the concatenation of the individual rotations of $s(e)$ and $t(e)$. After the contraction, a traversal of the tree in clockwise order yields the same order of looping edges as before the contraction. We repeat the process for all edges in the spanning tree. As a spanning tree contains all vertices in the graph, we get a bouquet graph with all edges being the looping edges. The rotation at the central vertex in the bouquet amounts to the order of looping edges in the clockwise (and therefore depth-first) traversal of the graph's spanning tree.
2. Assume two edges e_1, e_2 in the rotation of edges around the central vertex in a bouquet graph. By Definition 13 of bouquet graphs, each edge appears in the rotation exactly twice. For e_1 and e_2 to be plane, they must not cross each other. This is the case if they appear in the rotation at the central vertex in the order e_1, e_1, e_2, e_2 (or a cyclic permutation of this order). An edge crossing would looks like e_1, e_2, e_1, e_2 in the rotation. The plane order is enforced by pushing an edge on a stack the first time we visit it in the rotation, and popping it at the time of the second visit. With this stack discipline we are unable to express the edge crossing as above, because by the time we reach e_1 for the second time, e_2 is on top of the stack and we cannot pop e_1. ◀

▶ **Corollary 21.** *A graph is plane if and only if it can be expressed as a term of type Graph.*

Proof. By standard graph algorithms, we can calculate the spanning tree for any graph. Then apply Theorem 20. ◀

▶ Remark 22. The statement of Theorem 20 can be interpreted in an alternative way: Every looping edge e *completes* a face of a graph's surface embedding, see Figure 11. Inserting a looping edge into a spanning tree amounts to splitting the surface into two regions, an "inner one" containing the newly completed face, and an "outer" one. This observation is matched in the structure of the stack, because every edge e can be interpreted to partition the stack into two segments. The "outer" segment contains edges that are stored below e on the stack and cannot be popped as long as e is still on the stack. In Figure 11 this is the case for all edges *outside* of the highlighted region. The "inner" segment of the stack describes edges that are stored on top of e and which have to be popped before e. This is the case for edges *inside* the region enclosed by e in Figure 11.

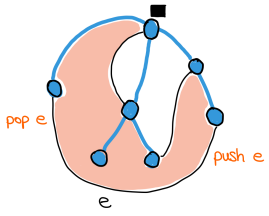


Figure 11 A looping edge enclosing a region (shaded) of the embedding.

2.5 Translation to rotation systems

As we define graphs as clockwise traversals of trees, the translation into a rotation system is straight-forward. We can read off the order of edges around each vertex by inspecting all elements of its traversal Star Step in order ⚙. As the spanning tree visits all vertices in the graph, we will eventually collect all rotations ⚙.

The converse direction of translating between a Graph and its rotation system is not as simple. This is because the spanning tree representation includes a choice of spanning tree and root, compare Remark 7. A translation from a rotation system to a Graph would require choosing a spanning tree for a graph which is an operation we have deliberately left out of the development.

3 Zippers for Plane Graphs

Diagrammatic reasoning consists of applying rewrite rules to subdigrams. To implement this *local* operation, we now introduce data structures to isolate a subgraph of interest. We implement an instance of Huet's Zipper [20] for the spanning tree representation of plane graphs. Zippers are defined for tree-like structures and consist of a subtree in focus together with a context, stored in a path from the tree's root to the focus. For us, this separation will ensure that a rewrite rule is only affecting a particular subgraph and not the context. At every step in a zipper's path, the sibling subtrees to its left and right are recorded. Paths are stored *bottom-up*, allowing for fast access to the subtrees closest to the focus. This provides zippers with a *cursor* operation: immediate neighbours in all directions can be accessed quickly.

▶ Remark 23. We define zippers to focus on a particular *corner* in a Graph, rather than a whole subgraph. We are interested in extending the implementation in the future, but focussing on an individual corner is sufficient for specifying the construction of the zipper.

(a) One corner in focus. **(b)** The layers of the zipper.

Figure 12 Example of a graph's zipper.

Zippers for plane graph consist of the corner in focus and the remaining graph as context. Alongside the path to the focus, we record neighbouring subgraphs and looping edges. The corner in the zipper's focus is located in a particular face of the graph embedding, enclosed by looping edges. These edges form the stack at the corner's position in a graph traversal and have to be included in the information stored in a zipper.

Path structure. Each step in a zipper's path is associated with a central vertex and records a layer of substructures to the left and right of that vertex. Multiple layers are connected to each other via spanning edges between their central vertices. Overall, the path is represented as a sequence of layers from the focus to the root.

