Combinatorial Presentations of String Diagrams for Non-Symmetric Monoidal Categories

Malin Altenmüller

Mathematically Structured Programming Group

Department of Computer and Information Sciences

University of Strathclyde Glasgow

A thesis presented for the degree of Doctor of Philosophy

October 24, 2025

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed: Malin Altenmüller

Date: October 24, 2025

Abstract

String diagrams are a well established graphical syntax for morphisms in monoidal categories. Reasoning with arrows in a category can be implemented as an instance of diagrammatic reasoning in its graphical language.

This thesis presents work on string diagrams for monoidal categories that do not necessarily contain a symmetry map, that is, a SWAP operation of two wires across each other. While graphs provide a suitable combinatorial structure for string diagrams of *symmetric* monoidal categories, they are not sufficient in our case and we will extend the framework to *surface-embedded* graphs to be able to capture the absence of symmetry in a category and the corresponding topological properties of its string diagrams. We develop the necessary categorical structure of surface-embedded graphs to implement their rewriting as an instance of double-pushout rewriting. Further, we implement the particular case of graphs embedded in the plane in the dependently typed programming language Agda. We develop a suitable, inductive, and finite representation of plane graphs in the setup of Agda's dependent type theory. We show how splitting a graph into a substructure of interest and its context, which is a crucial operation for a rewriting step, can be implemented, and establish a context comonad structure not just for plane graphs but for a much larger class of tree-like data types.

Table of Contents

Previously published work

The contents of Chapter 2 of this thesis have previously been published at Applied Category Theory 2022 [5]. My contribution included the development of the notion of boundary and dual boundary vertex and the corresponding structures for the formulation of pushouts (Definitions 2.41 and 2.58 in this thesis), the structure of morphisms in the category of graphs (Definition 2.17) as well as the proofs of the two main theorems (Theorems 2.49 and 2.62).

Signed:

Date:

Chapter 0

Introduction

String diagrams [87] are a graphical formalism to reason about monoidal categories. Although initially introduced to formalise certain coherence questions in pure category theory [55] they have become a major tool in the theoretical computer science community, finding applications in computability [81, 82], concurrency theory [88, 15], functional programming [73, 83], quantum computing [19, 21], economics [48], natural language processing [22, 53] and control theory [8, 16], among many others. Equational reasoning in symmetric string diagrams can be implemented as graph or hyper-graph rewriting subject to various side conditions to capture the precise flavour of monoidal category intended [28, 29, 30, 57, 14, 13]. These techniques have proven effective both for pen and paper calculation, and as the basis for automated systems [58, 9, 23].

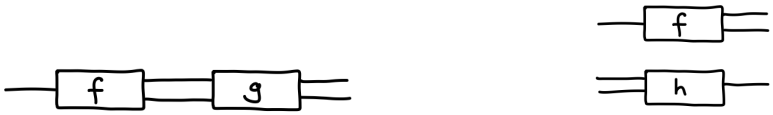
However, the vast majority of existing work treats only the *symmetric* monoidal setting where the wires are allowed to cross freely. The non-symmetric setting has received comparatively little attention despite its importance in applications. There are both theoretical and practical reasons why the non-symmetric case is of interest. From the abstract perspective, symmetric theories can be formalised inside a more general non-symmetric framework, where the symmetry is realised by explicit operations and equations; the same is true for theories where the underlying category is braided monoidal, for example in conformal field theory [54]. By excluding the symmetry from the formalism it becomes possible to reason about this larger class of theories. A more practical motivation comes from the area of quantum computing, where string diagrams are often used to model quantum circuits [20]. From this perspective allowing the wires to cross freely means that SWAP gates are implicit, and any connectivity restrictions imposed by the qubit architecture cannot be represented. This is inadequate for reasoning about realistic quantum devices, where such architectural restrictions are often

severe and overcoming them can significantly increase the size and complexity of the circuit [24].

We focus on the case of graphs embedded in the plane the preservation of the planarity property during graph operations. But this work is actually more general. Rotation systems can represent graphs embedded in higher genus surfaces, and an extension can accommodate even non-orientable surfaces. Therefore, the results we present are relevant not only in the plane setting but more generally for rewriting constraint by the topology of more exotic surfaces which has important applications, for example in quantum software [20].

0.1 Monoidal categories and their graphical syntax

Monoidal categories are widely used to model processes in which we cannot only progress in time but also in space. Processes are represented by the morphisms of the category together with two operations to place them in parallel or in sequence. Ordinary composition models sequential composition in which one process runs ahead of the other one. This is illustrated by placing the two morphisms next to each other horizontally, as illustrated in Figure 0.1a, with data flow going from left to right.



(a) horizontal composition $g \circ f$ (or: $f \fatsemi g$). (b) vertical composition $f \otimes h$.

Figure 0.1: Horizontal and vertical composition. Time flows from left to right.

In addition to sequential composition, by supplying a tensor product between objects and morphisms, monoidal categories provide a way of composing processes in parallel. This is illustrated by placing morphisms next to each other vertically, as shown in Figure 0.1b.

String diagrams are a two-dimensional graphical syntax for monoidal categories. This syntax is very natural as the two dimensions of the plane can represent exactly the two possibilities of joining processes by composition and tensor product, respectively.

The data of a monoidal category contains certain equations that have to hold between morphisms, called coherence conditions. In the diagrammatic syntax these naturality conditions hold automatically which makes them such a convenient tool to use for reasoning in monoidal categories. A famous example is the interchange law which states an equation between terms containing both function composition and tensor product. It states that the order of horizontal

and parallel composition does not matter: $(f \fatsemi g) \otimes (h \fatsemi k) = (f \otimes h) \fatsemi (g \otimes k)$. In the graphical language, this equation is true by definition, as the diagram for both sides of the equation is the same. This is illustrated in Figure 0.2.

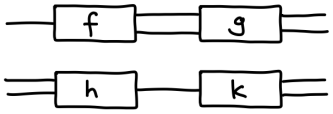


Figure 0.2: The interchange law:$(f \fatsemi g) \otimes (h \fatsemi k) = (f \otimes h) \fatsemi (g \otimes k)$.

The translation from morphisms in a monoidal category to its graphical representation is not a one-to-one mapping. Instead, the graphical language represents equivalence classes of morphisms. These equivalence classes arise precisely from the coherence axioms for monoidal categories [67]. Therefore, different flavours of monoidal categories (specified by its operations and equations) require different characteristics of the diagrams in the graphical syntax [87]. A few examples are illustrated in Figure 0.3 and we discuss them here:

- In a *symmetric* monoidal category (SMC) wires can cross freely, expressed by a symmetry operation $\sigma_{A,B} : A \otimes B \to B \otimes A$ (Figure 0.3a) together with an equation $s_{B,A} \circ s_{A,B} = id_{A \otimes B}$. This *double swap* equation expresses that crossing the same two wires twice amounts to the identity morphism. The only information encoded in the morphisms of a SMC is the connectivity relation of objects between the morphisms. Any spatial arrangement of the objects does not matter. This property is sometimes referred to by the slogan "only connectivity matters". For the graphical language of SMCs this means that it only matter to which boxes a wire is attached to, but not how two different wires related to each other.

- Similarly, in a *braided* monoidal category wires can be swapped with each other (see Figure 0.3b), but in this case the operation can not be undone by swapping the same two wires again. Instead, other equations on the braiding operation have to hold.

- In an *autonomous* category, objects A have *duals* A^*. In the graphical language, duals correspond to wires going from right to left. The coherence conditions for autonomous categories include the existence of unit and counit map which translate to "cup" (Figure 0.3c) and "cap" (Figure 0.3d) diagrams, respectively.

This work is concerned with the diagrammatic language of monoidal categories with particular *topological* properties: wires may only cross each other in non-trivial ways (including

(a) symmetry $A \otimes B \to B \otimes A$. (b) braiding $A \otimes B \to B \otimes A$. (c) cup $A \otimes A^* \to I$. (d) cap $I \to A^* \otimes A$.

Figure 0.3: Maps typical of braided, symmetric, and autonomous monoidal categories.

no crossing at all). Monoidal categories containing this type of information represent process theories for systems which are surface sensitive, with the most prominent example being quantum circuit diagrams. The swapping of two elements in a quantum circuit is not a trivial operation, hence any occurrence has to represented explicitly in the theory as well as in the graphical language. But topology-sensitive string diagrams have other applications beyond quantum programs. Any monoidal theory with a non-trivial symmetry operation does not allow for arbitrary wire crossings as part of of the graphical language. To be able to generalise both symmetric as well as more complex theories (such as braided monoidal categories), we will have to start by assuming no symmetric structure at all, and later on incorporate additional operations and equations as required.

0.2 Combinatorial representation of string diagrams

To be able to implement this diagrammatic syntax and specify their rewrite theories, we require a *combinatorial* representation for string diagrams. In this representation, the diagrams themselves are the mathematical objects which can be manipulated by operations in the theory. By optical analogy, graphs seem a reasonable candidate for this combinatorial representation, and most works, including ours, use graphs as the preferred structure. Importantly, graph theory is a well studied subject and provides existing frameworks for the manipulation of graphs. One example is subgraph substitution which is an important operation for the implementation of rewrite theories for string diagrams.

As this work is concerned with diagrammatic languages that are sensitive to their topology, we will use graphs *embedded onto surfaces* as our combinatorial representation. The specific topological requirements on the diagrams are directly translated into surface conditions for the corresponding graph embeddings. We therefore move away from "only connectivity matters" towards a more sophisticated structure of edges in a graph. Central to the representation of surface-embedded graphs is the fact that any embedding can be encoded by a fixed order of edges around each vertex, due to Edmonds [52].

Part I of this work describes the modelling of a category of surface-embedded graphs (in

Chapter 2) and the construction of an operad of plane graphs (in Chapter 3). The underlying categorical structure is designed to include connectivity information of all edges involved, thereby admitting the addition of ordering information to each vertex. Further, the category of surface-embedded graphs admits double-pushout rewriting, which we use to encode diagram rewriting theories. Additionally, we show that surface-embedded graphs together with substitution of a vertex for a subgraph form an operad, in the style of Spivak's operad of wiring diagrams [90].

In Part II of this work we develop a data type for plane graphs in the dependently typed programming language Agda. We will discuss multiple steps of this development.

Firstly, graphs are data structures that contain cycles [11]. We explain how to define a graph *inductively* by using its spanning tree as cycle-free, inductive structure. Alongside the spanning tree we store the remaining graph edges. When manipulating the graph, we will have to take care of both its spanning tree and the additional edges. This is content of Chapter 5.

Secondly, for implementing graph rewriting we need to be able to express focussing on a certain substructure inside a graph. We specify how to split a graph into a subgraph in focus and its surrounding context graph [51] and how to move the focus to a different subgraph (in Chapter 6). We are eventually able to navigate to any position in the graph and calculate its context graph. From there we can perform a graph rewrite step by replacing any node in focus with an entire new subgraph.

We will observe that the structure of the additional edges in the implementation of the data type of graphs determines the surface in which the graph can be embedded in. This structure consists of a traversal order of the tree. By picking a suitable traversal order and data type to store the additional edges in, we construct a type of intrinsically plane graphs, whose terms are not only plane by definition but also retain planarity when manipulated, for example by the action of a rewrite rule.

0.3 Related work

There are a number of tools that implement the rewriting theory for string diagrams. Each of them models the manipulation of diagrams as an instance of graph rewriting for a certain class of monoidal theories.

The main difference in our work compared to the existing tools is the fact that we have to consider surface-embedded graphs because we do not assume any implicit symmetries in

the string diagrams. This requires a different notion of boundary of a graph as we have to be sensitive to the order of edges around each face of the surface embedding (including the outside face) and open edges are not a useful notion in this context. Furthermore, implementing graphs and their rewriting in Agda means that we can reason about the diagrams that we implement directly in the same environment. We can also enforce the topological constraints on graphs in their data type, thus excluding any operations on them that do not preserve these constraints.

- CARTOGRAPHER [89] is a graphical tool for string diagrammatic reasoning modulo the laws of symmetric monoidal categories. The system uses a notion of *open hypergraph* [13] to represent a diagram's inputs and outputs and to ensure that composition and substitution is well formed. In our work on graphs for non-symmetric monoidal categories, the special treatment of the outside edges is essential, too. As in our case the order of edges around vertices matters, we use a different notion than open hypergraphs, but the motivation is the same.

- Quantomatic [58] is a tool that allows for reasoning with string diagrams modulo compact closed categories. The boundary of a graph is specified by special vertices that have a single incident edge. In Quantomatic, users can specify a set of rewrite rules and, based on these rules, generate larger proofs of equivalence between diagrams.

- Globular [9] and its descendant homotopy.io [23] are graphical tools implementing reasoning for higher categories. These tools are more general than our work as they do not assume any deformations of the diagrams corresponding the structural isomorphisms in of the monoidal category by default.

- Rewalt [45] is a tool implementing reasoning for string diagrams of higher categories. Diagrams in Rewalt are represented as *diagrammatic sets* (a generalisation of simplicial sets), and thus generalise well to higher dimensions.

- Chyp [59] is an interactive theorem prover for string diagrams of symmetric monoidal categories [12]. It directly translates between a declarative language and the corresponding string diagram. Users can specify rewrite rules which are used by a small tactic language in Chyp to show equivalences of larger diagrams. Chyp is based on cospans of hypergraphs [13] where it also takes its name from.

Part I

Categories for Open Surface-Embedded Graphs

Chapter 1

Introduction

In this Part I we develop the theory of string diagrams for non-symmetric monoidal categories, and their rewriting systems.

String diagrams provide a graphical syntax for morphisms of monoidal categories (MCs). Using them, we can express terms of the monoidal category in a two-dimensional, graphical way. Any operation on terms in a monoidal category can be expressed graphically, by the coherence theorem for string diagrams [87]. To be able to formalise this graphical language and reason with it we need to express the diagrams themselves as mathematical objects. Typically, for string diagrams as well as other types of process diagrams, these mathematical objects are graph structures. In their most general form, graphs consist of sets of vertices and edges together with a way to connect them with each other. The properties of a concrete monoidal category determines the shape of its graphical syntax which in turn influences the particular shape of the graphs that represent the string diagrams.

For this work, we are concerned with a class of graphs which are suitable to represent string diagrams for non-symmetric monoidal categories. This flavour of monoidal categories has some very specific characteristics, all of which influence the choice of graph representation for their diagrammatic language. We will describe the three main properties here.

From graphs to graph embeddings In a symmetric monoidal category (SMC), we have, alongside the usual equations for coherence, a symmetry condition which states that swapping two objects and then swapping them again has the same effect as not swapping them in the first place. For the string diagrams of a SMC, this amounts to the fact that we can move wires around arbitrarily, even across each other. As long as their ends stay attached to the same boxes, the semantics of the morphism does not change. This feature of SMC is often referred

to by the slogan "only connectivity matters".

As a combinatorial representation for string diagrams graphs are a good candidate. Graphs are network-like structures; in their most general form they consist of a set of vertices and a set of edges that are related to each other by functions. Graphs are especially suitable for representing morphisms of *symmetric* monoidal categories. The key information stored in a graph is the incidence relation between vertices and edges which precisely expresses the connectivity information required for SMCs.

In the case of non-symmetric monoidal categories we have to think more carefully about the arrangement of wires between boxes. There is no equation that allows wires to cross each other, therefore we can only consider graph structures in which edges do not share any points. In addition to connectivity information, now topology matters as well. Graphs which only contain sets are now too general as a combinatorial representation. To incorporate the topological component we will instead be using particular *drawings* of graphs, also known as surface embeddings of graphs. Drawings are sensitive to the placement of edges of a graph, thus by using them as combinatorial representation we are able to distinguish diagrams with different edge layouts. An example of two different drawings of the same graph is illustrated in Figure 1.1.

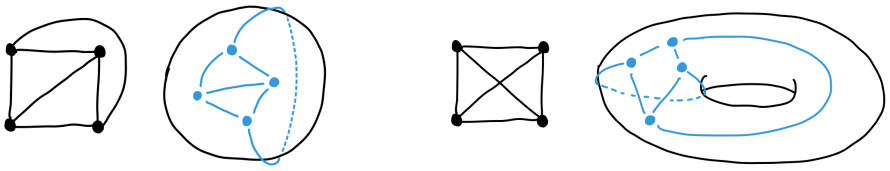


Figure 1.1: Two different surface embeddings of $K4$, the complete graph on 4 vertices.

Remark 1.2. The case of non-symmetric monoidal categories (or, more precisely: not necessarily symmetric monoidal categories) is not only interesting in itself, but it also serves as a generalisation of MC that impose equations on the arrangement of objects and morphisms. The non-symmetric case captures their joint underlying theory which can be instantiated to the particular flavour of monoidal category afterwards, for example by adding symmetry or braiding equations.

Open graphs With string diagrams as graphical syntax for morphisms of monoidal categories, they typically contain a notion of inputs and outputs, representing the domain and codomain of the morphism they encode. Sequential composition of two string diagrams is well defined if the inputs and outputs at the composition boundary match. Parallel composition combines inputs

and outputs of the diagrams involved. Inputs and outputs — together, a diagram's *interface* — are the connection points between a diagram and its *environment* (or *context*). They serve as the channels over which information can be transmitted.

In their representation as graphs, diagrams with inputs and outputs can be modelled by *open* graphs. Edges in open graphs need not be connected to a vertex at one (or both) of their ends. We use these edges connected to a vertex on one end only to represent the inputs and outputs of a diagram. Composition of open graphs may identify open edges, thus they can connect outputs and inputs at the composition boundary to each other.

In the case of surface-embedded graphs though, open edges are not a suitable structure to represent a graph's interface. When composing two graph embeddings we need to be able to to guarantee that the result is again a valid embedding on the same surface. With open graphs we cannot achieve this property easily. Edges that are only connected at one of their endpoints do not contribute to the characterisation of the graph's embedding. Once we compose two graphs and identify certain edges, they become connected at both ends and now do contribute the the specification of the graph's embedding. In the standard notion of graphs, edges are treated as (unordered) sets, and therefore can be connected arbitrarily during composition, potentially interfering with the topology of the resulting graph embedding. Additionally, as encoding of surface-embedded graphs we will use the structure of the endpoints of edges attached to vertices (called *flags*). A particular arrangement of these endpoints uniquely determines a graph embedding. Therefore, if some of the endpoints of edges are not attached to a vertex at all, they are not captured by our notion of graph embedding.

Instead of using open edges, we analyse the faces of a graph's surface embedding, and introduce an auxiliary vertex representing the region surrounding of a graph which we call a graph's "outside" face. We attach the open ends of a graph's input and output edges to the auxiliary vertex and therefore are able to include them in the definition of graph embedding. An illustration of this operation is shown in Figure 1.3. In fact, all graphs in our framework will be *total*, meaning that they do not contain any open edges at all.

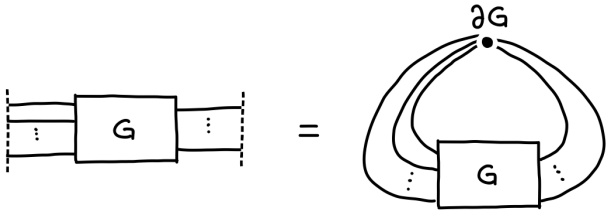


Figure 1.3: An open graph G can be represented by a graph with a *boundary vertex* ∂G.