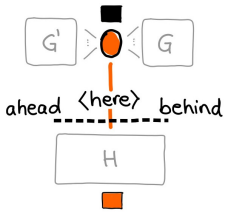
Looping edges. To record looping edges in the path, we introduce an indexing type for layers which contains a pair of edge stacks, one for each side of the path. In fact, every layer is indexed by two of these pairs, one at the "root-side" of the layer and one at the "focus-side" (i.e. four stacks in total). As we traverse subtrees in a layer, the looping edge stacks change. The overall path records the change of looping edges between the stack at the focus and the empty stack at the graph's root. This indexing structure ensures that the planarity property is preserved when a number of layers is rewritten, as long as the change in edge stack remains the same.

3.1 Indexing type

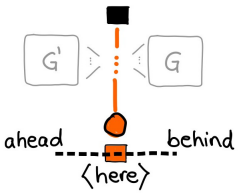
To keep track of the stack of looping edges and preserve the surface of the embedding, each layer in the path will be relating elements of a *zipper type* ⚙. To explain the elements of a zipper type, we imagine a horizontal cut through a path, as illustrated in Figure 13. The indexing type describes the state either in between two layers (Figure 13b) or at one end of the path (Figure 13c). As before, we carefully arrange the constructor arguments in the same diagrammatic order as the graphs they describe.

```
record ZipTy : Set where
  constructor _⟨_⟩_
  field ahead  : List E
        here   : Next
        behind : List E
```

(a) Definition of the zipper type ZipTy as an Agda record.



(b) A ZipTy in between two layers, where here = edge.



(c) A ZipTy at the position of the focus, where here = corner.

Figure 13 Illustration of a zipper type as a horizontal cut through a path.

The field here stores where in the path the cut is made, either through an edge or through a corner In a path, there are only two positions in which the field here is a corner: at the very top of the path (with here being the root corner) and the focus (where here is the corner in focus). At any other position on the path, the ZipTy describes the state of a spanning edge.

The other two fields in a ZipTy record the state of the looping edge stack when the tree traversal is approaching here (which occurs twice in a full graph traversal). In the (clockwise) traversal we first approach here from the top right hand side, with some edges already pushed onto the stack (by the subgraph G in Figure 13). The state of the stack is stored in the field behind, emphasising that it represents the information in the previous traversal steps. To continue the traversal from here, we move further down into the structure. In case of here = edge this means traversing the subtree H which the edge is leading to. In contrast, if here = corner there is no further inner subtree and we immediately return. As the traversal returns from a substructure, it visits the cut position here again, this time approaching from the bottom left hand side. The state of the stack at this point is stored in ahead. The edges on the stack ahead will be popped in the remaining traversal of G' between the cut and the root.

3.2 Path structure

Every step in the path is associated with a vertex v and contains v's subgraphs that are located to the left and right of the path. Additionally, it stores the edge through which the path continues to the next vertex towards the root. As we implement the path to start at

the focus and grow "upwards" to the root, we encode each step in the path as a backwards *layer*, called a Reyal. A Reyal at a vertex v is indexed by two elements s and t of the zipper type, with s encoding the Reyal's root side and t its focus side. These elements store the state of the edge stacks before and after traversing v's side subgraphs ⚙.

```
record Reyal (s t : ZipTy) : Set where
  constructor reyal
  field v    : V
        this : What (here s)
        star : Star Step (ahead t , after (here t)) (ahead s , here s)
        rats : Rats Pets (behind s , after (here s)) (behind t , here t)
```

(a) $s = \_ \langle$ corner $\rangle \_$. **(b)** $s = \_ \langle$ edge $\rangle \_$.

Figure 14 Illustration of the information in a Reyal s t.

Each Reyal is linked to a vertex v on the path to the corner in focus. It relates two ZipTys, s towards the root, and t towards the focus. In addition to the central vertex, a Reyal contains a field this which stores here of s towards the root. This element is either the root corner (see Figure 14a) or an edge (Figure 14b). The sibling subtrees alongside the path in a Reyal are stored in the fields star and rats. On the left hand side of the path are subtrees that occur in the graph's traversal *after* here t has been visited. They are stored as a list of Steps, similar to the traversal of subtrees in the definition of Graphs themselves. The subtrees traversed *before* reaching the focus are located on the right hand side in the Reyal and stored as the *backwards* list rats (whose structure we will explain in the following). The orientation of star and rats allow for easy access to the sibling subtrees closest to the focus of the zipper.

Lists and backward lists. To resemble the cursor-like nature of the original definition of Huet's Zipper, we structure data both in a forwards and backwards way. The the left of the vertex in a Reyal we store a (standard) forwards list of subtrees, represented as Star Step s t, just as we have seen in the definition of Step in Section 2.3. As a Reyal is oriented towards the root, the subtrees to the right of each vertex in the path are stored in an *anticlockwise* direction. Therefore, we introduce a backwards version of a Step, called a Pets ⚙:

```
data Pets : TravTy → TravTy → Set where
  corner : (c : C) → Pets (es , corner) (es , edge)
  push   : (e : E) → Pets (es , edge) (e ,- es , corner)
  pop    : (e : E) → Pets (e ,- es , edge) (es , corner)
  span   : (e : E) (v : V) → Rats Pets (es , corner) (es′ , edge) → Pets (es , edge) (es′ , corner)
```

Similar to Steps, we can sequence Pets, this time using the reverse relation composition Rats (recall Section 3.2). Importantly, the information in a Rats Pets is the same as in a Star Step. The only difference is their orientation.