Specifying surface embeddings of graphs with these kinds of interfaces, and developing their categorical structure, is the main focus of Chapter 2.

Rewriting A key characteristic of monoidal categories is the ability to compose objects and morphisms both in sequence and in parallel with each other. Within this framework we are interested in another, derived notion of combining diagrams: by substitution of a smaller diagram into a larger one. Equational theories typically consist of rules that equate subdiagrams, with reasoning strategies for those theories implementing equations as rewrite rules. Therefore, rewriting is a key operation for our specification of graphs and graph morphisms.

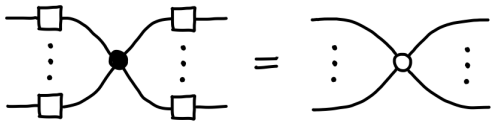


Figure 1.4: A diagram equality in the zx-calculus.

As an example, Figure 1.4 shows the Hadamard rule of the zx-calculus [19]. It states that a black vertex with multiple inputs and outputs corresponds to a white vertex with Hadamard gates (represented by white boxes) applied to all its inputs and outputs.

Replacing the left hand side of a zx-rewrite rule with the right hand side within a larger diagram produces an equivalent structure according to the theory. To realise this behaviour we require a notion of *substitution* for diagrams. Figure 1.5 shows an example rewriting operation of the Hadamard rule inside a larger zx-diagram. We replace the left hand side of the rule with the right one while the rest of the diagram remains the same.

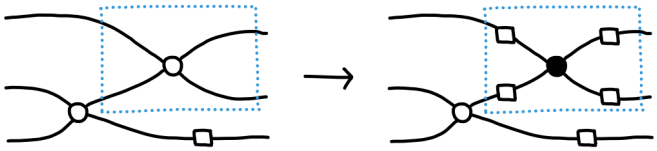


Figure 1.5: Application of rewriting the rule from Figure 1.4 inside a larger zx-diagram.

The substitution operation for graphs consists of two steps: First we remove a subgraph from a larger graph, and afterwards we insert the new subgraph at the same position. The implementation of this operation exhibits an interesting new structure: a graph with a *hole* where a subgraph has been removed and another one will be inserted. Correspondingly, our theory will have to accommodate for graphs that have holes. As these structures may contain open edges, too (which we were trying to avoid, for topological reasons) we will treat graphs with holes in an analogous way to graphs with interfaces.

Overall, we present a monoidal category of graphs in Chapter 3 which has all of the above properties: graphs are represented as plane surface embeddings, with composition and tensor product preserving these properties. We introduce auxiliary vertices to deal with open edges in graphs to guarantee their topological properties. Lastly, we derive a suitable notion of substitution and rewriting for this class of graphs.

The definition of the substitution operation motivates an alternative presentation of graphs. In contrast to monoidal categories, this framework assumes substitution of subgraphs as a *primitive* (and both compositions as derived operations). A graph in this framework is not defined as a number of sequential and parallel compositions, but as a collection of its subgraphs with a certain edge structure between them. With this characteristics, graphs and substitution form an operad. In general, operads are used to specify structures which are formed from multiple substructures. We define this alternative representation of open graphs as operads in Chapter 3. Furthermore, we suggest a dual perspective where graphs are not build by combining multiple subgraphs, but they are taken apart into subgraphs according to a particular pattern language. As both graphs and patterns live in the same underlying category, we can study their interaction and define a notion of pattern matching for graphs.

1.1 Categories, monoidal categories, and string diagrams

We give some definitions that will be important throughout the following chapters. For a more comprehensive compilation we recommend a text book on category theory and monoidal categories [66, 64].

Category theory provides a framework for studying structured data. Numerous structures in mathematics and theoretical computer science can be expressed and analysed in this framework. Properties of entire classes of data is defined generically using the abstracted notions in the theory and can be instantiated from there to any concrete data that fits into the framework. This approach both removes the need of duplication and allows for transferring well known properties and constructions from one concrete case to another. For example, in this work we will use a generic notion of *rewriting* which includes a certain format of rewrite rules as well as the structures we can apply them to.

We start with the definition of a category which is the simplest building block in category theory. Categories and their properties are studied by themselves, but we will also be interested in interactions between different categories.

Definition 1.6. A category $\mathbf{C}$ consists of the following data:

- a collection of *objects* $A, B, C, ...$, called $\mathsf{Ob}(\mathbf{C})$,
- for each pair A, B of objects, a collection of *morphisms* (or: *arrows*) $f : A \to B$,
- for each pair of morphisms $f : A \to B$, $g : B \to C$, their *composition* $g \circ f : A \to C$ (or, alternatively: $f \fatsemi g : A \to C$),
- for each object A, an *identity morphism* $\mathrm{id}_A : A \to A$,

such that the following properties hold:

- The identity morphism is the (left and right) unit of composition: for every morphism $f : A \to B$ in $\mathbf{C}$, we have $f \circ \mathrm{id}_B = f = \mathrm{id}_B \circ f$.
- Composition is associative: for every $f : A \to B$, $g : B \to C$, $h : C \to D$ in $\mathbf{C}$, we have $(h \circ g) \circ f = h \circ (g \circ f)$.

Remark 1.7. If the collection of morphisms from A to B in a category $\mathbf{C}$ is *small* (i.e. a proper set), we sometimes refer to it as the *hom-set* and write $\mathbf{C}(A, B)$ for it.

Example 1.8. These examples are going to be important in the next chapters:

1. The category $\mathbf{Set}$ has sets as objects and functions between sets as morphisms.
2. The category $\mathbf{Pfn}$ has sets as objects and partial functions between sets as morphisms.
3. the category $\mathbf{Inj}$ has sets as objects and injective functions between sets as morphisms.
4. the category $\mathbf{Pos}$ has partially ordered sets as objects and monotone maps as morphisms.
5. the category $\mathbf{Mon}$ has monoids $(M, \_ \bullet \_, e)$ as objects and monoid morphisms as arrows. More specifically, arrows are functions on the underlying set M which preserve the multiplication $\_ \bullet \_$ and identity element e.
6. Given two categories $\mathbf{C}$ and $\mathbf{D}$, the *product category* $\mathbf{C} \times \mathbf{D}$ is defined as:
   - the objects are pairs (C, D) of objects $C \in \mathbf{C}$ and $D \in \mathbf{D}$,
   - a morphism $(C, D) \to (C', D')$ is a pair (f, g) of morphisms $f : C \to C' \in \mathbf{C}$ and $g : D \to D' \in \mathbf{D}$.

The definition of a category captures a wide class of mathematical objects. In addition to exposing the compositional structure of these objects, we are interested in relations between different categories. The first relation is containment with which we can highlight and analyse a certain subclass of a category.

Definition 1.9. A *subcategory* $\mathbf{D}$ consists of objects and morphisms in $\mathbf{C}$, subject to conditions:

- For any object $A \in \mathbf{D}$, its identity morphism id_A is also in $\mathbf{D}$.
- For any morphism $A \to B \in \mathbf{D}$, both A and B are in $\mathsf{Ob}\,(\mathbf{D})$.
- If $f : A \to B$ and $g : B \to C$ are in $\mathbf{D}$, then so is their composite $g \circ f : A \to C$.

$\mathbf{C}$ is called *supercategory* of the category $\mathbf{D}$. A subcategory $\mathbf{D}$ is called *full* if for each pair of objects $A, B \in \mathsf{Ob}(\mathbf{D})$, all morphisms $f : A \to B$ from $\mathbf{C}$ are also in $\mathbf{D}$. A subcategory is called *wide* if it includes all objects of its supercategory, i.e. $\mathsf{Ob}(\mathbf{D}) = \mathsf{Ob}(\mathbf{C})$.

More generally, we can compare the data and properties between two categories by specifying maps between them. Maps between categories that preserve the composition and identity structure are called functors.

Definition 1.10. Given two categories $\mathbf{C}$ and $\mathbf{D}$, a functor $\mathcal{F} : \mathbf{C} \to \mathbf{D}$ consists of:

- a function on objects : $\mathcal{F} : \mathsf{Ob}(\mathbf{C}) \to \mathsf{Ob}(\mathbf{D})$,
- a function on morphisms: $\mathcal{F} : (f : A \to B) \to \mathcal{F}(A) \to \mathcal{F}(B)$,

such that the following properties hold:

- $\mathcal{F}$ preserves identities: for every object A in $\mathbf{C}$,

$$\mathcal{F}\mathrm{id}_A = \mathrm{id}_{\mathcal{F}A}.$$

- $\mathcal{F}$ preserves composition: for every $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$ we have:

$$\mathcal{F}(g \circ f) = \mathcal{F} \circ \mathcal{F} f.$$

Example 1.11. Forgetful functors $\mathcal{U}$ forget some of the structure of a category, but preserve at least the composition and identity operations. For example, there is a forgetful functor from the category $\mathbf{Mon}$ of monoids to the category $\mathbf{Set}$. This functor maps every monoid $(M, \_ \bullet \_, e)$ to its underlying set M and monoid maps to their underlying functions on sets.

Example 1.12. We will later encounter a definition of a graph as a functor. In Definition 2.8 we define a graph as a functor $G : (\bullet \rightrightarrows \bullet) \to \mathbf{Set}$ where $(\bullet \rightrightarrows \bullet)$ is the category with two objects and two arrows between them (plus the identity arrows). This functor picks two sets (one for each element $\bullet$), and two functions between those sets. Thus, a graph consists of an edge set E and a vertex set V together with two functions $s, t : E \to V$ between them, defining the source and target vertices for each edge. Graphs defined in this way are a typical example of functors imposing a certain shape on data by mapping a "pattern" category onto a more general category (like $\mathbf{Set}$).

In general, there may more than one functor between two categories. To analyse different functors between categories we now define maps between them. The bijective subclass of these maps define equality between functors.

Definition 1.13. Given two functors $\mathcal{F}, \mathcal{G} : \mathbf{C} \to \mathbf{D}$, a *natural transformation* $\alpha : \mathcal{F} \Rightarrow \mathcal{G}$ is a family of maps consisting of:

- for each object $A \in \mathbf{C}$ a morphism $\alpha_A : \mathcal{F}A \to \mathcal{G}A$ in the category $\mathbf{D}$.

such that the following square commutes for any $f : A \to B$ in $\mathbf{C}$ (called the *naturality* of α):

$$\begin{array}{ccc} \mathcal{F}A & \xrightarrow{\alpha_A} & \mathcal{G}A \\ \downarrow{\scriptstyle \mathcal{F}f} & & \downarrow{\scriptstyle \mathcal{G}f} \\ \mathcal{F}B & \xrightarrow{\alpha_B} & \mathcal{G}B \end{array}$$

If for every object $A \in \mathbf{C}$, the morphism α_A is an isomorphism in $\mathbf{D}$, α is called a *natural isomorphism*.

Example 1.14. The *identity natural transformation* of a functor $\mathcal{F} : \mathbf{C} \to \mathbf{D}$ maps each object $A \in \mathbf{C}$ to the identity morphism $\mathrm{id}_{FA} \in \mathbf{D}$.

We will be interested in a slightly weaker variant of a natural transformation. For a *lax* natural transformation, the naturality condition does not need to hold strictly, but requires a map from one path to the other only. For our application it is enough to express this map as an order on the paths, therefore the lax naturality condition is defined as an arrow in the category of partially ordered sets $\mathbf{Pos}$.

Definition 1.15. Given functors $\mathcal{F}, \mathcal{G} : \mathbf{C} \to \mathbf{Pos}$, a *lax natural transformation* is a map $\alpha_A : \mathcal{F}A \to \mathcal{G}A$ for each object $A \in \mathbf{C}$, such that a lax naturality condition holds: for every $\mathbf{C}$-morphism $f : A \to B$, there exists a morphism $\mathcal{G}f \circ \alpha_A \to \alpha_B \circ \mathcal{F}f$ in $\mathbf{Pos}$.

Monoidal Categories We now introduce an important class of categories for our application. The structure of standard categories is enough to talk about computational systems that can be composed sequentially. In addition, monoidal categories provide the structure to compose systems *in parallel* as well. This is a very important property to capture the nature of a lot of systems which are build from small components that are placed next to each in horizontally *and* vertically.

Definition 1.16. A category $\mathbf{C}$ is called *monoidal* if it is equipped with

- a functor $\_ \otimes \_ : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, called the *tensor (or: monoidal) product*,
- an object $I \in \mathsf{Ob}(\mathbf{C})$, called the *monoidal unit*,

such that the following natural isomorphisms exist for any objects $A, B, C \in \mathbf{C}$:

- an *associator* $\alpha : (A \otimes B) \otimes C \cong A \otimes (B \otimes C)$,
- a left unitor $\lambda : I \otimes A \cong A$,
- a right unitor $\rho : A \otimes I \cong A$,

satisfying certain coherence conditions [55].

A monoidal category is called *strict* if the unitors and associator are identity natural transformations. A monoidal category is called *symmetric* if for any two objects A, B an additional natural isomorphism, $\sigma : (A \otimes B) \cong (B \otimes A)$, exists and satisfies certain equations, including: $\sigma_{B,A} \circ \sigma_{A,B} = \mathrm{id}_{A \otimes B}$.

Monoidal categories provide operations to compose morphisms in two dimensions, horizontally and vertically. Therefore they provide enough information to draw them on a page. Making the definition of such a drawing precise results in a graphical language for monoidal categories. The elements of this language look similar to process diagrams but they are mathematically rigorous in that we can use the graphical syntax alone to specify morphisms of the monoidal category and operations on them.

Definition 1.17. Objects and morphisms of monoidal categories can be expressed with a graphical syntax, called *string diagrams*. In this syntax, objects are represented as *wires* and morphisms are represented as *boxes*. Each morphism has its domain object drawn as an incoming wire (to the left) to the box and its codomain object as an outgoing wire (to the right). The identity morphism is drawn as a single wire. Composition of morphisms is depicted

as two boxes next to each other that are connected via wires representing the objects at the composition boundary. The tensor product of two objects is drawn with two wires in parallel, and the tensor product of morphisms places the two boxes next to each other vertically.

Figure 1.18 shows an example string diagram, for a given composite of morphisms.

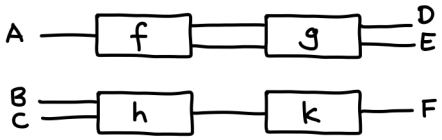


Figure 1.18: String diagram of the morphism: $(f \fatsemi g) \otimes (h \fatsemi k) : A \otimes (B \otimes C) \to (D \otimes E) \otimes F$.

Remark 1.19. To be able to reason about the equality of morphisms in monoidal category in the corresponding graphical syntax, we need a mathematical representation for string diagrams which is typically done using graphs. This thesis is about developing a suitable combinatorial representation of string diagrams for a certain flavour of monoidal category.

Functors between monoidal categories have to satisfy an additional condition as they have to preserve the structure of the tensor product:

Definition 1.20. A functor $\mathcal{F} : \mathbf{C} \to \mathbf{D}$ between two monoidal categories $\mathbf{C}$ and $\mathbf{D}$ is called *(lax) monoidal* if it preserves the monoidal structure, i.e. there exist natural transformations:

- $\epsilon : \mathcal{F}(I_{\mathbf{C}}) \to I_{\mathbf{D}}$,
- $\mu : \mathcal{F}(A) \otimes \mathcal{F}(B) \to \mathcal{F}(A \otimes B)$,

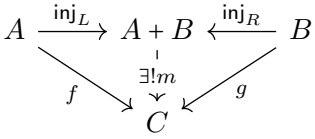
satisfying certain conditions. A monoidal functor is called *strict*, if both ϵ and μ are natural isomorphisms.

The most important example of monoidal category for our application is a product category:

Example 1.21. A *PRO* [61] ("product category") is a strict monoidal category whose objects are the natural numbers, and whose tensor product (on objects) is given by addition. A morphism of PROs is a strict monoidal functor which is the identity on objects.

We will now introduce a few construction inside a fixed category, starting with some canonical ways of combining objects in a category to form new objects. Firstly, coproducts describe the "categorification" of the disjoint union of sets.

Definition 1.22. Given a category $\mathbf{C}$ and two objects $A, B \in \mathsf{Ob}(\mathbf{C})$. A *coproduct* of A and B is given by an object $A + B$ (the coproduct) and two arrows (called *injections*) $A \xrightarrow{\mathsf{inj}_L} A + B \xleftarrow{\mathsf{inj}_R} B$, such that, for any object C and arrows $A \xrightarrow{f} C \xleftarrow{g} B$, there exists a unique map $m : A+B \to C$ such that the following triangles commute.

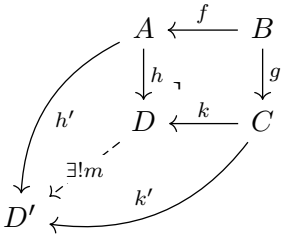


Example 1.23. In the category $\mathbf{Set}$, the coproduct $A + B$ is the disjoint union of sets $A \uplus B$. We may therefore use + for sets to mean disjoint union.

The triangle property in the definition of coproducts determines that the coproduct is the smallest possible way to combine the two objects. This is called a *universal property*. It additionally ensures that the forming of a coproduct is canonical.

The next construction describes objects with a universal property again, a generalisation of coproducts. It describes another way to combining two objects in a category, but this time the objects may share some common structure. This shared structure acts as a kind of interface along which the objects are combined.

Definition 1.24. Given a span of maps $A \xleftarrow{f} B \xrightarrow{g} C$ in a category $\mathbf{C}$, a *pushout* of A and C consists of an object D and arrows $A \xrightarrow{h} D \xleftarrow{k} C$, such that the top right square commutes:



,and such that, for any object D' and pair of morphisms $A \xrightarrow{h'} D' \xleftarrow{k'} C$ that also make the square commute, there exists a unique morphism $m : D \to D'$ making the triangles commute.

Lemma 1.25. *The category* $\mathbf{Set}$ *of sets and functions has pushouts.*

Proof (sketch). Given a span of sets $A \longleftarrow B \longrightarrow C$, the pushout is defined as follows: $D = (A + C)/\sim$ where $\sim$ is the least equivalence relation such that $f(b) = g(b)$ for $b \in B$. □

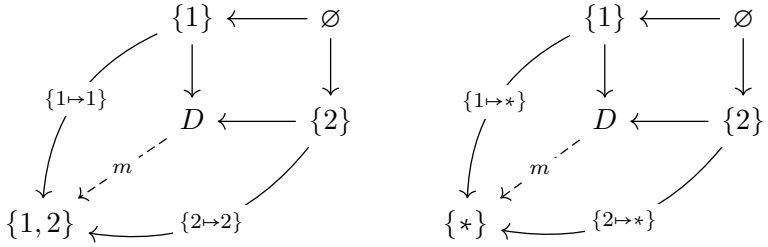
Remark 1.26. In $\mathbf{Set}$, the pushout of the inclusions of the empty set $A \supseteq \varnothing \subseteq C$ corresponds to the coproduct $A + C$.

Lemma 1.27. *The category* $\mathbf{Pfn}$ *of sets and partial functions has pushouts.*

Proof (sketch). The definition of pushouts in **Pfn** looks similar to the one in **Set**: Given a span of sets $A \longleftarrow B \longrightarrow C$, the pushout is defined as: $D = (A + C)/\sim$ where $\sim$ is the least equivalence relation such that $f(b) = g(b)$ for all $b \in B$ on which both f and g are defined. Note that the pushout in **Pfn** may contain more elements than the pushout in **Set** because the equivalence relation only identifies elements on which both f and g are defined. □

Lemma 1.28. *The category* **Inj** *of sets and injective functions does not have pushouts.*

Proof. We assume the existence of pushouts in **Inj** and analyse their shape. Consider the span $\{1\} \longleftarrow \varnothing \longrightarrow \{2\}$ and the following (pushout) diagrams in **Inj**:



From the left diagram, we observe that the pushout D has to consist of the disjoint union of elements of the feet of the span (same as in **Set**), otherwise m would not be right-unique (i.e. not a function). Now consider the right diagram: if the pushout D is the disjoint union $\{1, 2\}$, then the mediating map $m : \{1, 2\} \to \{*\}$ is not an injection. □

We finish this little round trip of some concepts in category theory with the definition of an enriched category. In the specification of an ordinary (small) category $\mathbf{C}$, the morphisms for any two objects are represented as a set. Sets themselves do not have much structure, but we may be interested in categories in which morphisms can have more structure. The notion of enriched category allows for specification of the structure of morphisms as *another category*:

Definition 1.29. Given a monoidal category $\mathbf{K}$, a category $\mathbf{C}$ is *enriched in* $\mathbf{K}$ if it has, in addition to a collection of objects as before, for each pair of objects $A, B \in \mathbf{C}$ a hom-object $\mathbf{C}(A, B)$ of the category $\mathbf{K}$. The identity arrow for every object $A \in \mathbf{C}$ is defined by an arrow $I \to \mathbf{C}(A, A)$ from the monoidal unit in $\mathbf{K}$. Composition in the category $\mathbf{C}$ is defined using $\mathbf{K}$'s tensor product by a specified arrow,

$$\_ \circ \_ : \mathbf{C}(B, C) \otimes \mathbf{C}(A, B) \to \mathbf{C}(A, C)$$

subject to a number of conditions.

Example 1.30. The category **Pfn** of partial functions is enriched in the category of posets **Pos**: its morphisms are partial functions, and we can equip these partial functions with a natural order relation which is the information contained in the category **Pos**. The order relation is defined element-wise: $f(a) \leq g(a)$ is true if either both f and g are defined on a and $f(a) = g(a)$ or neither f nor g are defined on a or f undefined and g defined on a. We will use this enrichment of **Pfn** in Definition 2.10 as the map defining a certain lax natural transformation (cf. Definition 1.15).

1.2 Graphs and graph embeddings

As our main goal is to study operations on certain kinds of string diagrams, we need a theory in which we can represent them as mathematical objects and define operations on them. These objects will be certain kinds of graphs. We will now explain some of the relevant definitions of graphs and their surface embeddings. For a more thorough introduction we refer to the literature on topological graph theory [42, 77].

Definition 1.31. 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 vertex, and vice versa.

Remark 1.32. When the direction of edges is not relevant for a particular example, we may omit it in illustrations of examples, for simplicity.

Definition 1.33. A graph is *finite* if both V and E are finite sets. For a vertex v in a finite graph, we define its *degree* as the number of edges attached to it:

$$\deg v = |\{e \in E | v = s(e) \vee v = t(e)\}|$$

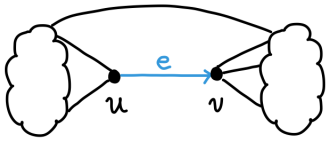
Remark 1.34. As we use graphs in this work to represent (iterable) diagrams, we assume all graphs in this work to be finite.

Definition 1.35. A graph is called *simple* if it does not contain any self-loops, that is, for any edge $e \in E$, $s(e) \neq t(e)$, nor parallel edges between vertices: for any two edges $e_1, e_2 \in E$, $s(e_1) = s(e_2) \wedge t(e_1) = t(e_2) \Rightarrow e_1 = e_2$. If a graph does contain self-loops and/or parallel edges, it is called a *multigraph*.

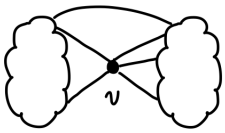
Definition 1.36. Given a graph G, a *subgraph* H of G has as vertex and edge sets subsets of G's vertices and edges, and the source and target structure is inherited from G: for any edge $s_H(e) = s_G(e), t_H(e) = t_G(e)$ for all $e \in E_H$.

Notation Given a subgraph H of a graph G. We write $G \setminus H$ for the graph obtained by deleting all the vertices and edges of H from G.

For some analysis of graphs it will be useful to apply the following operation which decreases the size of a graph but preserves certain properties we will be interested in. *Edge contraction* removes an edge from a graph and at the same time merges its source and target vertices. An illustration of edge contraction is shown in Figure 1.37.



(a) before contracting edge e.



(b) result of edge contraction.

Figure 1.37: Edge contraction of edge e identifies its source and target vertices.

Definition 1.38. 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.

Example 1.39. If the graph G contains another edge $e' \in E$ with the same source and target vertices as e, i.e. $s(e') = u$, $t(e') = v$, then this edge forms a self-loop at v after e has been contracted. This scenario is illustrated in Figure 1.40.

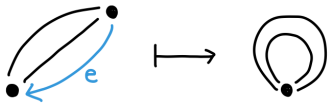


Figure 1.40: Contracting an edge e may create self-loops.

Lemma 1.41. *The order of repeated edge contractions does not matter: for any graph H and edges $e1, e2 \in H$, we have $H/e_1/e_2 \cong H/e_2/e_1$.*

The inverse operation of edge contraction is called *vertex splitting* and consists of dividing a vertex (together with its incident edges) into two, and inserting an edge between them.

The structure we will mainly be working with are drawings of graphs on surfaces:

Definition 1.42. A *manifold* is a topological space in which the neighbourhood of every point is topologically isomorphic to an open disc.

Definition 1.43. A *surface* is a two-dimensional connected manifold. Two surfaces are *homeomorphic* if we can continuously deform one into the other.

Remark 1.44. The concept of a disc-like region of a surface will be important in the following chapters. Every part of a surface which is enclosed by the edges of a graph embedding is homeomorphic to an open disc. This intuition motivates the introduction of explicit structures for the outside region of a graph, as well as any hole inside it.

Remark 1.45. For the scope of this work we assume surfaces to be closed and orientable. Extending our framework to more complex surfaces could be an interesting project in the future.

Figure 1.46 shows some closed, oriented surfaces. The sphere makes the start with genus 0, the torus contains one hole and has genus 1, the double-torus (of genus 2) contains two holes, etc. We can increment the genus of a surface by adding "handles".

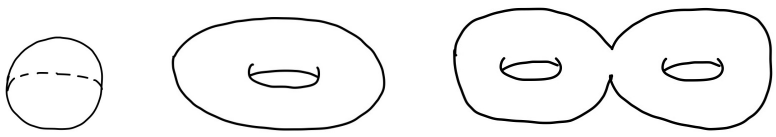


Figure 1.46: Sphere, torus, and double-torus are the lowest genus, orientable surfaces.

Proposition 1.47. *The plane is homeomorphic to the sphere with one point removed.*

Proof. The proof is by constructing the stereographic projection from the sphere onto the plane, with the point removed as the centre of the projection. See Figure 1.48 for an illustration of the construction. □

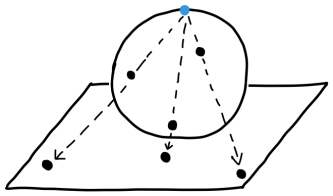


Figure 1.48: Stereographic projection, using the "north pole" as the centre of projection.

Definition 1.49. 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. No two arcs share any point other than their endpoints which means there are no crossing edges in a graph embedding. The *faces* of a graph embedding are regions of the surface that are enclosed by cycles of edges of the graph. We assume embeddings to be *cellular*, meaning that every face of an embedding is homeomorphic to an open disc.

Definition 1.50. Two embeddings are *equivalent* if they are equal up to homeomorphism of the surface they are embedded in. We consider the resulting equivalence classes as the same embedding, sometimes called a *map*. We will make this precise by representing equivalent graph embeddings by this notion by the same combinatorial structure.

Of particular interest to us are graph embeddings in surfaces of genus 0. These are called *plane graphs*, and they are the ones that can represent structures in which edges are not allowed to cross at all.

Definition 1.51. We characterise graph embeddings by the surface with the smallest genus that it can be embedded in. In the special case of a graph embedding of genus 0 we call the embedding *plane*. If a graph G has *a* plane embedding, G itself is called *planar*.

Proposition 1.52. *A graph can be embedded into the sphere if and only if it can be embedded into the plane.*

Proof. Given the plane, we add a point "at infinity" and identify the boundary of the plane with this point to create the sphere. This construction does not change the graph embedding. Conversely, given a graph embedded into the sphere, we pick a point which does not lie on the

graph as the north pole which is to be removed. Then we can apply stereographic projection as in Proposition 1.47. □

The following observation about plane graph embedding provides an intuition about the relation between edges and faces of a graph embedding.

Theorem 1.53 (Euler Formula)**.** *Let G be a plane graph with vertices V, edges E. Let F denote the faces of the embedding. Then $|V| - |E| + |F| = 2$.*

For example, consider a graph embedding G. Adding an edge to G that connects two vertices splits one of its faces into two. The Euler Formula still holds, as the additional edge and additional face cancel each other out.

Remark 1.54. For illustration purposes we draw graph embeddings somewhat planarly (with the page being the "plane"). For plane graphs this is a true depiction. For any higher genus embedding, its illustration contains some edges that cross. We are nevertheless able to illustrate the combinatorial information of any graph embedding by a drawing onto the page. The reason for this is the key property of our combinatorial representation of graph embeddings, and content of Theorem 1.61. Figure 1.55 shows an example of two embeddings of the same graph on two different surfaces.

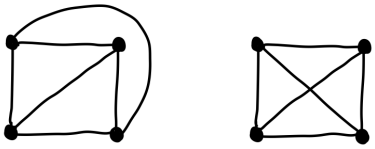


Figure 1.55: Two different embedding of the graph $K4$.

Choice of combinatorial presentation

Graphs as specified in Definition 1.31 consist of sets of edges and vertices and the incidence relation between them. To characterise surface-embedded graphs, we additionally require a specification of the arrangement of faces on the surface the graph is embedded in. Because we assume graph embeddings to be cellular, the arrangement of the faces can be expressed as the cycles of edges encompassing each face.

There are various ways to encode graph embeddings combinatorially of which we choose *rotation systems*. In addition to the source and target relation between vertices and edges, rotation systems store the order of edges around each vertex. This is enough information to encode the embedding of a graph into a surface.

Definition 1.56. A *cyclic list* is a member of an equivalence class of lists (i.e. finite, ordered sets), generated by the following equivalence relation: Given a set A, an element $a : A$, and a list as with elements from A, $\{a\} +\!\!+ as = as +\!\!+ \{a\}$, where $+\!\!+$ defines concatenation of lists.

Definition 1.57. 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 a graph G consists of a rotation for each of its vertices $v \in V_G$. We choose the convention that rotations correspond to *clockwise* incidence lists around each vertex.

Remark 1.58. Some authors use the term *pure* rotation system for the above structure. Having no use for the impure kind (sometimes also called *embedding schemes* [77]), which correspond to graphs embedded in non-orientable surfaces, we will omit this qualifier; see Gross and Tucker [42] Ch. 3 for details.

Remark 1.59. Since our focus here is on plane graphs — that is, graphs equipped with a specific plane embedding — we will frequently refer to rotation systems themselves as graphs.

Definition 1.60. Given a graph $G = (V, E)$ and a rotation system for it. Two edges $e_1, e_2 \in E$ are called *adjacent* if they share a common end $v \in V$ and they occur next to each other in the rotation of v.

The following theorem is the key for using rotation systems as combinatorial representation for surface embeddings of graphs. It is usually attributed to Heffter [49] and Edmonds [52]; we refer to Gross and Tucker [42].

Theorem 1.61. *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.*

Proof (Sketch). Given a rotation system, we calculate the corresponding embedding as follows:

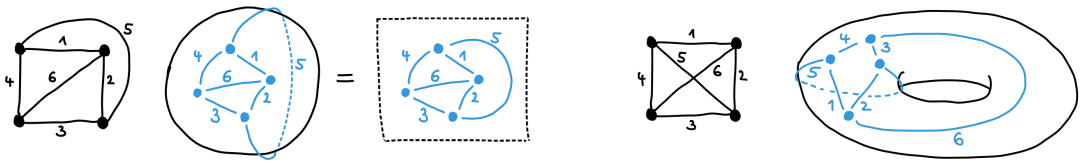
1. Construct the faces of the embedding: Starting at one endpoint of an edge e, we trace a face by following e to its other endpoint. We then select the neighbouring edge of e in the rotation at the endpoint vertex. We repeat the process with the next edges, until we have reached the initial edge again.

2. Construct the surface the graph is embedded in: each edge in the graph is an element of the boundary of exactly two faces, located to either side of it. Having calculated all faces

of the embedding we now glue them together along the edges (considering the direction of the edges). This will construct the surface of the embedding.

For the reverse direction we simply read off the order of edges around each vertex from a graph's embedding. □

Example 1.62. We can distinguish the two graph embeddings in Figure 1.55 by their rotation systems. The way in which we have drawn them defines an order of edges for each vertex. We can translate this order into a surface embedding. Figure 1.63a shows the embedding of the graph in the surface of a sphere. The drawing on the left is already plane, therefore the embedded graph looks the same. The illustration of the toroidal embedding contains edges that cross each other (cf. Remark 1.54). As shown in Figure 1.63b, we can draw the embedding on the torus *without* any edges crossing. The embedding is determined by the rotations around each vertex which are the same in both versions of the illustration.



(a) on the sphere. (b) on the torus.

Figure 1.63: Two different embeddings of $K4$, defined by their rotation system.

Remark 1.64. Theorem 1.61 refers to connected graphs but Definition 1.57 does not impose this requirement, so how should disconnected graphs be treated? Explicitly, if a surface-embedded graph has two connected components then one component must be a contained in a face of the other. This containment relationship can be established by identifying a common face in each component. Multiple connected components lead to a hierarchy of containment relations between components sharing faces. The choice of root component is arbitrary so it will be useful to equip this structure with a notion of *refocussing* which we will not attempt here. For this reason, the graphs we formalise here are *component-wise* plane.

Our main reason for choosing rotation systems is that we can use them as additional structure on top of the representation of graphs as vertices and edges per Definition 1.31. In our development, we will start from the well studied category of graphs and modify to fit our purpose. In a second step we add rotation information to the vertices. The main effort goes into the precise definition of graphs and their morphisms, while all along we keep in mind the

purpose of adding rotation information afterwards. Once we have arrived at a suitable category of graphs, adding a rotation system is very straightforward and all the structural properties follow from the underlying category of graphs.

Remark 1.65. Even though the examples and illustrations (and really, the main motivation of this work) are plane graphs, they only appear later in this Part. The definitions and results of Chapter 2 apply to all surface-embedded graphs specified by rotation systems. In fact, even rotation systems only come into appearance quite far into the chapter. Before then, we spend a lot of effort on defining a certain category of graphs for which the addition of rotation systems is straight-forward. We add rotation — and hence surface embedding information — in Section 2.4, and discuss the particular case of plane graphs in Chapter 3.

1.3 Graph rewriting

Equational theories for monoidal categories typically consist of a set of rewrite rules of subterms. We can apply a rule whenever a bigger term contains a subterm which is the subject of the left hand side (LHS) of a rewrite rule. Replacing this subterm with the right hand side (RHS) of the rule results in equivalent terms according to the theory.

Given a monoidal theory, we can apply rewrite rules not only to its terms but also to its string diagrams. The idea is the same: given a rule equating two subdiagrams, we replace its left hand side subdiagram with its right hand side at some position inside a bigger diagram. As we use graphs to represent string diagrams, we can implement this operation formally as an instance of graph rewriting.

The following discussion assumes a (not yet defined) suitable category of graphs where objects are graphs and morphisms express a subgraph relation: an arrow $G \to H$ expresses the relation "G is a subgraph of H". Given such a category of graphs, applying a rewrite rule to a graph amounts to an instance of *double pushout (DPO) rewriting* [34]:

$$\begin{array}{ccccc} L & \xleftarrow{l} & B & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & \lrcorner & \downarrow{\scriptstyle c} & \lrcorner & \downarrow{\scriptstyle n} \\ G & \xleftarrow[g]{} & C & \xrightarrow[h]{} & G[R/L] \end{array} \tag{1.1}$$

We start from a rewrite rule $L \Rightarrow R$ of subgraphs L and R. In general, these subgraphs may be *open graphs*. As long as both subgraphs have the *same* interface, the rewrite rule is well formed. When rewriting a subgraph for another one, we have to ensure that the proposed substitute actually fits into the same spot. Therefore, rewrite rules have to contain graphs of

the same interface type on both sides. To ensure this preservation of interfaces, rewrite rules $L \Rightarrow R$ are represented as spans $L \leftarrow B \rightarrow R$, where B is a common subgraph of both L and R specifying the interface of both.

Apart from the rule itself we require some information on where to apply it inside a larger graph. This is specified by a morphism $m : L \rightarrow G$ which defines a *match* of L within G. The match points at the precise subgraph to be replaced by the rewrite. The actual rewriting happens in two stages: First we remove the LHS graph L from G. This operation amounts to taking the pushout complement of the composite arrow $B \rightarrow L \rightarrow G$. The result of removing L from G is a so called *context graph* $C = G \setminus L$ with a "hole" at the position of L. The fact that this operation defines a pushout square ensures that the hole that we have created has again the same interface as L and R. The second step is the insertion of the RHS graph R into the hole in the context graph. This operation is calculated as the pushout of the span $C \leftarrow B \rightarrow R$ which completes the "double-pushout". We know that R fits into the hole in the context graph because of the common boundary between L and R, and the hole inside C.

The DPO approach to rewriting captures an important characteristic of rewrite rules. It emphasises that applying a rule only changes the subgraph in focus and no other part of the graph; the context is unaffected by a rewrite rule. This is ensured by the separation of the subgraph and the context by the span $L \leftarrow B \rightarrow C$ (and the corresponding commutation of the LHS square) before the application of the rewrite. The actual rewriting operation is only defined on the small subgraph and not on the context graph. The overall result graph is calculated after the rewrite has taken place. This is an elegant framework because it does not only isolate the subgraph in question but also makes rewriting *modular* in that we can rewrite the same subgraph in a different context graph very straightforwardly, as long as the interfaces of the subgraph and its context match.

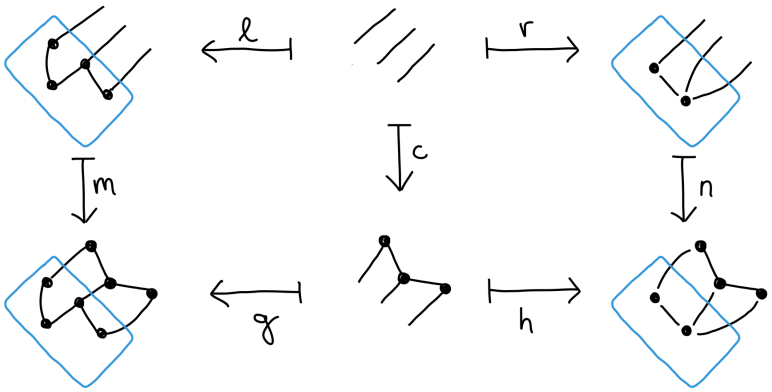


Figure 1.66: A concrete example of DPO rewriting

Figure 1.66 shows a concrete example of a DPO rewrite, where the boundary graph consists of three edges representing the open edges of the left and right hand side of the rewrite rule, and therefore also the open edges of the hole in the context graph.