Sequences of Reyals. An entire path is defined as the concatenation of multiple Reyals, or, more precisely, a Rats Reyal. To construct this structure, we use relation composition with one additional requirement: concatenation of two Reyals is well defined only if they share an edge between them. Any Reyal with an index of shape (_ ⟨ corner ⟩ _) is located at one end of the path. This property of a Rats Reyal being well formed is called connectivity ✿:

```
ConReyals : ∀{ t t′ } → Rats Reyal t t′ → Set
ConReyals (snoc refl { _ ⟨ n ⟩ _ } (rz -, r) r0) = ConReyals (rz -, r) × (n ≡ edge)
ConReyals rz = One
```

Connectivity is only meaningful if a path consists of two or more Reyals. We encode this requirement in the first case of the definition of ConReyals by asking the inputs Rats Reyal to contain a snoc as well as its tail rz being a non-empty sequence of the format (_ -, _). This pattern match exhibits two reyals, $r0$ and r, together with a tail rz. The implementation of the predicate then asks for an edge in between the first two Reyals as well as for the remaining sequence rz -, r to be connected. The base case applies if there are less than two Reyals present because there is no connectivity boundary, and thus the connectivity is trivial.

▶ Remark 26. A Reyal's distinguished element this is located at its root-side. A Rats Reyal does not include the focussed corner of the zipper, but rather leaves a *hole* in its position. Thus, a path stores precisely the *context* for a particular corner in the graph. The construction of a zipper adds the corner in focus to a path.

3.3 The type of zippers for Graphs

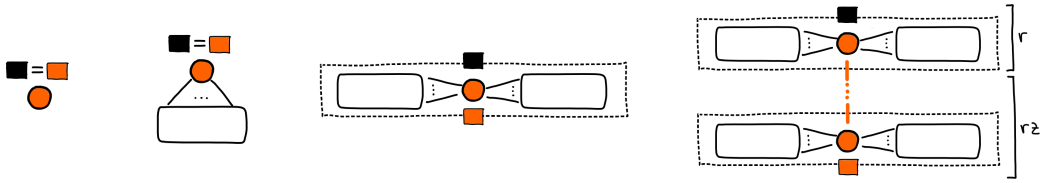
Overall, a zipper is implemented as a corner in focus together with a Reyal structure encoding the path between the root corner and the focus. The Zipper data type is indexed by a list of edges. This list represents the state of the stack of looping edges at the position of the focus when traversing the graph's spanning tree in clockwise direction ✿.

```
data Zipper : List E → Set where
  empty    : C → Zipper []
  vertex   : C → V → Zipper []
  root     : C → V → Star Step ([] , edge) ([] , corner) → Zipper []
  root-vtx : (r : Reyal ([] ⟨ corner ⟩ []) (es ⟨ corner ⟩ es)) → C → Zipper es
  reyals   : (r : Reyal ([] ⟨ corner ⟩ []) (ds ⟨ edge ⟩ ds))
           → (rz : Rats Reyal (ds ⟨ edge ⟩ ds) (es ⟨ corner ⟩ es))
           → ConReyals rz → C → Zipper es
```

Let us unfold this definition slowly. As the focus consists of a corner only, the state of the stack is the same to either side of it, therefore we express it as a single list in the index of the data type.

All constructors for a Zipper ask for an element of type C which is the corner in focus. The remaining input information encodes the context of this focussed corner. Each constructor covers a different shape of graph and a different location of the corner in focus inside the graph. The different constructors are illustrated in Figures 15.

Degenerate graph. In case a graph is empty or consists of a single vertex with no edges attached to it, the only corner a zipper can focus on is the root, see Figure 15a. As there are no edges in the graph, the list of edges in the zipper's index is empty.



(a) vertex. **(b)** root. **(c)** root-vtx. **(d)** reyals r rz.

Figure 15 The different constructors of a Zipper.

Root corner. A Zipper may focus on the root corner of a more complex graph, see Figure 15b. In this case, the entire graph (except the root) constitutes the context. We do not need any layer structure to encode this case; a single Star Step is sufficient. Additionally, as the path from the root to the focus is empty, so is the stack of looping edges at the focus of the Zipper.

Root vertex. If the zipper focuses on any corner that is incident to the root *vertex* (other than the root), the context of this corner may be expressed by a single Reyal r, as shown in Figure 15c. Importantly, this Reyal sits between *two corners*, the root corner and the corner in focus. Therefore, both indices are of the shape (_ ⟨ corner ⟩ _). Because of the definition of a Reyal, the star and rats fields of r are guaranteed to be non-empty. This ensures no overlap with the vertex constructor. The behaviour of a single Reyal cannot be expressed by the more general construction of a zipper as reyals which is why we require the additional constructor root-vtx.