Adhesive categories

In order for DPO rewriting to be well defined for a category of graphs $\mathbf{C}$, this category has to have certain properties. These are:

1. Monomorphisms of $\mathbf{C}$ express a subgraph relation.

2. In $\mathbf{C}$, pushouts of monomorphisms exist.

3. $\mathbf{C}$ has pushout complements of monomorphisms, and they are unique.

Condition 1 ensures that we actually deal with embeddings of graphs into bigger graphs. Condition 2 expresses that we can form the pushout of two graphs, and therefore be able construct the two squares in a DPO diagram. Calculating the context graph by removing the LHS subgraph from G requires the existence of pushout complements. Additionally, for the whole rewrite to produce a deterministic result, removing L from G needs to produce a unique result. These requirements are captured by Condition 3. *Adhesive* categories [62, 63] are a class of categories in which DPO rewriting is well defined. They satisfy certain properties about monomorphisms and pushouts from which Conditions 1 – 3 can be proven. Unfortunately, adhesivity will not be a suitable characteristic for our purposes. In the category of graphs we will be constructing, monomorphisms do not express a useful subgraph relation. This is mainly due to the fact that the category includes some partial functions, and a monomorphism would address the total fragment only, which is too restrictive. However, we will be able to prove the above conditions on pushouts and pushout complements for a relevant class of graph embeddings, and therefore we will have sufficient structure to use the framework of DPO rewriting.

1.4 Related work

Equational reasoning in string diagrams is typically implemented as a graph or hypergraph rewriting system [57, 30, 29, 13, 95] subject to various side conditions to capture the precise flavour of the monoidal category intended.

In the interpretation of diagrams as hypergraphs, the objects of the monoidal category are encoded as the graph's vertices (with the condition that they have at most two incident edges) and the morphisms are expressed by hyperedges with the appropriate number of input and output elements.

In our work we use graphs (and not hypergraphs) to represent diagrams, with wires being translated to edges and boxes to vertices of the graph. This notion follows the structural similarity between diagrams and graphs. Equipping a graph with a rotation system corresponds to fixing the order of input and output edges of a box in a string diagram. In a string diagram, the arity of a generator never changes as any particular morphism has a fixed number of inputs and outputs. This property is ensured by restricting graph morphisms to those that preserve the vertex' rotation.

In addition to choosing a graph representation of a string diagram, any system which models rewriting has to contain a notion of outside interface of a graph. This ensures that composition of two graphs as well as substitution of a subgraph into a larger graph is well formed. In certain applications, the distinction between the outside face of a graph and the graph itself is more important. An example is the modelling of quantum circuits [31] in which the geometry of the circuit is as important as its connectivity. the work in this thesis is motivated by a similar argument in that we have to be very explicit about which face of a graph embedding is the outside. In general, there are various approaches to representing the interface of a graph:

Hypergraphs with interfaces This work [14] equips a category of hypergraphs in which rewriting by double pushout is already defined and extends it to a category of graphs with interfaces, constructing the corresponding instance of DPO rewriting and showing that confluence for rewriting systems for this kind of graph is decidable. A graph with interface is a morphism $J \to G$ where J is a discrete graph that represents the inputs and outputs of the graph.

Cospans of hypergraphs The idea behind cospans of hypergraphs [13, 12] is similar to graphs with interfaces, except that interfaces are split into an input and an output side. A graph with m input and n output edges is represented as the cospan $m \longrightarrow G \longleftarrow n$. In this framework, the graphs representing the number of input and output edges are *discrete*: they consist of vertices only. These discrete graphs are mapped onto the "dangling" edges of an open hypergraph. While this is enough structure for composition to type check, it is not quite enough to talk about graph embeddings. Composition of open graphs may introduce a crossing

of two edges at the composition boundary which would violate the embedding property of the graphs involved. Our work therefore has to deal with more complex "interface graphs" in which the graphs at the feet of a cospan are not discrete but contain edges.

Structured and decorated cospans Structured [6] and decorated [7, 37] cospans describe systems with interfaces more generally, with open graphs being one instance of the framework. A cospan in a category $\mathbf{C}$ describes a system at the apex and its input and output interfaces at the feet. Composition of two cospans with a shared composition boundary corresponds to forming a pushout along this boundary. Some additional information in the two formalisms of structured and decorated cospans is used to distinguish the amount of information encoded by the system itself versus its interfaces. In general, the apex contains more information than the feet. In the instance of cospans of hypergraphs, this is the fact that the boundaries are discrete graphs whereas the apex contains edges as well. A decorated cospan contains an additional element to the cospan which equips (or *decorates*) the apex with this additional information. In a structured cospan, the interface objects live in a different category $\mathbf{D}$ and the cospan is formed in $\mathbf{C}$ by applying a free functor $\mathcal{F} : \mathbf{D} \to \mathbf{C}$ to the feet. Structured and decorated cospans can be used to describe graphs, but they also apply to structures like circuits or Petri nets.

Contexts in monoidal categories Coend calculus [85] is a formalism that presents processes which do not consume (or produce) all of their inputs (or outputs) at the same time. This leads to non-standard shapes of morphisms and the need for incomplete diagrams in the formalism. The calculus specifies shapes of open diagrams in a monoidal category of profunctors. A similar idea of defining a theory in which contexts of diagrams exist motivates Monoidal Context Theory [86]. It is both influenced by the theory of lenses and comb diagrams as well as work on a splicing-contour adjunction [74] which expresses processes with "gaps" into which other processes can plug into. In our work the notion of diagrams with holes is a very important one, because the notion of double-pushout rewriting includes graphs with holes (as the context graph of the rewrite). As we work with graph embeddings, graphs with holes are not a suitable structure, but we are able to represent them by introducing the auxiliary structure of *boundary vertices* which stand for holes in a graph.

The main difference between these related works and our project is that we care about monoidal categories without implicit symmetries and thus the order of edges around each

vertex matters very much. Open edges are not a suitable notion in our framework as composing them with each other may violate the surface embedding property of a graph. We will take a different approach to represent the outside face of a graph which influences the objects and morphisms in the graph category. This approach is convenient to encode context graphs which have holes in them, too.

Chapter 2

A Category of Surface-Embedded Graphs

2.1 Closing open systems

2.1.1 Open graphs

Monoidal categories can describe systems with a number of inputs and outputs over which they communicate with their environment. These input and output channels are a diagram's *interface* (or: *boundary*) with its surrounding environment. Information flows in both directions: the environment may provide data via the system's inputs and the system may return data to the environment via its outputs. When composing two diagrams, in sequence or in parallel, their boundaries are combined. Sequential composition is well defined only if the inputs and outputs at the composition boundary match. A similar restriction holds for substitution of diagrams: A diagram can be inserted into the hole of another diagram (which is called a *context*) if the interfaces of the hole in the context and the diagram match. For the combinatorial representation of the particular class of diagrams we are interested in, the implementation of interfaces of graphs — both on the outside and on the inside in form of a hole — has to ensure that the topological properties are preserved when they are composed. Therefore we will introduce explicit structures to represent the boundary of graphs and graphs with holes.

In standard notions of graphs, interfaces are typically represented by certain *open* edges, which are connected to a vertex at one of their ends only. The set of open edges of a graph constitutes the graph's interface. When composing two graphs with each other in sequence, the open edges at the composition boundary will be identified and may be closed (i.e. connected at

both of their ends) during the operation.

As we are interested not only in the connectivity information of a diagram but also its topological properties, we will work with graph *embeddings* instead of graphs. In this setting the notion of open graphs is not suitable as a representation of their interfaces any more.

Any open edge in a graph has to be contained in one of the faces of any of the graph's surface embeddings. As only one of the edge's ends is attached to a vertex, the edge does not carry any information about the graph's surface embedding. See Figure 2.1 for an illustration.

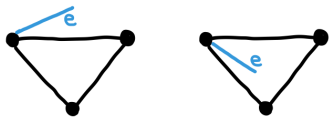


Figure 2.1: Different positioning of an open edge (in blue) in the same graph embedding.

Whereas this property by itself is harmless for merely *defining* surface-embedded graphs, when specifying their composition operation it may create complications with the calculation of the surface embedding of the resulting graph. Composition of graphs treats the edges at the interfaces as (unordered) sets and therefore may arbitrarily connect them with each other. This operation can interfere with the topology of the resulting graph embedding. Figure 2.2 illustrates two possible outcomes of a composition with open edges, one of which results in a plane graph and one which does not. As we are interested in the preservation of a certain genus of the surface of the graph embeddings involved, we will have to exclude this behaviour in our framework. Our aim is to express this property as part of the definition of graph embedding instead of an extrinsic property (such as an ordering of interface edges).

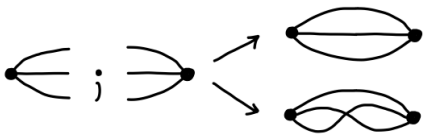
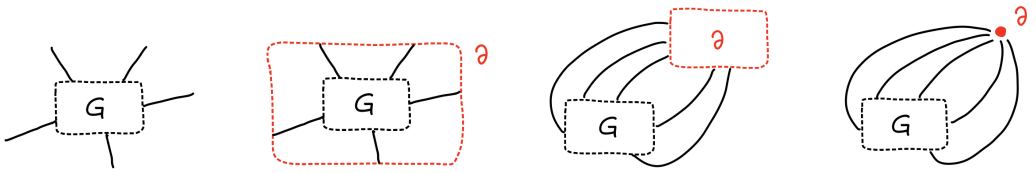


Figure 2.2: Composing two open graphs may violate their embedding property.

The issue of not being able to control the positioning of open edges becomes apparent in the rotation system representation of graph embeddings: rotation systems define the ordering of edges around *vertices*, but open edges are not attached to vertices on both ends. A rotation system does not have access to the dangling end of an open edge and therefore we are unable to impose an order on them. We require some additional structure to establish an order for the input and output edges.

This chapter is about a notion of graph to which we can add rotation information easily.



(a) A graph embedding including open edges. (b) Marking the graph's outside face. (c) Draw the outside as an enclosed face. (d) Introducing an auxiliary boundary vertex.

Figure 2.3: Introduction of a boundary vertex to represent the outside face of a graph.

This requirement imposes certain restrictions on the shape of graph, one of which being the treatment of the interface edges. Overall, we will aim to avoid the use of open edges altogether in our framework.

The outside face of a graph embedding To be able to incorporate embedding information about the open edges of a graph, we will introduce an auxiliary structure that represents the graph's *outside*.

Recall that the embedding of a graph into a surface is determined by its faces. Any closed cycle of edges determines a face of the embedding. If we consider an embedding in the plane, there is one face that is special to the others: the *outside* face of a graph. In the case of a plane string diagram, we observe that this outside faces contains all input and output edges of the corresponding graph. This is illustrated in Figure 2.3a. We can separate the graph from its outside face spatially by drawing an enclosing box around the graph, see Figure 2.3b. The graph's interface edges are completely contained within this box and their open ends are attached to it. With this separation we have created a bipartition of the surface with the graph inside the box and its *boundary* face, denoted ∂, being positioned outside of the box and taking up the rest of the plane.

We recall Proposition 1.52 and consider an embedding of the boxed graph on the sphere. This embedding still defines a bipartition of the surface, with the boundary face now taking up the remainder of the sphere. Because of the geometry of the sphere, this remainder is itself a region homeomorphic to a disc. Therefore we can create an equivalent drawing of the graph embedding, with both the graph and the outer face begin enclosed in a box, see Figure 2.3c. Crucially, the interface edges are still attached to the outside region ∂. These edges now connect the bipartite regions of the surface containing the graph and its boundary, respectively.

Remark 2.4. In general, the *outside* face is merely a choice of face of a graph embedding. In the special case of embeddings in the plane there is a canonical choice of outside face, but this is

not the case for an embedding on the sphere (or any other, higher genus, surface). For a plane graph embedding the choice of centre for the stereographic projection (cf. Proposition 1.47) defines its outside face. Put differently: the choice of outside face is the key difference between a plane graph on a sphere and in the plane.

In a last step we turn the auxiliary structure (i.e. the box) representing a graph's outside face into an element of the embedding itself: we introduce a special vertex which we call the *boundary vertex*. Graphically, the boundary vertex is created by contracting the (empty) outside region to a single vertex. After this operation, the interface edges of the graph are connected to its boundary vertex, as shown in Figure 2.3d. The boundary vertex can be seen as an auxiliary vertex *at infinity* to which all open edges are connected [31].

We have used the intuition about *plane* embeddings to introduce these special faces of graph embeddings explicitly. The resulting definition can accommodate higher genus surface embeddings though, not just the plane ones. A graph's context is *a region* of the surface the graph is embedded in, represented by a boundary vertex. This might be a simplification for certain higher genus surfaces, and we will discuss generalisations of the framework in Section 2.3.3. For now, we focus on structures with a disc-like context graph. Crucially, the interface edges between a graph and its context are explicit in the representation with a boundary vertex.

The boundary vertex is an element of the vertex set of the graph, but we still interpret it as a *distinguished* vertex as some graph operations act differently on the boundary vertex than on the other vertices. As the boundary of a graph is part of the graph structure itself, we can treat all edges in the graph uniformly and attach embedding information in form of rotation systems to all vertices *including* the boundary vertex. Further, closing the open edges of the graph (by the introduction of a boundary vertex) guarantees that all graphs will be total in our formalism which greatly simplifies graph operations.

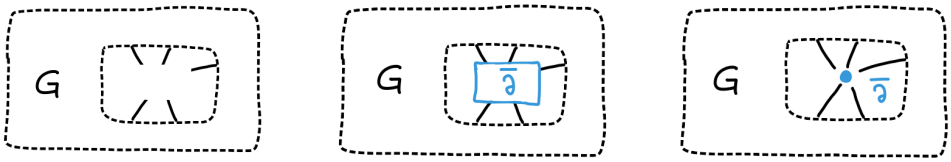
Throughout this work, we assume the boundary region not to contain any further information apart from the specification of the interface of the graph. When embedding an open graph into a context, the boundary region will be replaced by a context graph while ensuring that the interface edges with the original graph stay unchanged. This operation is non-trivial as it consists of replacing the boundary vertex of a graph with an entire graph. This is content of Section 2.2.

Remark 2.5. When we attach the open ends of edges to a specific region in a graph, we have ensured that these edge are part of the interface of that region. But additionally we also know

the contrary: no other edges interact with it. A graph can interact through edges at its boundary only.

2.1.2 Graphs with a hole

Open edges can model not only the relation with an outside environment of a graph. Sometimes we are interested in a similar kind of relationship on the inside: graphs can have *holes* in them. Holes are places where a subgraph is removed with the aim to insert another subgraph, for example in a graph rewriting step. Similar to composition at the outside boundary, this *substitution* operation has to respect the interface type. For the same reason as before, open edges are not a suitable structure. As graphs with holes need to be explicit structures in our category, we need to deal with the open edges around the hole. Conveniently, we already have a strategy to represent open edges at hand! We propose a very similar solution of closing the open edges around the hole by the introduction of an additional vertex to the graph: the *dual boundary vertex*. The construction of this vertex is analogous to the boundary vertex and schematically shown in Figure 2.6: Given a graph with a hole (Figure 2.6a), we can represent the hole as an explicit region with all interface edges attached to it (Figure 2.6b). Contracting this explicitly drawn region to a single vertex generates the dual boundary vertex (Figure 2.6c). Analogous to the boundary vertex, the dual boundary vertex acts as a placeholder for a potential subgraph to be inserted and ensures that the ordering of edges around the hole is preserved.



(a) Open edges around a hole inside G.

(b) Attach edges to an auxiliary region.

(c) Introduction of a dual boundary vertex $\bar{\partial}$.

Figure 2.6: Introduction of a dual boundary vertex to represent a hole inside a graph G.

Duality of inside and outside boundaries Embeddings of graphs are defined by the arrangement of their faces on the surface. In a general setting, there is no difference between the properties of the faces, they are all *a* face of the embedding. In particular, we may not have a distinction between the *outside* and *inside* of the embedded graph by default (cf. Remark 2.4. In such a framework we are not be able to identify any of the faces as the outside face or an inside face representing a hole. Similarly, we are not be able to recognise the boundary and dual boundary vertices as special vertices. Defining an outside face consists of a pointer to one

of the faces of the embedding. Additionally, this choice characterises every other region of the surface as an inside face. In our framework the distinction between the outside face and a hole in the graph is crucial because in the context of string diagrams we need to be able to distinguish between a context graph and a subgraph. Graph operations such as rewriting and composition are defined differently on the outside and inside of a graph. Embedding a graph into a context corresponds to composition at its *outside* boundary and inserting subgraphs is implemented as composition at its *inside* boundary. We therefore insist on the distinction of the two different kinds of boundaries and boundary vertices throughout this work.

Remark 2.7. For now we will assume graphs to have at most one outside and one hole. We consider graphs with multiple inside or outside interfaces as an interesting future project and discuss it further in Section 2.3.3.

2.1.3 Boundary graphs

Combining the concepts of boundary and dual boundary vertices motivates the notion of *boundary graph*. This graph consists of one boundary vertex, one dual boundary vertex, and a set of edges between them. This simple shape specifies a bipartition of a graph, with a number of edges between the parts. We will use boundary graphs as the skeleton for composing a context graph with a subgraph while preserving the edges at the composition boundary. Embedding a graph into a context means replacing the graph's boundary vertex with the context. Similarly, inserting a subgraph into a context amounts to replacing the dual boundary vertex of the context with the subgraph. Boundary graphs will be formally specified in Definition 2.39.

2.2 A suitable category of graphs

In this section we define a category of directed graphs with boundaries. Graphs do not store any embedding information at this point, but they are designed to accommodate topological structure in form of rotation systems in a straightforward manner. We will add rotation systems to these kind of graphs in Section 2.4. The main challenge for the specification of these kinds of graphs is a suitable correct notion of graph morphism. We have particular applications for graphs and their morphisms in mind which impose restrictions on their precise definition. Firstly, as discussed in Section 2.1 we use distinguished vertices for all boundaries in the graph. We benefit from this notion in that all the graphs in the category are total, but it has a significant impact on the definition of graph morphism, as their vertex component has to be a *partial* map.

Secondly, the interpretation of graphs as terms in a monoidal category as well as the definition of double pushout rewriting require a notion of *injective* morphism. Lastly, in order to allow the addition of rotation systems to a graph we need to ensure that the number of edges around a vertex does not change when applying a graph morphism. Conventional graph rewriting does not need to worry about this property, but for the specification of this category of graph embeddings, we will have to take it into account.

We start by recalling the standard category of graphs and graph morphisms, and successively modify the notion of morphism to meet all the required properties.