General case. In the general case of a Zipper, its path contains at least two vertices and thus at least two reyals. The constructor asks for a non-empty Rats Reyal structure encoded as an individual Reyal and a Rats Reyal, together with a proof that this non-empty sequence is connected. We enforce the structure to be non-empty by requiring the Reyal r, which is located closest to the root, as a separate argument from the remaining Rats Reyal rz, as is illustrated in Figure 15d. This argument structure in reyals allows for fast access to both the outermost layer r and the innermost layer (at the head of rz). We will make use of this property when reconstructing the original graph from its zipper.

As we have taken great care about storing the looping edges in a stack-like fashion when defining the type of Zipper, we get exactly the right structure to partition a plane graph into a context and a focus:

▶ **Proposition 28.** *A* Zipper *defines a plane graph context together with a corner in focus.*

Proof. The stack of the looping edges is stored in the zipper type at each Reyal. The way we compose Reyals ensures that the stack is preserved in between any two of them. Subtrees to the right and left of a Reyal can alter the stack but only by a valid push or pop operation (guarded by the valid changes of edge stack) which does not introduce any crossing edges. ◀

▶ **Example 29.** This example describes the zipper of a graph containing one vertex $v1$ and one self-loop $e1$. The focus of the zipper is the corner $c1$ which is the only other corner apart from the root r. The zipper is illustrated in Figure 16, and implemented using the root-vtx constructor as we only need one Reyal to describe the subtrees between root and focus.

The index of the zipper contains one edge $e1$, as this edge is on the stack at the corner in focus. In the graph traversal, $e1$ is pushed before reaching $c1$ and popped afterwards ⚙.

```
zi-root-vtx : Zipper (e1 ,- [])
zi-root-vtx
  = root-vtx (reyal v1 r (pop e1 ,- []) ([] -, push e1)) c1
```

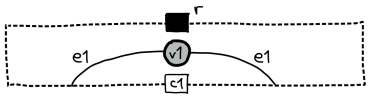
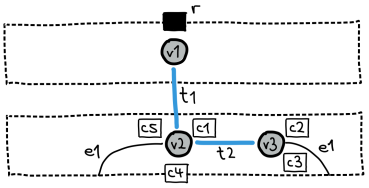


Figure 16 Example of a zipper focussed at the root vertex.

▶ **Example 30.** This larger example of a zipper assumes vertices $v1$, $v2$, $v3$, spanning edges $t1$ and $t2$, a looping edge $e1$, a root corner r, and further corners $c1,\ldots,c5$. The zipper consists of two Reyals and focuses on the corner $c4$. The zipper's index indicates that the edge $e3$ lies on the stack at the corner in focus. The zipper is illustrated and implemented as shown in Figure 17 ⚙.




```
zi-reyals : Zipper (e1 ,- [])
zi-reyals
  = reyals (reyal v1 r [] []) ([] -, reyal v2 t1
             (pop e1 ,- corner c5 ,- [])
             ([] -, corner c1 -, span t2 v3
               ([] -, corner c2 -, push e1 -, corner c3 )))
           ⟨⟩ c4
```

Figure 17 Example of a zipper with two reyals.

4 Computing the Original Tree

In this section we explain how to recover the original graph representation from one of its zippers. This operation is of relevance for the implementation of graph rewriting. The substitution of a subgraph typically operates locally, leaving the context graph unchanged. This property can be easily implemented with zippers. To compute the overall result of a rewrite, we then need to transform the layered structure of the zipper back into a standard spanning tree traversal. The translation operation is illustrated in Figure 18.

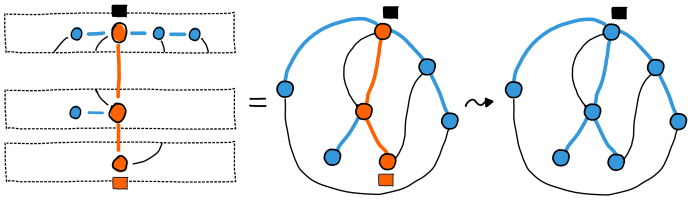


Figure 18 Calculating the original tree from a zipper.

As the focus of a zipper is a corner only in our implementation, this operation has to merely insert the corner in focus at the correct position in the context graph. Whereas the specification seems simple, in the implementation the reorganisation of data from the zipper into a clockwise traversal requires great care as the stack property of the looping edges has to be maintained at all times. Let us first have a look at the implementation of the overall operation ziToOldRoot ⚙ and unfold all the details afterwards.

```
ziToOldRoot : { as : List E } → Zipper as → Graph
ziToOldRoot (empty r)     = empty r
ziToOldRoot (vertex r v)  = vertex r v
ziToOldRoot (root r v st) = tree r v st
ziToOldRoot (root-vtx (reyal v r star rats) c) = tree r v (rats ⟨*⟩ (corner c ,- star))
ziToOldRoot (reyals (reyal v r star rats) rz cR c)
  = let (e , ls , cS) = toLayers rz cR c in starToGraph r (layer v rats e star) ls cS
```

In the first three cases the zipper is focussing on the root corner which makes for an easy calculation of the overall graph. All subtrees (if there are any) are stored in the right order already and together with the root corner r we can build the graph straight away.

Fish and chips. In all other cases, the zipper contains both forward and backward sequences of subtrees. As we move through a focussed structure, we will need to transform forwards and backwards components into each other, preserving their order like beads on an abacus. This is the purpose of two operations, called "fish" and "chips" [17][1]. The fish operator _⟨*⟩⟨_ takes a backwards and a forwards sequence as inputs and computes a backwards sequence ⚙. Similarly, the chips operator _⟨*⟩⟩_ transforms a backwards and a forwards sequence into a forwards sequence:

```
_⟨*⟩⟨_ : Rats Pets k l → Star Step l m → Rats Pets k m
_⟨*⟩⟩_ : Rats Pets k l → Star Step l m → Star Step k m
```

In case of a root-vtx, the zipper focuses on a corner at the root vertex other than the root corner. In this case we have to work with one Reyal of information containing all the subtrees attached to the root vertex, both to the left and right of the corner in focus. To restore the original graph traversal, we combine the two fields of the Reyal which contain the subgraphs, *star* and *rats*. To do this, we call "chips" on the two sequences subtrees *star* and *rats* while inserting the corner in focus in between them. This returns the traversal of the entire graph. Note that because of the indices of type ZiTy, this implementation is only accepted because we are inserting a corner in between *star* and *rats* to concatenate them. This makes sense as the two sequences are the direct neighbours of the corner in focus.

In the most general case, the zipper is a reyals structure with the path containing at least two vertices (and hence at least two reyals). In this case, calculating the original graph consists of two steps: First we transform the bottom-up Reyal structure into a top-down Layer structure and afterwards convert the layered representation into a clockwise traversal.

The first step involves stepping thorough the path and turning each Reyal into a Layer ⚙. This operation orientates the path such that it is centred around the original root (as opposed to the corner in focus), which will be helpful when calculating the clockwise traversal.

```
toLayers : { n : Next }
         → (rz : Rats Reyal (as ⟨ edge ⟩ bs) (ks ⟨ n ⟩ ms)) → ConReyals rz
         → (focus : What n)
         → E × Σ (Star Layer (as ⟨ edge ⟩ bs) (ks ⟨ n ⟩ ms)) λ ss → ConLayers ss
```

Recall that the most general constructor for a zipper, reyals, the Reyal closest to the root is stored separately from the rest of the path. This ensures the correct connectivity property between all Reyals, as the base case of ConReyals describes both the empty Reyal

[1] "Fish" is called fish, because it resembles a fish. And "chips" is the counterpart of fish.

and a single Reyal. We will turn the top-most Reyal into a Layer manually, but the turning of the remaining Rats Reyal is specified by the function toLayers. As the function only has to address these remaining Rats Reyal, the index at the root side contains an edge. This also ensures that we can recursively call toLayers on all subsequent Rats Reyal.

As we step through the entire path and turn backwards into forwards layers, the order of sibling subtrees in the fields star and rats stays the same. The main difference when changing the orientation of the path is the location of the element this: in a Reyal structure, this is located at the root-side and in a Layer structure it is stored at the focus-side. As the corner in focus is not part of the Reyal structure, toLayers requires an additional argument *focus*. After "re-shuffling" the information, the function returns not only a Star Layer but also the edge that is located closest towards the root. This edge was originally part of the Reyal structure but is not include in the Layer structure. Crucially, we are not allowed to forget about this edge: it will be needed in the second step of the calculation. Any function which is reordering the information in a graph or a zipper (such as toLayers) has to be linear in its input arguments. Even though the graph structure may be reorganised, all of its individual elements have to be preserved.

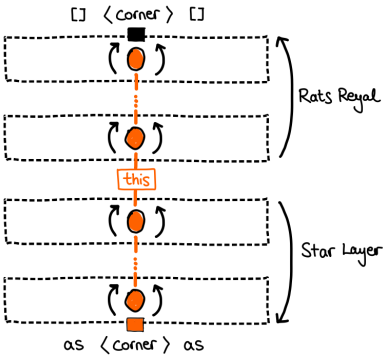


Figure 19 Intermediate state of a toLayers operation.