Definition 2.8. A total graph is a functor $G : (\bullet \rightrightarrows \bullet) \to \mathbf{Set}$. Concretely, such a graph is a pair of sets V and E, of *vertices* and *edges* respectively, and a pair of functions s and t assigning *source* and *target* vertices to each edge:

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

In the functor category $[\bullet \rightrightarrows \bullet, \mathbf{Set}]$, a morphism of graphs $(V, E, s, t) \to (V', E', s', t')$ is a pair of functions $f_V : V \to V'$, $f_E : E \to E'$, such that the following squares commute:

$$\begin{array}{ccc} E & \xrightarrow{f_E} & E' \\ {\scriptstyle s}\downarrow & & \downarrow{\scriptstyle s'} \\ V & \xrightarrow[f_V]{} & V' \end{array} \qquad \begin{array}{ccc} E & \xrightarrow{f_E} & E' \\ {\scriptstyle t}\downarrow & & \downarrow{\scriptstyle t'} \\ V & \xrightarrow[f_V]{} & V' \end{array} \tag{2.8.1}$$

With our notion of graphs with boundary vertices, embedding a graph into a context, or inserting a subgraph into a hole requires *replacing* the boundary or dual boundary vertex with a graph. Therefore we need to consider graph morphisms that forget these vertices, hence they need to have a *partial* vertex component. An example of a graph morphism replacing the boundary vertex with a graph is shown in Figure 2.9. Here, the morphism forgets about the boundary vertex ∂, but it does remember the three edges attached to ∂.

Figure 2.9: Example of replacing the boundary vertex with a graph.

To relax the notion of morphism we consider working in the subcategory $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]$ of total graphs and *partial* graph morphisms. This framework allows the vertex map to be partial,

but otherwise its behaviour does not quite match our requirements. Commutation of the naturality squares (Equation 2.8.1) in $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]$ is *strict*: it includes equality of the domains. This means that if a morphism forgets a vertex it must also forget all the edges incident at this vertex to meet the commutation condition, which is of no use to us. To accommodate this behaviour we use the poset enrichment of $\mathbf{Pfn}$ (remember Example 1.30) and consider a less restrictive version of the functor category. Note that this is still an intermediate stage of the development and we will have to restrict the category further to match our requirements.

Definition 2.10. The category $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]_{\leq}$ has as objects functors:

$$G = (E, V, s, t) : (\bullet \rightrightarrows \bullet) \to \mathbf{Pfn}$$

similar to Definition 2.8, and as morphisms $G \to G'$ pairs of functions (f_E, f_V) which are equipped with the following *lax* natural transformations:

$$\begin{array}{ccc} E & \xrightarrow{f_E} & E' \\ {\scriptstyle s}\downarrow & \leq & \downarrow{\scriptstyle s'} \\ V & \xrightarrow[f_V]{} & V' \end{array} \qquad\qquad \begin{array}{ccc} E & \xrightarrow{f_E} & E' \\ {\scriptstyle t}\downarrow & \leq & \downarrow{\scriptstyle t'} \\ V & \xrightarrow[f_V]{} & V' \end{array} \tag{2.10.1}$$

The lax commutation allows the vertex component of a morphism to be undefined at some vertex v while its incident edges may be preserved. In this case, the square in Equation 2.10.1 commutes by the top path being defined and the bottom path not defined. Conversely, if an edge is "forgotten" then its source and target vertices must also be so. Eventually we will need a slight refinement of this condition to be able to include all relevant cases, but let us take $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]_{\leq}$ as our ambient category for now.

In addition to partiality, we will need to incorporate two further properties of our notion of graph morphism:

1. Graphs are intended to represent terms in string diagrams of a monoidal category. The graph's vertices represent boxes in the string diagram which, in turn, define morphisms of the category. When we apply a function to a string diagram, the arity of its boxes, i.e. the number and type of edges around vertices, must never change. This preservation of structure is not guaranteed by conventional graph rewriting.

2. For classifying graph morphisms as *embeddings* we need to specify an injectivity property on them. Merely asking for injectivity of both vertex and edge component will not suffice: if the edge component is an injective function, we are unable to represent certain string diagrams, for example the identity morphism as illustrated in Example 2.13.

Example 2.11. The *identity* graph on one object consists of no vertex, and a single edge which has both ends attached to the "outside". This amounts to the boundary vertex with a self-loop. The name identity graph comes from the identity morphism for an object in a monoidal category whose string diagram is precisely a single edge. Figure 2.12 shows the identity graph. The boundary vertex with n nested self-loops may be called n-identity graph in correspondence with the tensor product of identity morphisms in a monoidal category.

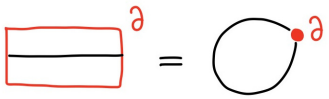


Figure 2.12: The identity graph.

Example 2.13. With the notion of identity graphs, our framework will need to allow for graph morphisms that encode embeddings into the identity context. An example of such a morphism is shown in Figure 2.14.

Figure 2.14: Embed a graph into the identity context.

This morphism replaces the boundary vertex of a graph with the identity graph, and therefore connects the two incident edges at the boundary vertex. Note that this morphism has a non-injective edge component.

Both requirements, preservation of vertex arity and a suitable notion of injective morphism, are properties of the incident edges around vertices. We will therefore make these connection points of edges at their source and target vertices (called *flags*) explicit and characterise properties of a graph morphism in terms of the flags involved.

Definition 2.15. Given a graph (V, E, s, t), its set of *flags* $F \subseteq (E \times V) + (E \times V)$ is defined as

$$F = \{(e, s(e)) \mid e \in E\} \uplus \{(e, t(e)) \mid e \in E\}.$$

Given a graph morphism $f : G \to G'$ there is an induced *flag map*, $f_F : F \to F'$,

$$f_F = (f_E \times f_V) + (f_E \times f_V).$$

Note that the flag map is in general a partial map: it is undefined on (e, v), whenever f_V is undefined on v or f_E is undefined on e. Whenever f_F is injective we say that f is *flag-injective*.

Flag injectivity allows for edges to be combined but prevents a morphism from decreasing the degree of a vertex in the process, hence it provides the first important attribute for the notion of graph embedding:

Lemma 2.16. *Let $f : G \to G'$ be a flag-injective graph morphism, $v \in V(G)$, and $f_V(v)$ defined. Then* $\deg v \leq \deg f_V(v)$.

Proof. The edges incident at v are given by the disjoint union of $s^{-1}(v)$ and $t^{-1}(v)$. Since $f_V(v)$ is defined, Equation 2.10.1 has to hold strictly for all incident edges at v: As the bottom-left path is defined, so is the top-right path, thus f_E is defined for all incident edges at v. Since f is flag-injective, f_E is injective these edges. Thus the number of flags at v does not decrease when applying f to G. □

Flag injectivity does not prevent a morphism from increasing the degree of a vertex: for this we require a notion of *flag surjectivity*. Given $f : G \to G'$, it does not suffice to require the flag map f_F to be surjective which expresses the fact that the *total* number of flags in the graph does not increase. This notion is of no use to us as we want to consider embeddings of smaller graphs into larger ones which is not a surjective operation. Our alternative definition of flat surjectivity does not look particularly intuitive, but we can approach it from pre-existing properties: The lax commutation of Equation 2.10.1 defines a property on *edges*, and is therefore missing certain special cases, for example when a vertex has no edges attached to it. To ensure the preservation of this vertex' arity, flag surjectivity has to be formulated as a property on *vertices*. We therefore modify Equation 2.10.1 by considering the *preimage* of source and target functions. The resulting lax equation is the definition of flag surjectivity:

Definition 2.17. Let $f : G \to G'$ be a morphism between the total graphs G and G'. We say that f is *flag-surjective* if the following two diagrams commute laxly:

$$\begin{array}{ccc} V & \xrightarrow{f_V} & V' \\ {\scriptstyle s^{-1}}\downarrow & \geq & \downarrow{\scriptstyle s'^{-1}} \\ P(E) & \xrightarrow[P(f_E)]{} & P(E') \end{array} \qquad \begin{array}{ccc} V & \xrightarrow{f_V} & V' \\ {\scriptstyle t^{-1}}\downarrow & \geq & \downarrow{\scriptstyle t'^{-1}} \\ P(E) & \xrightarrow[P(f_E)]{} & P(E') \end{array} \tag{2.17.1}$$

where s^{-1} and t^{-1} are the preimage maps of s and t respectively, and P is the powerset functor.

We show that flag-surjective graph morphisms, together with the condition that the edge map is total, defines a subcategory of the lax functor category $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]_{\leq}$:

Proposition 2.18. *Let $f : G \to G'$ be a graph morphism. If f is flag-surjective, and f_E is total, Equation 2.10.1 holds.*

Proof. Let $v \in V$. We show that if Equation 2.17.1 holds for v, then Equation 2.10.1 holds for all $e \in s^{-1}(v)$ under the assumption that f_E is total. (The proof works analogously for the target map t). We observe that, because f_E is total, the left-bottom path in Equation 2.17.1 is always defined. Similarly, the top-right path in Equation 2.10.1 is always defined. We distinguish the remaining two cases in which f is a flag surjection:

- If $(s'^{-1} \circ f_V)(v)$ is undefined, then $f_V(v)$ is undefined, because s'^{-1} is total. Then Equation 2.10.1 holds laxly immediately for any edge $e \in s^{-1}(v)$.
- If the square in Equation 2.17.1 commutes strictly, we have $(s'^{-1} \circ f_V)(v) = (P(f_E) \circ s^{-1})(v)$. This equations states that, for any edge $e \in s^{-1}(v)$, its image across f_E has $f_V(v)$ as its source vertex, i.e. its image is an element of the set $s'^{-1}(f_V(v))$. If we express this observation as a condition on the edge e itself, we obtain $(f_V \circ s)(e) = (s' \circ f_E)(e)$, which is precisely the property specified by Equation 2.10.1.

□

Next we show that a flag-surjective morphism does not increase the degree of flags at any vertex in its domain.

Lemma 2.19. *Let $f : G \to G'$ be a flag-surjective graph morphism, $v \in V(G)$, and $f_V(v)$ defined. Then* $\deg v \geq \deg f_V(v)$.

Proof. Let $v' = f_V(v)$. The edges incident at v' are given by the disjoint union of $s^{-1}(v')$ and $t^{-1}(v')$. Because f is flag-surjective, all the flags at v' are in the image of $f_E(s^{-1}(v)) + f_E(t^{-1}(v))$. Since $f_V(v)$ is defined, f_E is defined for all $e \in s^{-1}(v)$ and $e \in t^{-1}(v)$ by Equation 2.10.1. Therefore the degree of v does not increase when we apply f to G. □

Example 2.20. The graph morphism shown in Figure 2.21 is not flag-surjective. The degree of the vertex involved decreases, therefore the morphism is not flag-surjective by Lemma 2.19.

We call a morphism which is both flag-injective and flag-surjective a *flag bijection.* This is quite a strong property; it is almost enough to make the vertex map injective, but not quite.

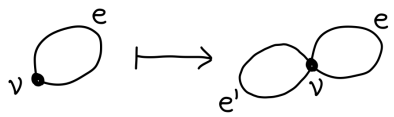
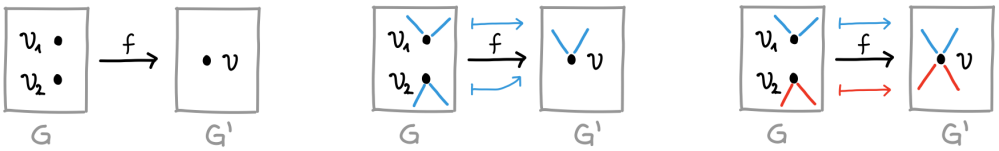


Figure 2.21: A graph morphism which is not flag-surjective.

Flags are defined as *pairs* of vertices and edges, therefore properties on flags only apply if a graph contains vertices *and* edges. In the special case of vertices without any edges incident (and therefore no flags associated with them), a flag-injective graph morphism may identify them. This situation is illustrated in Figure 2.22a and expressed by the following lemma.



(a) Valid flag bijection. (b) Not flag-injective. (c) Not flag-surjective.

Figure 2.22: (Counter-) Examples of graph morphisms as described in Lemma 2.23.

Lemma 2.23. *Let $f : G \to G'$ be a flag bijection, and suppose that $f_V(v_1) = f_V(v_2)$ and both are defined; then* $\deg v_1 = \deg v_2 = 0$.

Proof. Let $v' = f_V(v_1) = f_V(v_2)$; since f is flag-injective, the set of flags at v' must contain (the image of) the disjoint union of the flags at v_1 and v_2; hence $\deg v' \geq \deg v_1 + \deg v_2$. Flag injectivity rules out situations as illustrated in Figures 2.22b where two vertices with edges attached cannot be identified. Since (by Equation 2.10.1) f_E is defined on all the flags at v_1, flag surjectivity implies that $\deg v_1 \geq \deg v'$, and similarly for v_2. Thus flag surjectivity prevents the identification of two vertices with edges attached, as shown in Figure 2.22c. Overall we have that if two vertices v_1 and v_2 are identified by f_V, we have $\deg v' = \deg v_1 = \deg v_2 = 0$. □

We now have the necessary structure to prove that a flag-bijective morphism does not change the degree of any vertex in its domain:

Lemma 2.24. *Let G and G' be total graphs, and let $f : G \to G'$ be a flag bijection. For all $v \in V$, if $f_V(v)$ is defined, then* $\deg v = \deg f_V(v)$.

Proof. This statement combines Lemma 2.16, Lemma 2.19, and Lemma 2.23. □

We can organise graphs and flag bijections in a category by observing that composing two of them together yields a flag bijection again:

Lemma 2.25. *Let $f : G \to H$ and $g : H \to J$ be flag bijections; then $g \circ f$ is a flag bijection.*

Proof. (1) flag injectivity: Assume injectivity of the flag maps induced by f and g. If f_V is undefined on a vertex v, so is the flag map f_F on (v, e) for any e incident at v. Consider flags (e, v) and (e', v') where $(f_E \times f_V)$ is defined, $v = s(e)$, $v' = s(e')$, and assume $g_F(f_F(e, v)) = g_F(f_F(e', v'))$. Because f is a flag surjection and defined on the given flags, Equation 2.17.1 commutes strictly on v and v'. Therefore, $f_E(e) = s_H(f_V(v))$ and $f_E(e') = s_H(f_V(v'))$. We can apply flag injectivity of g to get $f_F(e, v) = f_F(e', v')$, and flag injectivity of f to reach $(e, v) = (e', v')$. The same argument applies to the target map.

(2) flag surjectivity: Assume lax commutation of Equation 2.17.1 for f and g and show that the composite diagram also commutes laxly:

$$\begin{array}{ccccc} V_G & \xrightarrow{f_V} & V_H & \xrightarrow{g_V} & V_J \\ {\scriptstyle s_G^{-1}}\downarrow & \geq & {\scriptstyle s_H^{-1}}\downarrow & \geq & \downarrow{\scriptstyle s_J^{-1}} \\ P(E_G) & \xrightarrow[P(f_E)]{} & P(E_H) & \xrightarrow[P(g_E)]{} & P(E_J) \end{array} \qquad \begin{array}{ccccc} V_G & \xrightarrow{f_V} & V_H & \xrightarrow{g_V} & V_J \\ {\scriptstyle s_G^{-1}}\downarrow & & \geq & & \downarrow{\scriptstyle s_J^{-1}} \\ P(E_G) & \xrightarrow[P(f_E)]{} & P(E_H) & \xrightarrow[P(g_E)]{} & P(E_J) \end{array}$$

In the case of either f_V or g_V being undefined, the composite $(g_V \circ f_V)$ is also undefined and the diagram commutes laxly immediately. If both f_V and g_V are defined, both their diagrams commute strictly, and by diagram gluing, their composite does as well. □

Lemma 2.26. *The identity graph morphism is a flag bijection.*

Proposition 2.27. *Total graphs and flag bijections define a wide subcategory of $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]_{\leq}$, which we will call $\mathbf{B}$.*

In Example 2.13 we have seen a special case of graph morphism: the vertex of a self loop can be forgotten (meaning the vertex is not in the image of the morphism). Loops, attached to a vertex as self-loops, or not being attached to any vertex at all, are a very degenerate case of graph and diagram. Nevertheless, they are necessary structures, because they express diagrams like the identity, cups or caps. Especially for the construction of double pushout rewriting diagrams loops require quite a lot of care and special treatment. In the definition of graphs with circles, we treat edge loops that do not contain any vertices as a separate set of circles O.

Definition 2.28. A *graph with circles* is a 5-tuple $G = (V, E, O, s, t)$ where (V, E, s, t) is a total graph and O is a set of *circles*. For notational convenience we define the set of *arcs* as the disjoint union $A = E + O$.

A morphism $f : G \to G'$ between two graphs with circles consists of two functions, a vertex map $f_V : V \to V'$ and a map on arcs $f_A : A \to A'$, satisfying the conditions listed below. We can present any f_A as four maps:

$$f_E : E \to E' \qquad f_{EO} : E \to O' \qquad f_O : O \to O' \qquad f_{OE} : O \to E'$$

The following conditions must be satisfied:

1. $f_A : A \to A'$ is total.
2. The component $f_{OE} : O \to E'$ is the empty function.
3. The pair (f_V, f_E) forms a flag surjection between the underlying graphs in $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]_{\leq}$.

If, additionally, the following four conditions are satisfied, we call the morphism an *embedding*:

4. $f_V : V \to V'$ is injective,
5. The component f_O is injective,
6. The component f_{EO} is *circle-injective*, see Definition 2.32.
7. The pair (f_V, f_E) forms a flag bijection between the underlying graphs in $[\bullet \rightrightarrows \bullet, \mathbf{Pfn}]_{\leq}$.

Remark 2.29. It's worth noticing that if some f_A maps an edge e to a circle, then $f_E(e)$ is undefined, but $f_{EO}(e)$ is defined. This, by the lax naturality property, implies that f_V is undefined on both $s(e)$ and $t(e)$.

Example 2.30. Let $G = (\{v\}, \{e\}, \varnothing, \{e \mapsto v\}, \{e \mapsto v\})$ be the (unique) total graph with circles with one vertex and one edge; let $G' = (\varnothing, \varnothing, \{e\}, \varnothing, \varnothing)$ be the graph with circles with no vertices and a single circle. Define $f : G \to G'$ by $f_V = \varnothing$ and $f_{EO} = \mathrm{id}_e$. This is a valid embedding in $\mathbf{G}$.

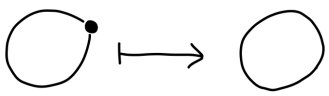


Figure 2.31: Example of a morphism $f_{EO} : E \to O'$ in $\mathbf{G}$.

Circle injectivity In the definition of graphs with circles we allow morphisms to transform an edge $e \in E$ attached to vertices into a circle $o \in O$. This may happen when the morphism forgets about e's source and target vertices (which could coincide). In the definition of graph embedding we have to consider this corner case by restricting the component f_{EO} by an injectivity constraint:

Definition 2.32. A graph morphism $f : G \to G'$ is called *circle-injective* if any two edges $e_1, e_2 \in G$ are mapped to the same circle $o \in O$ (by the component f_{EO}) only if they share at least one of their endpoints in G.

Figure 2.33a shows an example of a circle-injective morphism. This is a valid embedding in $\mathbf{G}$. In contrast, Figure 2.33a illustrates a graph morphisms which is not circle-injective, and therefore not a valid embedding.