The transformation from Reyals to Layers exposes an important intermediate state which is shown in Figure 19. The intermediate state occurs after a few (but not all!) recursive calls to toLayers on a zipper. In this intermediate state, the element this is located somewhere in the middle on the path. The path between this location and the focus is stored as a Star Layer. These Layers have already been turned around with the head of the sequence being located next to the element here. The other half of the path, between here and the root of the graph, is stored as a Rats Reyal. These Reyals have yet to be turned into Layers. Again, the head of the sequence is located next to the element here. The operation toLayers transforms Reyals one-by-one into Layers and moves the element here closer to the root with every step. Recall the intuition of zippers acting like cursors inside a graph structure, ready to access any neighbouring elements quickly. The intermediate state in the toLayers operation is precisely implementing this cursor structure. From the position this both the head Reyal and the head Layer are located just next to it. At each call to the function, the head Reyal is moved to the other side of here and forms the new head Layer.

During the entire operation we rely on the fact that every layer and reyal is connected to its neighbours via an edge. This is ensured by the connectivity properties ConReyals and ConLayers which we can easily translate between.

At the end of the toLayers operation, all Reyals have been turned into Layers and we are left with the root sector and one overall Star Layer. We now compute the clockwise traversal of the entire graph by concatenating the subtree sequences star and rats from each layer in the correct order. This is achieved by the function layersToSteps which returns a graph traversal in the form of a Star Step ⚙:

```
layersToSteps : (ss : Star Layer (zs ⟨ edge ⟩ as) (ms ⟨ corner ⟩ ms)) → ConLayers ss
              → Star Step (as , corner) (zs , edge)
layersToSteps (layer v rats this star ,- []) con = rats ⟨*⟩ (corner this ,- star)
layersToSteps (layer v rats this star ,- ss@(_ ,- _)) (refl , con)
  = let st′ = layersToSteps ss con in rats ⟨*⟩ (span this v st′ ,- star)
```

This function is defined for a non-empty Star Layer only which is encoded by the fact that the its index types go from an edge to a corner. The base case covers a single layer, positioned at the very bottom of the path, precisely where the focussed corner is located. In this case we can "chips" the subtrees together, inserting the focus (corner *this*) in the middle.

At any layer further towards the root, layersToSteps concatenates the subtrees in a clockwise manner: the right hand side *rats* is added to the beginning of the Star Step and the left hand side *star* is added to its end. For this case, the property that the layers are connect with each other, expressed by ConStar *ss*, is crucial. The recursive call provides a traversal of the Star Layer structure except of its the topmost layer, as a Star Step. This traversal will be added as a spanned subtree in the overall result, requiring an edge via which it can be reached. The edge has to be located in between the current layer and the rest of the layer structure. This is precisely the property that is expressed by the connectivity of the Star Layer. We match on the proof ConLayers *ss* (which becomes refl) and hereby constraining *this* to be an edge. Therefore the expression span *this* on the right hand side of the definition is well formed.

Together, the operations toLayers and layerToSteps provide the functionality needed to turn a zipper into a graph, rooted at its original root. Carefully incorporating planarity as an intrinsic property of Graphs now pays back, as the proof of the following property is trivial.

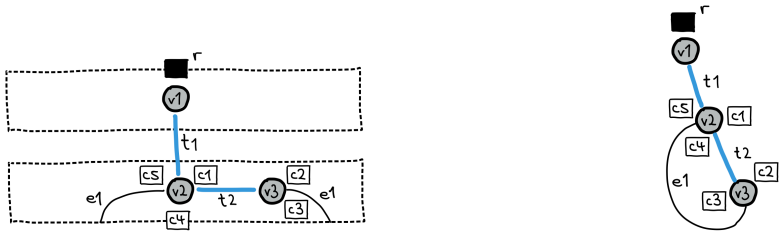
▶ **Proposition 32.** *Calculating the original graph using starToGraph and toLayers from a zipper returns a plane graph.*

▶ **Example 33.** The original tree of the zipper from Example 30 is calculated by the following function and depicted in Figure 20 ⚙.

```
reyals-oldTree : ziToOldRoot zi-reyals
  ≡ tree r v1 (span t1 v2 (corner c1 ,- span t2 v3
      (corner c2 ,- push e1 ,- corner c3 ,- []) ,- corner c4 ,- pop e1 ,- corner c5 ,- []) ,- [])
```

5 Rewriting

Zippers provide the structure to focus on a specific substructure inside a graph. For the goal of implementing diagram rewriting, this separation of a substructure from its context is an important step. A rewrite step then replaces the subgraph in focus with a new graph. In this section, we sketch a rewrite operation for our notion of zipper.



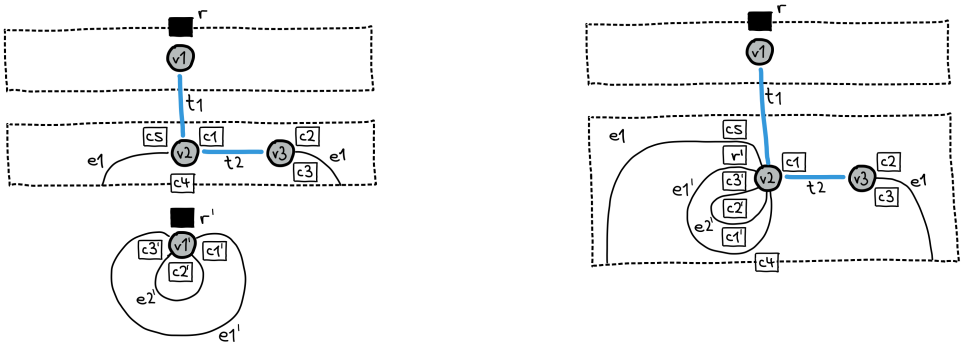
(a) The zipper's path is encoded as two Reyals. **(b)** The original graph.

Figure 20 Example of recalculating the original graph from a zipper.

As our specification of zipper focusses on a single corner only, a rewrite step inserts an entire subgraph at the position of the focussed corner. The operation amounts to placing the subgraph into an enclosed face of the context graph, potentially connecting some looping edges to other vertices in the graph without introducing any crossings. As the corner in focus itself has no influence of the edge stack, we can rewrite it by any graph which takes an empty stack to an empty stack. The rewrite operation inserts the new graph at the vertex where the corner in focus is located, positioned right next to it ⚙:

```
rewriteFocus : { es : List E } → Zipper es → Graph → Zipper es
```

We will omit the implementation details of rewriteFocus and refer to the additional material. Instead, let us look at an example.



(a) Inserting a graph at the zipper's focus,... **(b)** ... updates the reyal closest to the focus.

Figure 21 Example of inserting a graph at the focus of a zipper.

▶ **Example 34.** As an example, we use the zipper from Example 30 and insert the graph with two self-loops from Example 19 at the position of the focus ⚙. The graphs involved and the result of the operation are shown in Figure 21. Observe that the graph is inserted into the Reyal of the zipper closest to the focus $c4$. Just to the left of the corner in focus we now have the two self-loops from the input graph. This example shows that the insertion merges the two vertices $v2$ and $v1'$.

```
re-example : rewriteFocus (zi-reyals v1 v2 v3 t1 t2 e1 r c1 c2 c3 c4 c5)
                          (exBouquet r' v1' e1' e2' c1' c2' c3')
  ≡ reyals (reyal v1 r [] [])
    ([] -, reyal v2 t1 (push e1' ,- corner c1' ,- push e2' ,- corner c2' ,- pop e2' ,- corner c3'
                       ,- pop e1' ,- corner r' ,- pop e1 ,- corner c5 ,- [])
                    ([] -, corner c1 -, span t2 v3 ([] -, corner c2 -, push e1 -, corner c3))) ⟨⟩ c4
```

6 Future Work

The work we present here discusses important design decisions in the implementation of plane graphs in Agda. The focussing operation is a crucial step towards formalising a full graph rewriting. The development of such an automated graph rewriter poses the following challenges which we would like to tackle as next steps in the project:

- Rewriting larger graphs: So far, our implementation covers the case of zippers focussing on a single corner. For a more general notion of graph rewriting, we are interested in focussing on and replacing larger subgraphs. We observe: As the Reyal structure grows bottom-up, we already have easy access to the region of the graph embedding immediately surrounding the focus. With every Reyal away from the focus, this regions grows, but it always forms a disc-like part of the surface. Therefore it can first be removed without invalidating the planarity property and subsequently be rewritten by concatenating a number of Reyals to the front of the remaining path. The indexing type ZipTy ensures that both old and new graph change the edge stack in the same way.
- Finding a graph match: Our implementation assumes a given graph match by providing a certain zipper focussed on the root where a rewrite rule may be applied. For implementing a full rewriter we need an operation which matches the left hand side of a rewrite rule to a subgraph inside a Graph. In particular, this operation needs to compute a zipper structure such that some number of Reyals around the focus exactly match the left hand side of a rule. This is highly non-trivial as not even the choice of spanning tree is known in the beginning (c.f. Remark 7). One possible solution is to compute *a* spanning tree via a simple algorithm like a depth-first approach, and then changing it to match a given graph, one edge at a time. We can imagine an function which forces a looping edge to become a spanning edge, recomputing which edge will then has to become looping in order to keep the non-cyclic property. Crucially, any function which manipulates a Graph, whether by changing the spanning tree, or replacing a subgraph, has to preserve the planarity property.

As additional future work, we would like to extend the framework to incorporate equivalence classes of spanning trees (cf. Remark 7) instead of distinguishing different choices of spanning tree for the same graph. We have already done some work towards this goal by implementing a function ziToNewRoot ⚙ which takes a zipper and *re-roots* its spanning tree to the corner in focus.

Lastly, we are interested in graphs embedded into surfaces of a higher genus than the plane. In Lemma 11 we discussed that the structure of the looping edges alone determining the genus of the surface a graph can be embedded in. If this structure is a stack, then the graph embedding is plane (cf. Theorem 20). We would like to explore which structure can accommodate higher genus surface embeddings. For example, which data structure implements the graph shown in Figure 1b?