(a) A circle-injective morphism. (b) A non circle-injective morphism.

Figure 2.33: Examples of graph morphisms creating circles, $f_{EO} : E \to O'$.

To organise graphs with circles in a category we need the following two properties.

Lemma 2.34. *Defining composition point-wise, the composite of two morphisms of graphs with circles is again such a morphism. Additionally, if both morphisms are embeddings, their composition is an embedding as well.*

Proof. Let $f : G \to G'$ and $g : G' \to G''$ be two morphisms; then $g \circ f = ((g_{V'} \circ f_V), (g_{A'} \circ f_A))$; since composition of partial functions is associative, we only need to check that the six properties of Definition 2.28 are preserved.

Conditions 1 and 4 follow from the properties of partial functions, and condition 7 (which includes condition 3) follows from Lemma 2.25. Observe that

$$\begin{aligned}(g \circ f)_O &= [g_{EO}, g_O] \circ (f_{OE} + f_O) \\ &= (g_{EO} \circ f_{OE}) + (g_O \circ f_O) \\ &= (g_{EO} \circ \varnothing) + (g_O \circ f_O) \\ &= g_O \circ f_O\end{aligned}$$

hence $(g \circ f)_O$ is injective since f_O and g_O are, satisfying condition 5. By a similar argument we have

$$\begin{aligned}(g \circ f)_{OE} &= [g_{OE}, g_O] \circ (f_E + f_{OE}) \\ &= (g_{OE} \circ f_E) + (g_O \circ f_{OE}) \\ &= (\varnothing \circ f_E) + (g_O \circ \varnothing) \\ &= \varnothing\end{aligned}$$

satisfying condition 2. Finally, we have

$$\begin{aligned}(g \circ f)_{EO} &= [g_{EO}, g_O] \circ (f_E + f_{EO}) \\ &= (g_{OE} \circ f_E) + (g_O \circ f_{EO}).\end{aligned}$$

Assume f and g to be embeddings. Then the left hand side is circle-injective: g_{OE} is circle-injective; f_E is not injective, but because f is flag-injective the only case in which f_E identifies two edges is when they share an endpoint which makes f_E circle-injective. On the right hand side we have a composite of the injective function g_O and the circle-injective function f_{EO} which is also circle-injective. Therefore, the remaining condition 6 is satisfied. □

We finally have introduced all the necessary structure to define our category of graphs.

Definition 2.35. Let $\mathbf{G}$ be the category whose objects are graphs with circles, and whose arrows are morphisms as per Definition 2.28.

There is an obvious and close relationship between the category $\mathbf{G}$ of graphs with circles and the category of partial graphs and flag bijections, $\mathbf{B}$. We can make this precise.

Definition 2.36. We define a forgetful functor $U : \mathbf{G} \to \mathbf{B}$ by

$$\begin{array}{ccc} U : (V, E, O, s, t) & \longmapsto & (V, E, s, t) \\ {\scriptstyle U:(f_V, f_A)}\downarrow & \mapsto & \downarrow{\scriptstyle (f_V, f_E)} \\ U : (V', E', O', s', t') & \longmapsto & (V', E', s', t') \end{array}$$

Example 2.37. Returning to Example 2.30, we see how the degenerate case of a single circle is treated by the forgetful functor. We start with G, the unique total graph with a single vertex and a single edge, illustrated in Figures 2.31. We will observe that there is only one valid way to forget about the vertex in G. Observe that $G'' = (\varnothing, \{e\}, \varnothing, \varnothing, \varnothing)$, the graph consisting

of single edge without any source or target vertex (but not a circle!), is not an object in **G**. However $G' = (\varnothing, \varnothing, \{e\}, \varnothing, \varnothing)$ *is* a valid graph, and the map $f : G \to G'$ which is undefined on the vertex and sends the edge to the circle is a valid morphism, is indeed the only valid one in **G**. Finally, observe that the image of UG' is the empty graph and Uf is the empty function.

The term "graph with circles" is unacceptably cumbersome, so henceforth we will simply say "graph" and refer to **G** as the category of graphs.

2.3 DPO rewriting

We now define the relevant structures to be able to implement double pushout rewriting in the category **G** of graphs. As explained in Section 1.3, the notion of adhesive category is not suitable for our purposes, since the monomorphisms of **G** are not useful for our purposes as they only consider the total subset of maps. We will instead consider a specific class of maps only, and for these maps show the existence of pushout and the existence and uniqueness of pushout complements which are similar properties to those of an adhesive category.

Remark 2.38. Most graph morphisms in this section are embedding of a small object into a larger one. Wherever unambiguous to do so, we will treat embeddings as actual inclusions, for example, we may write $m_E(e) = e$ despite the domain and codomain of the map being different graphs.

We will construct a DPO rewriting diagram such as in Equation 1.1, this time having in mind the structures we defined for explicit outside and inside faces, and boundaries between them. First, we formally define the notion of a boundary graph:

Definition 2.39. A *boundary graph* is a graph with exactly two vertices, called ∂ and $\bar{\partial}$ (boundary and dual boundary vertex), where $s(e) = \partial$ and $t(e) = \bar{\partial}$ (or vice versa) for all edges $e \in E$, and there are no circles.

We will use the notion of boundary graph to specify a bipartition between a graph G and its environment. The boundary vertex ∂ may be replaced with a concrete environment and the dual boundary vertex $\bar{\partial}$ with a concrete graph. The edges between the two vertices ∂ and $\bar{\partial}$ in the boundary graph encode the interface between the two structures.

Notation 2.40. Boundary and dual boundary vertices are usually called ∂ and $\bar{\partial}$. When they are used represent the boundary of a particular graph G we write ∂G or $\bar{\partial} G$. The corresponding boundary graph, encoding the bipartition of G and its environment is denoted $\partial\bar{\partial}_G$.

Boundary graphs are located in the top middle of a DPO diagram. They represent the joint outer boundary between two graphs involved in a rewrite rule. The boundary vertex is required to be preserved across both legs of the span, which ensures that all graphs involved share the same outer interface. The bottom half of a DPO diagram is the application of the rewrite rule within a bigger graph, or *context*. Therefore the context graph (in the bottom middle position) has to contain a hole which fits the subgraphs. The map c downwards from the boundary graph to the context graph is required to preserve the *dual* boundary vertex which ensure that the type of the hole stays unchanged. With this structure in place, inserting the left or the right graph into the hole in the context is well formed.

For the category $\mathbf{G}$ of graphs with circles we will express these characteristics of morphisms involved in a DPO diagram in the form of particular spans and composites. The definitions seem quite specific at first, but they define precisely the cases in which graph rewriting makes sense, ensuring that the types of subgraphs and composition boundaries match where needed, just as the original definition for DPO rewriting is motivated.

2.3.1 Pushouts

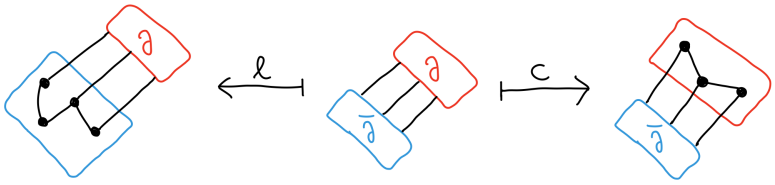
We now define a specific kind of span which we use to compute pushouts. Recall that boundary graphs consist of two vertices which specify a bipartition of a surface into one region for the graph and one for its context. A *partitioning span* encodes the instantiating of these regions with concrete graphs. This is implemented by the substitution of graphs for the boundary vertices. In particular, each leg of the span substitutes a graph for *one* of the vertices in the boundary graph, while leaving the other half the same.

Definition 2.41. A *partitioning span* is a span $L \xleftarrow{l} B \xrightarrow{c} C$ in $\mathbf{G}$, where B is a boundary graph, the vertex component l_V is defined on ∂ and undefined on $\bar{\partial}$ and, dually, c_V is undefined on ∂ and defined on $\bar{\partial}$. Further, we require l and c to be embeddings.

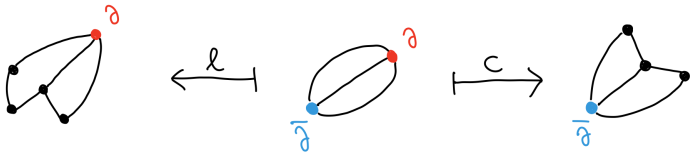
Remark 2.42. The names for the morphisms involved in a partitioning span originate from the names in the schema of DPO diagram, remember Equation 1.1.

An example of a partitioning span depicted in Figure 2.43.

Recall from Lemma 1.28 that the category of injective functions does not have pushouts. As graph embeddings are defined as injective maps, care is needed on the structure of the spans we use to calculate pushouts. Partitioning spans provide exactly the information we need: they ensure that the two graphs at the feet are non-overlapping as each leg is defined exactly one



(a) Boundary ∂ and hole $\bar{\partial}$ drawn as regions.



(b) Boundary ∂ and hole $\bar{\partial}$ drawn as vertices.

Figure 2.43: Example of a partitioning span, drawn in two different (but equivalent) ways.

half of the vertices. Gluing together the two subgraphs by computing the pushout therefore produces a well defined and sensible result.

In contrast to the boundary vertices, the *edges* of a boundary graph are not forgotten in the process of constructing a pushout. Instead they need to be preserved by all arrows in the pushout square to ensure that the interface between graph and context is preserved. Depending on the particular context or subgraph, two edges may be identified by one of the maps (cf. Example 2.13). This non-injectivity property on edges is a feature of our category. The following lemma shows that at most two edges can be identified by a graph morphism.

Lemma 2.44. *Let $L \xleftarrow{l} B \xrightarrow{c} C$ be a partitioning span and suppose that $e = l_E(e_1) = l_E(e_2)$ in L for distinct e_1 and e_2 in E_B. Then e is a self-loop at ∂ in L and for all other e_2 and e_3 with $e_1 \neq e_3$ and $e_1 \neq e_2$ we have $e \neq l_E(e_3)$. The same holds correspondingly for C and c. The two scenarios are depicted in Figure 2.45.*

Figure 2.45: Examples of a valid and a non-valid graph morphism in **G** involving boundary graphs.

Proof. By flag bijectivity, all flags at ∂ must be preserved, including the distinct flags for $l_E(e_1)$ and $l_E(e_2)$. By the hypothesis, these two edges are identified by l, thus we have that $s_B(e_1) = \partial$ and $s_B(e_2) = \bar{\partial}$ (or vice versa). Because $s_B(e_1) \neq s_B(e_2)$, e is a self loop at ∂ in L. If we suppose further that $l_E(e_3) = e$, then l would not be flag-bijective, which is a contradiction. □

Self-loops in the codomain graph of a partitioning span indicate that the boundary is connected to itself by an edge that does not include a vertex. This property is responsible for the failure of injectivity on edges and gives rise to degeneracies when constructing pushouts. To characterise these degeneracies, we study the structure of self loops at boundary vertices from a dual perspective.

Definition 2.46. The *pairing graph* for a partitioning span $L \xleftarrow{l} B \xrightarrow{c} C$ is a labelled directed graph whose vertices are E_B; each vertex receives a *polarity*: + if $s_B(e) = \partial$, – if $s_B(e) = \bar{\partial}$. We draw a *blue* edge between e_1 and e_2 if $l_E(e_1) = l_E(e_2)$ i.e. if e_1 and e_2 form self-loop in L; similarly we draw a *red* edge between e_1 and e_2 if they form a self-loop in C. Blue edges are directed from positive to negative polarity; red edges the reverse.

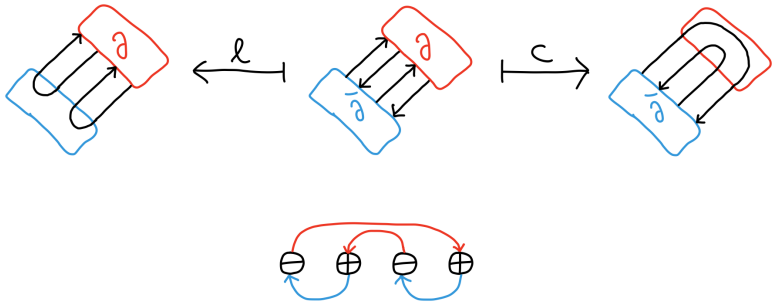


Figure 2.47: Example of a partitioning span with its pairing graph.

An example of a pairing graph is shown in Figure 2.47. The pairing graph is always bipartite: it is immediate from the definition that vertices of the same polarity are never connected. Further, due to Lemma 2.44, each vertex can have a maximum of one edge of each colour incident to it. In consequence every connected component is just a path, possibly of length zero, possibly a cycle. From these properties, we have the following immediate proposition.

Proposition 2.48. *Let* $\mathbf{P}$ *be the pairing graph of the partitioning span* $L \xleftarrow{l} B \xrightarrow{c} C$*; then each connected component* p *of* $\mathbf{P}$ *determines a path in* B*. For those components which are not cycles, if the first vertex of* p *is positive, then the path starts at* ∂*; if negative the path starts at* $\bar{\partial}$*. Conversely, if the last vertex of* p *is positive, the path ends at* $\bar{\partial}$ *and vice versa.*

When we form the pushout of a partitioning span, the components of the pairing graph determine which edges in B will be identified. This forms an intermediate result (Lemma 2.53) in the proof of the next theorem.

Theorem 2.49. *In* **G**, *pushouts of partitioning spans exist. Further, the maps into the pushout are embeddings.*

Proof. The proof will proceed via several intermediate results. First we will explicitly define the pushout candidate $L \overset{}{\underset{m}{\rightarrow}} G \overset{}{\underset{g}{\leftarrow}} C$ (Definition 2.51), show the constructed object G is a valid graph (Lemmas 2.52 and 2.54), show that m and g are indeed embeddings in **G** (Lemma 2.55), and finally show that the required universal property holds in in **G** (Lemma 2.56). □

An example of the pushout of a partitioning span is shown in Figure 2.50.

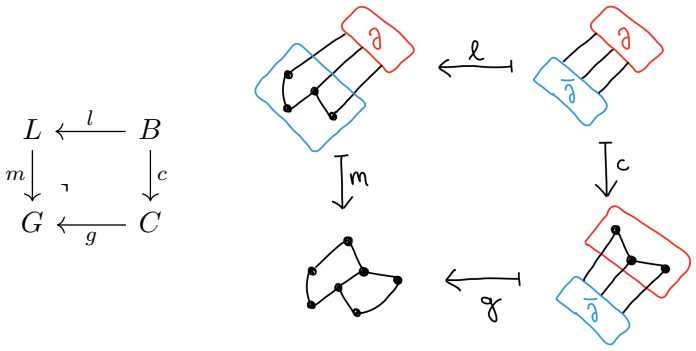


Figure 2.50: Pushout of the partitioning span from Figure 2.43.

Definition 2.51. Given the partitioning span $L \overset{l}{\leftarrow} B \overset{c}{\rightarrow} C$, we define the *pushout candidate* $L \overset{}{\underset{m}{\rightarrow}} G \overset{}{\underset{g}{\leftarrow}} C$ as follows.

We construct the underlying sets and functions by pushout in **Pfn**,

$$\begin{array}{ccc} V_L & \overset{l_V}{\leftarrow} & \{\partial, \bar{\partial}\} \\ {\scriptstyle m_V}\downarrow & \ulcorner & \downarrow{\scriptstyle c_V} \\ V_G & \underset{g_V}{\leftarrow} & V_C \end{array} \qquad \begin{array}{ccc} A_L & \overset{l_A}{\leftarrow} & E_B \\ {\scriptstyle m_A}\downarrow & \ulcorner & \downarrow{\scriptstyle c_A} \\ A_G & \underset{g_A}{\leftarrow} & A_C \end{array} \tag{2.51.1}$$

so explicitly we have

$$V_G = (V_L + V_C) \setminus \{\partial, \bar{\partial}\} \qquad A_G = (A_L + A_C)/\sim$$

where $\sim$ is the least equivalence relation such that $l_A(e) = c_A(e)$ for $e \in E_B$. Note that we do not need to quotient the set of vertices V_G as, by definition of partitioning span, l_V and c_V are

defined on disjoint subsets of B. Next we define the source map by

$$s_G(e') = \begin{cases} s_L(e) & \text{if } e' = m_A(e) \text{ and } s_L(e) \text{ is defined and } s_L(e) \neq \partial \\ s_C(e) & \text{if } e' = g_A(e) \text{ and } s_C(e) \text{ is defined and } s_C(e) \neq \bar{\partial} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.51.2}$$

for all $e' \in A_G$. The target map t_G is defined similarly. (Strictly speaking we have defined s and t on all of A; they will be restricted to E when we have defined the following.) Finally we divide the arcs into edges and circles by setting

$$E_G = \{e \in A_G : \text{ both } s_G(e) \text{ and } t_G(e) \text{ are defined}\} \tag{2.51.3}$$

$$O_G = A_G \setminus E_G \tag{2.51.4}$$

There are two properties that need to be checked to ensure that the definition above yields a valid graph. The source and target maps should be well defined partial functions; and all arcs should either have two end points (i.e. they are edges) or none (they are circles).

Lemma 2.52. *Equation* (2.51.2) *defines a partial function: if $s_G(e')$ is defined, it is single-valued.*

Proof. There are two things to check. First we show that if the first or second clause of the definition applies it is single valued. We then show that at most one of those clauses can apply.

Suppose that in L we have distinct e_1, e_2 such that $m_A(e_1) = m_A(e_2)$ and $s_L(e_1) \neq \partial$. Since they are distinct in L and identified in G, we must have distinct $e_1, e_2 \in B$ such that $c_A(e_1) = c_A(e_2)$ in C. By Lemma 2.44 this gives a self-loop at $\bar{\partial}$ in C, which in turn implies that $s_L(e_2) = \partial$. Hence L provides at most one candidate source vertex for every edge in G, and a similar argument can be made for C.

Now suppose $m_A(e_1) = g_A(e_2)$, and that $s_L(e_1) \neq \partial$ and $s_C(e_2) \neq \bar{\partial}$. Since the edges are identified in G they are both present in B. Since $s_L(e_1) \neq \partial$ we have $s_B(e_1) = \bar{\partial}$, from which $s_C(e_1) = \bar{\partial}$. Since $s_C(e_2) \neq \bar{\partial}$, e_1 and e_2 are distinct in C. Therefore we must have e_1 and e_2 identified in L; therefore, by Lemma 2.44, e_1 must be a self-loop at ∂ which contradicts our original assumption. Therefore there is at most one candidate source vertex and the map s_G is well defined in (2.51.2). □

The preceding argument applies equally to the target map t_G.

Lemma 2.53. *Let* $\mathbf{P}$ *be the pairing graph of the partitioning span* $L \xleftarrow{l} B \xrightarrow{c} C$*, and let* G *be its pushout candidate.*

1. *Suppose* e *and* e' *are edges in* B*; if* e *and* e' *are in the same component of* $\mathbf{P}$ *then* $(m_A \circ l_E)(e) = (m_A \circ l_E)(e')$*.*
2. *Let* e *be any arc in* A_G*; then its preimage in* B *is either empty or is exactly one connected component of* $\mathbf{P}$*.*

Proof. (1) Suppose that e and e' are the same component of $\mathbf{P}$. We use induction on the length of the path from e to e' in $\mathbf{P}$. If the path is length zero, then $e = e'$ and the property holds trivially. Otherwise, let e'' be the predecessor of e'. By induction, and (2.51.1), we have

$$(m_A \circ l_E)(e'') = (m_A \circ l_E)(e) = (g_A \circ c_E)(e) = (g_A \circ c_E)(e'')$$

Since e' and e'' are adjacent in $\mathbf{P}$ we must have either $l_E(e') = l_E(e'')$ or $c_E(e') = c_E(e'')$ depending on the colour of the edge. From this the result follows.