References

1 Marie Albenque and Dominique Poulalhon. A generic method for bijections between blossoming trees and planar maps. *Electron. J. Comb.*, 22:2, 2015. doi:10.37236/3386.

2 Malin Altenmüller. *Combinatorial Presentations of String Diagrams for Non-Symmetric Monoidal Categories*. PhD thesis, University of Strathclyde, 2025. URL: https://maltenmuller.github.io/thesis/thesis.pdf.

3 Malin Altenmüller and Ross Duncan. A category of surface-embedded graphs. In Jade Master and Martha Lewis, editors, *Proceedings Fifth International Conference on Applied Category Theory, ACT 2022, Glasgow, United Kingdom, 18-22 July 2022*, volume 380, pages 41–62, 2022. doi:10.4204/EPTCS.380.3.

4 Olivier Bernardi. Bijective counting of tree-rooted maps and shuffles of parenthesis systems. *Electronic Journal of Combinatorics*, 14:R9, 2007. doi:10.37236/928.

5 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004. doi:10.1007/978-3-662-07964-5.

6 Richard S Bird. On building cyclic and shared structures in haskell. *Formal Aspects Comput.*, 24:609–621, 2012. doi:10.1007/S00165-012-0243-6.

7 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. Rewriting modulo symmetric monoidal structure. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 710–719. ACM, 2016. doi:10.1145/2933575.2935316.

8 Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13:43016, 2011.

9 Lucas Dixon and Aleks Kissinger. Open-graphs and monoidal theories. *Math. Struct. Comput. Sci.*, 23:308–359, 2013. doi:10.1017/S0960129512000138.

10 Ross Duncan. *Types for quantum computing*. PhD thesis, University of Oxford, UK, 2006. URL: http://ora.ox.ac.uk/objects/uuid:c2901ae8-9386-4dbf-879d-e37bbc2692bd.

11 Jack R Edmonds. A combinatorial representation for oriented polyhedral surfaces. *Notices of the American Mathematical Society*, 7:646, 1960.

12 Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11:467–492, 2001. doi:10.1017/S0956796801004075.

13 Neil Ghani, Makoto Hamana, Tarmo Uustalu, and Varmo Vene. Representing cyclic structures as nested datatypes. In *Proc. of 7th Symp. on Trends in Functional Programming, TFP*, volume 2006, 2006.

14 Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55, 2008.

15 Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Dongming Wang and Zeng Zhibin, editors, *Computer Mathematics, 8th Asian Symposium, ASCM 2007*, volume 5081, pages 21–22. Springer, 2008. doi:10.1007/978-3-540-87827-8_3.

16 Jonathan L Gross and Thomas W Tucker. *Topological Graph Theory*. Dover Publications, 2001. Original work published 1987 by Wiley-Interscience.

17 Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 43–54. ACM, 2010. doi:10.1145/1863597.1863608.

18 Makoto Hamana. Initial algebra semantics for cyclic sharing structures. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, volume 5608, pages 127–141. Springer, 2009. doi:10.1007/978-3-642-02273-9_11.

19 Lothar Heffter. Über das problem der nachbargebiete. *Mathematische Annalen*, 38:477–508, 1891.

20 Gérard Huet. The zipper. *Journal of Functional Programming*, 7:549–554, 1997. doi:10.1017/S0956796897002864.

21 David J King and John Launchbury. Lazy depth-first search and linear graph algorithms in haskell. *Gla*, 1993.

22 Aleks Kissinger and Vladimir Zamdzhiev. Equational reasoning with context-free families of string diagrams. In Francesco Parisi-Presicce and Bernhard Westfechtel, editors, *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings*, volume 9151, pages 138–154. Springer, 2015. doi:10.1007/978-3-319-21145-9_9.

23 Stefan Milius, Lutz Schröder, and Thorsten Wißmann. Regular behaviours with names - on rational fixpoints of endofunctors on nominal sets. *Appl. Categorical Struct.*, 24:663–701, 2016. doi:10.1007/S10485-016-9457-8.

24 Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins University Press, 2001. URL: http://jhupbooks.press.jhu.edu/ecom/MasterServlet/GetItemDetailsHandler?iN=9780801866890&qty=1&source=2&viewMode=3&loggedIN=false&JavaScript=y.

25 Andrey Mokhov. Algebraic graphs with class (functional pearl). In Iavor S Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 2–13. ACM, 2017. doi:10.1145/3122955.3122956.

26 Peter Selinger. A survey of graphical languages for monoidal categories, 2011. doi:10.1007/978-3-642-12821-9_4.

27 Noam Zeilberger and Alain Giorgetti. A correspondence between rooted planar maps and normal planar lambda terms. *Log. Methods Comput. Sci.*, 11, 2015. doi:10.2168/LMCS-11(3:22)2015.