(2) Let $e \in A_G$ and suppose that $e_1 \in (m_A \circ l_E)^{-1}(e)$ in B. Either e_1 is a component on its own, or it has a neighbour e_2. By the definition of $\mathbf{P}$ either $l_E(e_1) = l_E(e_2)$ or $c_E(e_1) = c_E(e_2)$ depending on the colour of the edge. Therefore we have

$$(m_A \circ l_E)(e_2) = (m_A \circ l_E)(e_1) = e$$

so e_2 is also in the preimage of e. By induction, the entire component containing e_1 must also be included in the preimage.

For the converse, recall that $A_G = (A_L + A_C)/\sim$ where $\sim$ is the least equivalence relation such that $l_E(e_i) = c_E(e_i)$ for $e_i \in E_B$. Therefore if distinct e' and $e'' \in E_B$ both belong to the preimage of $e \in A_G$, there necessarily exists a chain of equalities

$$l_E(e') = l_E(e_1), \quad c_E(e_1) = c_E(e_2), \quad l_E(e_2) = l_E(e_3), \quad \ldots, \quad c_E(e_n) = c_E(e'')$$

to place them in the same equivalence class. Such a chain of equalities precisely defines a path from e' to e'' in $\mathbf{P}$, hence if two edges of B are identified in the pushout, they belong to the same component in the pairing graph. □

Lemma 2.54. *Let* G *be the pushout candidate defined above. For all arcs* $e \in A_G$ *either both* $s_G(e)$ *and* $t_G(e)$ *are defined or neither is.*

Proof. Consider the preimage of e in B; if it is empty then e is simply included in G from either L or C, along with both its end points.

Otherwise, by Lemma 2.53, e corresponds to a connected component p of the pairing graph $\mathbf{P}$. By Corollary 2.48 such components can be either line graphs or closed loops. If p is a closed loop, for all $e_i \in p$ we have

$$s_L(l_E(e_i)) = t_L(l_E(e_i)) = \partial \qquad \text{and} \qquad s_C(c_E(e_i)) = t_C(c_E(e_i)) = \bar{\partial}$$

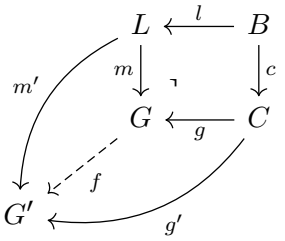
so, by Equation 2.51.2, neither $s_G(e)$ nor $t_G(e)$ is defined. If, on the other hand, p forms a path $e_1, e_2, \ldots, e_n$, its ends provide the source and target. Specifically, if e_1 positive in $\mathbf{P}$ then $s_C(c_E(e_1)) \neq \bar{\partial}$ and if it is negative $s_L(l_E(e_1)) \neq \partial$; if e_n is positive $t_L(l_E(e_n)) \neq \partial$, and if e_n is negative $t_C(c_E(e_n)) \neq \bar{\partial}$.

Hence $s_G(e)$ is defined if and only if $t_G(e)$ is defined. Therefore the division of A_G into edges and circles is correct and G is indeed a valid graph. □

Lemma 2.55. *The arrows of the cospan $L \xrightarrow[m]{} G \xleftarrow[g]{} C$ defined by the pushout candidate are embeddings in* $\mathbf{G}$.

Proof. We will show the result for m; the proof for g works analogously. Note that Properties 4 and 1 are automatic from the underlying pushouts in $\mathbf{Pfn}$. Since the graph B has no circles, the m_O component is injective by construction (Property 5) and since no arc gets a source or target in G unless its preimage had one, the component m_{OE} is empty as required (Property 2). Finally we have to show that the induced map (m_V, m_E) is a flag bijection. First note that if $m_E(e)$ is undefined then e is necessarily a self-loop at ∂, and $m_V(\partial)$ is always undefined, so the squares (cf. Equation 2.10.1) commute. Otherwise if $(f_V \circ s_L)(e)$ is defined then the square commutes directly by the definition of s_G above, and similarly for t_G. Finally, for all $v \in V_L$ with $v \neq \partial$, we have that $m_V(v)$ is defined. By the definition of s_G and t_G, e is a flag at v if and only if $m_E(e)$ is a flag at $m_V(v)$. Flag injectivity and flag surjectivity follow immediately. Hence m is an embedding in $\mathbf{G}$. □

Lemma 2.56. *the cospan $L \xrightarrow{m} G \xleftarrow{g} C$ has the required universal property.*



Proof. Since the underlying sets and functions are constructed via pushout the required mediating map $f = (f_V, f_A)$ exists; we need to show that it is a morphism of $\mathbf{G}$. Property 1 follows from m' and g' satisfying it as well. For the f_{OE} to be empty (Property 2), use the fact that m'_{OE} and g'_{OE} are empty for circles in L and C, because they are morphisms in $\mathbf{G}$. The remaining case for a circle to appear in G is as the pushout of some edges in B being identified, computed by using the corresponding pairing graph. In this case, because the outer square has to commute for the edge component, these edges have to be identified, and hence form a circle, in G', too. This makes f_{OE} empty. For flag surjectivity between the underlying graphs (Property 3), observe that the vertex set V_G is the disjoint union of vertex sets V_L and V_C. Because m' and g' are valid morphisms in $\mathbf{G}$, they are flag-surjective, and therefore so is f. □

Since pushouts of partitioning spans are the basis of the rewrite theory we wish to pursue, for the rest of this work the term "pushout" should be understood to imply "of partitioning span".

2.3.2 Pushout complements

We continue with the other required ingredient for DPO rewriting: pushout complements. Just as we did with partitioning spans for pushouts, we will introduce a specific kind of composite of two morphisms, called *boundary embedding*, for which the pushout complement exists. The concept is analogous to partitioning spans in that each morphism replaces *one half* (i.e. one of the two boundary vertices) of the boundary graph with a concrete graph.

Definition 2.58. A *boundary embedding* is a pair of maps $B \xrightarrow{l} L \xrightarrow{m} G$ in $\mathbf{G}$, where B is a boundary graph, where: (i) $l_V(\partial)$ is defined but $l_V(\bar{\partial})$ is undefined; and (ii) $(m_V \circ l_V)(\partial)$ is undefined. Further, L has to be a connected graph, and m an embedding.

See Figure 2.57 for an example of a boundary embedding. Similar to the case of pushouts,

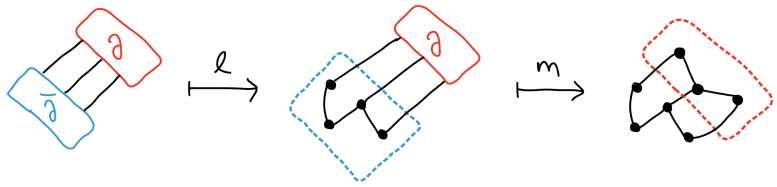


Figure 2.57: Example of a boundary embedding.

we have to think carefully about the structure of self-loops at boundary vertices for which we use pairing graphs once more. Boundary embeddings only encode one half of a pairing graph straight-forwardly, but as they also contain the pushout graph we are able to compute the other half. This is one instance of the *re-pairing problem*:

Definition 2.59. Given a boundary embedding $B \xrightarrow{l} L \xrightarrow{m} G$, we can immediately construct half a pairing graph $\mathbf{P}$, consisting of only the blue edges using the mapping $l : B \to L$. The *re-pairing problem* is to construct the other half (the red edges) so that the connected components map to the edges of G (cf. Lemma 2.53). See Figure 2.60 for examples.

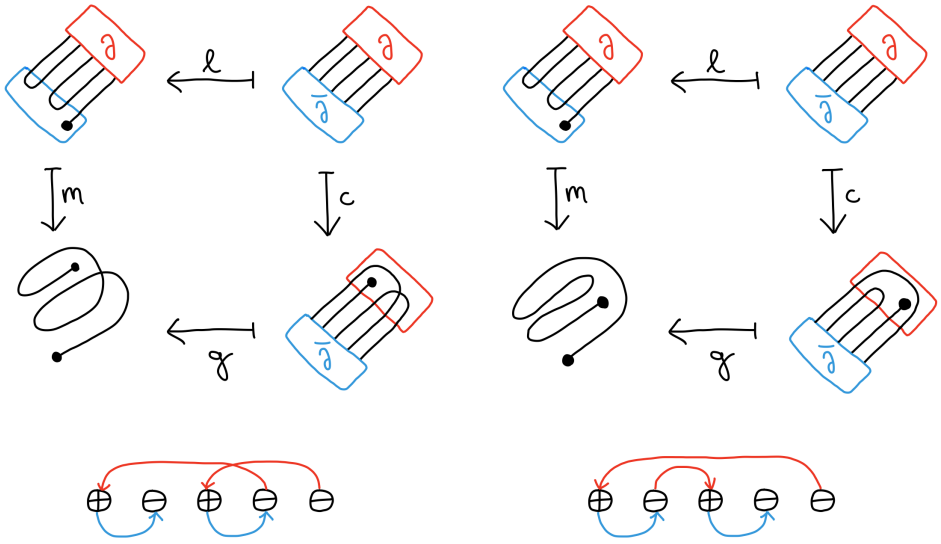


Figure 2.60: Two different solutions to the same re-pairing problem, together with the corresponding pairing graphs.

Lemma 2.61. *Given a boundary embedding $B \xrightarrow{l} L \xrightarrow{m} G$, a solution to the re-pairing problem always exists.*

Proof. Any half-pairing graph has connected components of at most two vertices, linked by a (blue) edge from a positive vertex to a negative one. Define the component of an arc by

$$k(a) = (m_A \circ l_A)^{-1}(a) \text{ for all } a \in A_G$$

Note that this defines a partition of the set $E_B \simeq \sum_{a \in A_G} k(a)$, and each (non-empty) $k(a)$ determines a connected component of the solution to the re-pairing problem. We abuse notation and use $k(a)$ to also denote the subgraph of the half-pairing graph whose vertices are $k(a)$. There are two cases depending whether a is a circle or an edge.

1. Suppose $a \in O_G$; we can form a closed loop involving all $e \in k(a)$, by adding red edges as follows. Pick a degree-one positive vertex p follow the incident blue edge to the negative vertex n; now pick another a degree-one positive vertex p' which is not connected to n. Add a red edge from n to p'. Repeat the process starting from p'. When no more vertices remain, close the loop by adding a red edge from the final negative vertex back to p. Since a is a circle, $k(a)$ necessarily contains an even number of vertices, so closing the loop is always possible.

2. The case when a is an edge is slightly more complex because edges have end points; $k(a)$ may contain zero, one, or two degree-zero vertices depending how many of its end points are defined by vertices in L. We will connect the vertices as previously, but in a line, rather than a loop. Since we can only add red edges, and only one at each vertex, the degree-zero vertices will necessarily be the end points of this line.

□

Theorem 2.62. *In* $\mathbf{G}$*, pushout complements of boundary embeddings exist, and give rise to partitioning spans.*

Proof. We use the boundary embedding $B \xrightarrow{l} L \xrightarrow{m} G$ to construct the complement C such that $L \xleftarrow{l} B \xrightarrow{c} C$ is a partitioning span, and show that G is indeed the pushout of this span.

Let C have vertex set $V_C = (V_G \setminus V_L) + \{\bar{\partial}\}$. We construct the edge set, and the source and target maps, in three steps:

1. Let E_C contain all the edges of the induced subgraph of G defined by the vertices V_C, and define the source and target maps on those edges correspondingly.

2. Let O_C contain $O_G \setminus m_O^{-1}(O_G)$.

3. Finally we add the edges between $\bar{\partial}$ and the rest of the graph, and simultaneously define the map $c : B \to C$. Let $\mathbf{P}$ be a solution to the re-pairing problem given by $B \xrightarrow{l} L \xrightarrow{m} G$. If in $\mathbf{P}$ there is a red edge between e_1 and e_2 we create a self-loop e at $\bar{\partial}$

and set $c(e_1) = c(e_2) = e$. If there is any vertex e in $\mathbf{P}$ which has no incident red edge, add e to E_C; if its polarity is positive set

$$s_C(e) = (s_G \circ m_E \circ l_E)(e) \qquad t_C(e) = \bar{\partial}$$

and if the polarity is negative, the source and target are reversed. We define $c_E(e) = e$.

The resulting span $L \xleftarrow{l} B \xrightarrow{c} C$ is evidently partitioning, and by construction has G as its pushout, as a consequence of Lemma 2.53. □

Theorem 2.63. *In* $\mathbf{G}$*, pushout complements of boundary embeddings are unique up to the solution of the re-pairing problem.*

Proof. Suppose that both $B \xrightarrow{c} C \xrightarrow{g} G$ and $B \xrightarrow{c'} C' \xrightarrow{g'} G$ are pushout complements for the boundary embedding $B \xrightarrow{l} L \xrightarrow{m} G$. Observe that given the boundary embedding, a solution to the re-pairing problem determines the map $c : B \to C$ and vice versa. Let us assume for now that $\mathsf{im}(c) = \mathsf{im}(c')$, i.e. they both correspond to the same pairing graph. Since m is an embedding, it follows that every part of C not in $\mathsf{im}(c)$ is preserved isomorphically in G, and similarly for C'. Since we have assumed $\mathsf{im}(c) = \mathsf{im}(c')$ this implies that $C \simeq C'$. Further, observe that different solutions of the re-pairing have the same number of edges, and hence produce the same number of self loops at $\bar{\partial}$. □

In Section 3.3 we will be interested in a structure very similar to boundary embeddings: its opposite path in a pushout square. This composite is very similar to boundary embeddings, the main difference is the order in which the boundary and dual boundary vertices are replaced.

Definition 2.64. An *opposite boundary embedding* is a map $B \xrightarrow{c} C \xrightarrow{g} G$ in $\mathbf{G}$, where B is a boundary graph, $c_V(\bar{\partial})$ is defined but $c_V(\partial)$ is undefined; and $(g_V \circ c_V)(\bar{\partial})$ is undefined. Further, C is connected, and c is an embedding.

We can calculate the pushout complement of an opposite boundary embedding by analogy with the case for boundary embeddings, and complete the pushout square when starting from the other side.

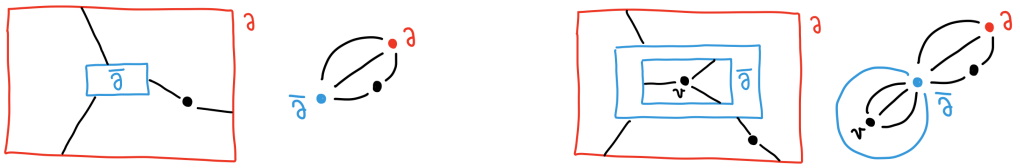
Proposition 2.65. *In* $\mathbf{G}$*, pushout complements of opposite boundary embeddings exists, give rise to partitioning spans, and are unique up to the solution of the re-pairing problem.*

Proof. Because opposite boundary embeddings are exactly symmetric to boundary embeddings (and they form the opposite side of a pushout square to boundary embeddings), this follows from Theorems 2.62 and 2.63. □

Remark 2.66. The category of graphs with circles is not adhesive. This is because the monomorphisms in $\mathbf{G}$ do not take into account partial maps which are central to our construction. Furthermore, weaker characterisations of adhesive categories do not suffice either. Quasiadhesive categories [63] ask for pushouts of regular monomorphisms only, but categories of partial morphisms does not have this property. The notion of $\mathcal{M},\mathcal{N}$-adhesive category [44, 18] provides another weaker alternative characterisation. But a similar issue arises with monomorphisms in $\mathbf{G}$ not expressing a useful class of morphisms for rewriting. For future work we are interested in using graph embeddings (as specified in Definition 2.28) as a class of *formal* monomorphisms for which we may be able to establish an adhesive property.

2.3.3 More complex boundary graphs

In our framework, boundary graphs have a fairly simple structure. They consist of two vertices and a set of edges between them. Both vertices are boundary vertices and can be replaced with a subgraph by any graph morphism. There are various ways of extending the framework (in terms of the genus of the surface or the complexity of a graph) by considering more general boundary graphs. We believe that these generalisations pose some interesting questions for future work. We sketch some initial ideas here:



(a) A hole that is contractible.

(b) A hole that is not contractible.

Figure 2.67: Examples of different kinds of boundary graph.

- By using *more than two boundary vertices* in a boundary graph we are able to represent graphs with multiple boundaries, or multiple holes. The placement of edges between boundary vertices may specify certain surface conditions, e.g. if two boundaries do not have an edge in between them, they have to be positioned in two different faces of a graph embedding. This framework could be interesting for considering the pants decomposition of a surface [47] for a graph embedding.

- A boundary graph may contain additional subgraphs, i.e. vertices that are not boundary vertices, and edges that are not placed between ∂ and $\bar{\partial}$. These graphs specify a more complex composition boundary between a graph and its context. The additional graph structure is required to be preserved by a pushout square and thus restricts the shape of graphs involved in the composition. By using a more complex boundary graph we get a more fine-grained way of specifying where in a graph a certain rewrite rule should be applied.

- We may introduce special boundary edges which can be forgotten by a graph morphism in the same style that boundary and dual boundary vertex are. They are present in the boundary graph, but not in a graph which the boundary graph embeds into. These special edges can impose particular topological properties on a graph, a context graph, or even a hole. We can imagine using a self-loop at a boundary vertex to separate two faces of a graph embedding (as no edge can cross this special edge). We reckon that this generalisation, even though a very interesting one, would require the most amount of work in terms of changing the underlying theory.

Figure 2.67b shows a (hypothetical) example of a boundary graph which specifies a graph with a non contractible hole. One region $\bar{\partial}$ of the graph is missing, but we know that this region contains a certain concrete subgraph. This boundary graph has the subgraph attached to its dual boundary vertex and a special self-loop at the dual boundary vertex. This self-loop indicates that the subgraph is completely surrounded by the region, and no edge from the outside can cross the region.

We believe there is a lot of potential in the extension of the notion of boundary graphs and the relation between multiple boundary vertices, both outside and inside.

2.4 A category of rotation systems

Up to this point we have defined graphs and their morphisms in a purely combinatorial setting. We will now add topological information to the representation of graphs. We will do so by equipping graphs with rotation systems, and thereby transforming graphs into maps. We augment our category of graphs with rotation systems in the form of *cyclic lists of flags* for each vertex, and strengthen the property of flag surjectivity (remember Equation 2.17.1). The required properties for DPO rewriting will follow more or less immediately from those of the underlying category of directed graphs.

Definition 2.68. Let $\mathsf{CList} : \mathbf{Set} \to \mathbf{Set}$ be the functor where $\mathsf{CList}\, X$ is the set of cyclic lists (cf. Definition 1.56) whose elements are drawn from X.

Definition 2.69. A *rotation system* R for a graph with circles (V, E, O, s, t) is a total function $\mathsf{inc} : V \to \mathsf{CList}(E \times \{\mathsf{src}, \mathsf{tgt}\})$ such that:

- $(e, \mathsf{src}) \in \mathsf{inc}(v) \Leftrightarrow s(e) = v$
- $(e, \mathsf{tgt}) \in \mathsf{inc}(v) \Leftrightarrow t(e) = v$

We call $\mathsf{inc}(v)$ the *rotation* at v.

Note that $\mathsf{inc}(v)$ is actually a cyclic ordering on the set of flags at v.

Morphisms of rotation systems require an additional property which states that the rotation at a vertex v is preserved by any morphism whose vertex component is defined on v. It is a strengthened version of flag surjectivity (cf. Equation refeq:flag-surj). Morphisms therefore either preserve both a vertex and its rotation, or neither of them.

Definition 2.70. A homomorphism of rotation systems $f : R \to R'$ is a $\mathbf{G}$-morphism (f_A, f_V) between the underlying graphs, satisfying the following additional condition:

$$\begin{array}{ccc} V & \xrightarrow{\quad f_V \quad} & V' \\ {\scriptstyle \mathsf{inc}}\downarrow & \geq & \downarrow{\scriptstyle \mathsf{inc}} \\ \mathsf{CList}(E \times 2) & \xrightarrow[\mathsf{CList}(f_E \times 2)]{} & \mathsf{CList}(E' \times 2) \end{array} \tag{2.70.1}$$

Definition 2.71. Let $\mathbf{R}$ be the category whose objects are tuples $(V, E, O, s, t, \mathsf{inc})$ where (V, E, O, s, t) is an object of the category of graphs $\mathbf{G}$ (see Def. 2.28) and inc is a rotation system for this graph. The morphisms of $\mathbf{R}$ are homomorphisms of rotation systems.

There is an evident forgetful functor $U' : \mathbf{R} \to \mathbf{G}$; this is especially clean since the morphisms of $\mathbf{R}$ are just $\mathbf{G}$-morphisms which satisfy an additional condition. Further, since we demand the inc structure to be preserved exactly, pushouts and complements are very easily defined from the corresponding structures in $\mathbf{G}$.

Definition 2.72. In $\mathbf{R}$, *boundary graphs*, *partitioning spans*, and *boundary embeddings* are the same structures as those of the underlying category $\mathbf{G}$. Recall Definitions 2.39, 2.41, and 2.58, respectively.

Lemma 2.73. *In* $\mathbf{R}$ *pushouts of partitioning spans exist.*

Proof. The pushout candidate is the one in the underlying category (see Theorem 2.49), together with the rotation system:

$$\mathsf{inc}_G(v) = \begin{cases} \mathsf{inc}_C(v) & \text{if } v \in V_C \\ \mathsf{inc}_L(v) & \text{if } v \in V_L \end{cases} \tag{2.1}$$

The vertex set of the pushout is the disjoint union of vertices from both input graphs, $V_G = (V_L + V_C) \setminus V_B$. Therefore, by the mediating map from Theorem 2.49, inc_G is indeed the pushout of the rotations. □

Lemma 2.74. *In* $\mathbf{R}$ *pushout complements of boundary embeddings exist, and are unique up to the solution of the re-pairing problem.*

Proof. This follows from the underlying construction in $\mathbf{G}$, recall Theorem 2.63. Note that the rotation for every vertex of C is specified by either those of G or of B, thus the additional structure is determined uniquely. □

Remark 2.75. We must sound a cautionary note about the "up to" in the preceding statement. While in $\mathbf{G}$ pushout complements that arise from different pairing graphs are essentially the same, this is not the case in $\mathbf{R}$. Since the rotation around $\bar{\partial}$ is preserved exactly by $c : B \to C$, different choices for which edges to merge as self loops will result in different local topology at $\bar{\partial}$. In particular, the re-pairing problem may have both plane and non-plane solutions; see Figure 2.60 for an example. With that caveat noted, since $\mathbf{R}$ has pushouts and their complements, specialised to the setting where the rewrite rules explicitly encode the connectivity at their boundary, we can use it as a setting for DPO rewriting of surface-embedded graphs.

Remark 2.76. As illustrated in Figure 2.60, we have adopted a particular convention for drawing a pairing graph: the vertices are placed next to each other in a row, with the red edges above and the blue edges below. If the vertices are drawn in an order compatible with $\mathsf{inc}_B(\partial)$ then the blue edges (partly) reproduce the local topology at ∂ in L. Any edge crossings imply the region around ∂ is not plane. This is sufficient but not necessary for L to be non-plane. Isomorphic statements can be made for $\bar{\partial}$ in C.

2.4.1 Closed curves

Closed curves (or "circles") are a very degenerate class of graphs, nevertheless they are valid graphs as they may occur in a string diagram containing cups, caps, and identities. We have already spent quite a considerable effort on the construction of graphs and their morphisms to

be able to accommodate closed curves in the category $\mathbf{G}$. In this section we discuss another property of circles: their behaviour in a double pushout rewriting step. As we discussed in the previous section, $\mathbf{R}$ admits DPO rewriting, but we might ask for more, for example to maintain a topological invariant. To be able to make a meaningful statement about the topology of graphs, we have to be particularly careful about the treatment of closed curves.

Rotation systems do not capture the topology of closed curves as they specify the order of edges around vertices only. Distinguishing between plane and non-plane embedded curves [94] will be necessary once we move to graphs embedded into higher genus surfaces. In the case of plane graphs we assume all *closed* loops to be embedded planarly. This justifies the representation of circles as a set in Definition 2.28 of graphs with circles. However, we have to distinguish *splittings* of closed curves into multiple segments when analysing different solutions of the re-pairing problem (compare Remark 2.75).

We have seen in Definition 2.59 that the re-pairing problem may have multiple solutions. This is the case whenever one foot of a partitioning span consists of several disconnected segments of the same edge, for example a closed curve. There are two issues arising from this which we discuss here.

Firstly, different pushout complements of the same boundary embedding can yield graphs with different topological properties. Consider the example graphs in Figure 2.77.

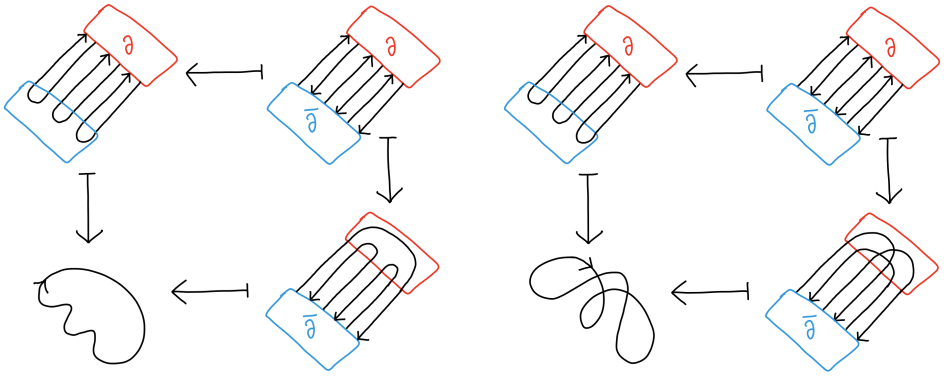


Figure 2.77: A plane and a non-plane solution of the same re-pairing problem.

In this example both pushout graphs are closed curves, and the left leg of the partitioning span is choosing three segments of the loop. For the pushout complement (in the bottom-right corner) we have to compute the possibilities of connecting these three segments to form a circle. The example on the left hand side shows a plane solution while on the right hand side the pushout complement is not a plane graph. The non-plane solution becomes an issue when we take the pushout complement as the context and insert a more complex graph into its hole

(by adding a second pushout square on the right), illustrated in Figure 2.78. Even if this more complex graph is plane itself, inserting it into a non-plane context graph results in a non-plane graph.

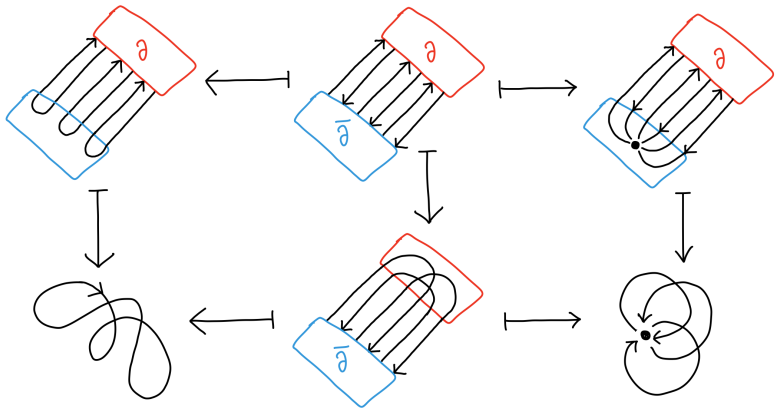


Figure 2.78: A non-plane embedding of a closed curve might lead to invalid rewrites.

The example in Figure 2.78 illustrates that having a plane rewrite rule and applying it to a plane graph (e.g. a circle) is not enough information to make any promises about the planarity of the resulting graph. To achieve preservation of planarity, we have to restrict a rewrite rule and graph to those that produce a plane context graph. We summarise this notion in the specification of a *plane rewrite step* in Definition 3.47. The re-pairing problem, together with our explicit notion of boundary vertices, provides a mechanism to detect a non-plane rewrite step early. We therefore have a tool to detect where a rewrite does not preserve the planarity of the graphs involved. In Proposition 3.48 we show that for any *plane* rewrite step the result graph of DPO rewriting is also plane.

Secondly, the re-pairing problem may return two distinct plane solutions, see Figure 2.79 for an example.

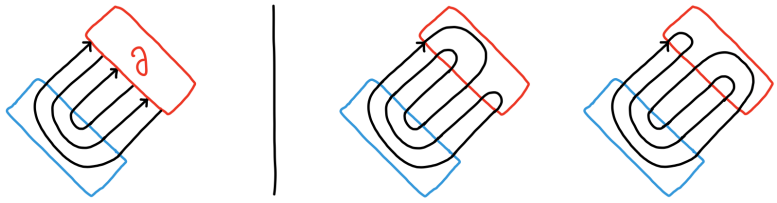


Figure 2.79: Two different, but plane, solutions of the same re-pairing problem.

This example illustrates two different ways of forming a closed loop from some of its segments (shown on the left hand side), but both results of this re-pairing problem are plane graphs. In this (very degenerate) case, the boundary embedding does not provide enough

information to calculate a unique splitting of the graph into a context and a subgraph. In this case, we will demand additional information from the user in the definition of plane rewrite step as to which context the rewrite rule should be applied in.

Summary

In this chapter we have defined a category of surface-embedded graphs. We intend graphs to include boundary vertices as auxiliary structures for any empty regions in the graph (cf. Section 2.1). These structures are necessary for adding topological information to a graph embedding in the form of rotation systems. For inserting a graph into a hole, we require a notion of substitution of a graph for a boundary vertex. Therefore, graph morphisms need to have a partial vertex component. We incorporate all of these properties in Definitions 2.28 and 2.35 of the category $\mathbf{G}$ of graphs with circles.

To be able to implement DPO rewriting for this category, we show that it has the required adhesive properties. We define partitioning spans (Definition 2.41) and boundary embeddings (Definition 2.58) as the cases in which applying a rewrite rule makes sense. We show in Theorem 2.49 that pushouts of partitioning spans exist in the category $\mathbf{G}$ of graphs with circles. Further, pushouts of boundary embeddings exist by Theorem 2.62 and are unique up to the solution of the re-pairing problem by Theorem 2.63. This is enough structure to define double-pushout rewriting for the category $\mathbf{G}$ of graphs with circles.

Finally, we define the category $\mathbf{R}$ of rotation systems (Definition 2.71) which equips graphs with circles with a cyclic ordering of edges around each vertex, thus uniquely determining a concrete surface embedding of a graph. By the careful setup of the category $\mathbf{G}$ of graphs with circles, the existence of pushouts (Lemma 2.73) and the existence and uniqueness of pushout complements (Lemma 2.74) in the category $\mathbf{R}$ follow from the corresponding structures in $\mathbf{G}$ in a straight-forward manner.

We now proceed to the special case of plane graphs as a particular instance of rotation systems in Chapter 3.

Chapter 3

Open Plane Graphs

In Chapter 2 we have specified the relevant categorical structure for graphs with rotation systems and their rewrite theory. We will now use this structure to define a particular class of surface-embedded graphs: open plane graphs. In general, rotation systems can accommodate a much larger class of graph embeddings, but for us the plane case is the most interesting one. We establish some general properties of plane graph embeddings and develop a monoidal category whose arrows are plane graphs and where composition and rewriting preserves planarity (Section 3.2). This category is a category of "pure graphs" or unlabelled string diagrams. To extend the system towards a diagrammatic presentation of concrete algebraic theories, we introduce labelled graphs in Section 3.2.3. We combine graph rewriting and graph labelling Section 3.3 where we show that substitution of subgraphs can be defined as an operad.

In the operad framework we emphasise the face structure of graph embeddings as disc-like regions on a surface. Boundary vertices act as the objects of the operad which we can interpret as graph *variables*. Substitution of a subgraph for a variable is a very natural operation in the framework as it can be implemented as operad composition.

In addition to the description of graphs as an operad, we explore the dual picture and specify a cooperad of graph *patterns*. A pattern specifies how a graph can be split into multiple different subgraphs. As both the operad and cooperad live in the same underlying category (of rotation systems), we study their interaction and propose a new definition of graph pattern matching.

Notation We write $L + K$ for the concatenation of lists L and K. We write $|L|$ to denote the number of elements of the list L. The ith element of list L is denoted $L^{(i)}$; the first element of L is $L^{(0)}$. We write $\overleftarrow{L}$ for the reverse of list L.

3.1 Plane graphs

We start by discussing some observations on graph embeddings, especially those on the sphere, while keeping in mind our aim to define the PRO of open graphs together with their rewriting theory. Rewriting is typically defined for a certain subgraph of a surface embedding. We recall the notion of disc-like region of a surface from Remark 1.44. We are interested in rewrites where the subgraph comprises a disc-like region of the surface and does not intersect any other parts of the graph. This ensures that the region can be contracted to a single vertex and therefore guarantees that the rewrite does not affect any other parts of the graph. We will go into more details about this property now.

We can characterise disc-like subgraphs using multiple iterations of edge contraction (remember Definition 1.38) on this subgraph and observing the shape of the resulting graph. We start by defining what we mean by a plane subgraph:

Definition 3.1. Given a graph G embedded into a surface S, a *plane subgraph* of the embedding is a (total) subgraph of G which is plane according to the rotation system of the embedding. (Remember from Theorem 1.61 that rotation systems uniquely determine a graph's surface embedding.)

Remark 3.2. Plane subgraphs may occur both in plane graph embeddings but also those embedded on surfaces of higher genus.

The plane subgraphs of an embedding play an important role as they are those substructure we can contract without changing the genus of the overall graph embedding. This fact is not only useful to define disc-like subgraphs of embeddings, but also to simplify a graph embedding without changing its topological properties. The analysis of those topological properties can be carried out on the smaller graph with the same results.

Proposition 3.3. *Let G be a surface-embedded graph; then:*

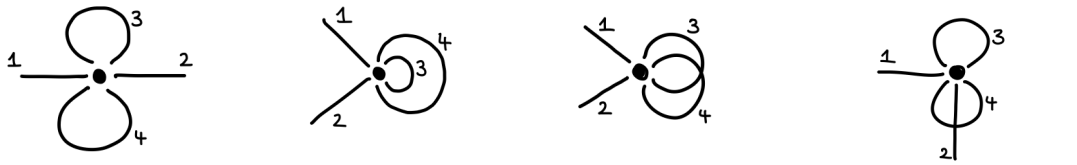
- *For any edge e, the edge contraction $G - e$ embeds in the same surface.*
- *Any connected plane subgraph of G can be edge contracted to a single vertex.*
- *The graph obtained by edge contracting a subgraph depends only on the subgraph and not on the order of contractions.*

Remark 3.4. Recall that edge contraction can only be applied to edges that have *different* source and target vertices, and not to self-loops.

As the contraction of any plane subgraph produces a single vertex, all of the subgraph's edges will form *self-loops* at this vertex. The resulting graph is called a *bouquet* graph, and any (maximally) contracted graph takes this shape. However, in the case of contracting a subgraph within a larger graph, this central vertex may still have other edges incident, connecting the subgraph with its context graph. The relation between the self-loops and these connecting edges determines if we can apply a rewrite rule to the subgraph. To make this precise, we define the notion of *locally plane vertex*.

Definition 3.5. Given a vertex v in a rotation system for G, v is called *locally plane* if its incidence list is obtained by concatenating well bracketed words over its self-loops and individual outgoing (or incoming) edges.

Remark 3.6. The words are formed from incoming and outgoing edges (each occuring once in the word), with the ends of the self-loops acting as pairs of parentheses. Note that the opening and closing parentheses are the same symbol, both orderings of the ends of the same edge can be well bracketed. Therefore, this notion is well defined for cyclic lists. In fact, for a locally plane vertex, any cyclic permutation of its incidence list (i.e. any choice as to where to break the cyclic list into a list) is a well bracketed word.



$[1,3,3,2,4,4]$ $[1,4,3,3,4,2]$ $[1,3,4,3,4,2]$ $[1,3,3,4,2,4]$

$[1],[3,3],[2],[4,4]$ $[1],[4,3,3,4],[2]$ $[1],[3,4,3,4],[2]$ $[1],[3,3],[4],[2],[4]$

Figure 3.7: Examples and counterexamples of locally plane vertices. The incidence lists are split into subwords which are well bracketed iff the vertex is locally plane.

Example 3.8. Figure 3.7 shows examples and counterexamples of locally plane vertices. In all four examples edges 1 and 2 are outgoing, and edges 3 and 4 are self-loops. The central vertex is locally plane if the subwords of its incidence lists are well bracketed. This is the case in examples 1 and 2. In examples 3 and 4, one self-loop is intersecting with another edge. In example 3, the two self loops intersect each other and in example 4 a self loop is intersecting with one outgoing edge. We can detect the intersections by observing the structure of the subwords which are not well-bracketed.

Proposition 3.9. *Given a graph embedding G and a plane subgraph H of G. H is embedded in a disc-like region of the surface if and only if contracting H to a single vertex results in that vertex being locally planar.*

Example 3.10. Figure 3.11 shows an example of a subgraph which is embedded in a disc-like surface area, and hence contracting it produces a locally plane vertex. There are two scenarios

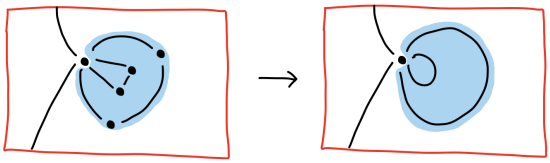


Figure 3.11: Example of contracting a disc-like subgraph (shaded region) into a locally plane vertex.

in which the contraction operation does not yield a locally plane vertex: Firstly, the order of self-loops might not be planar, such as shown in the third image in Example 3.8. Secondly, edges of a different subgraph may be enclosed by a self-loop, because they are positioned in an "inner" face of the graph embedding. This is the case in the fourth picture of Example 3.8, and Figure 3.12 illustrates a subgraph whose contraction results in such an order of edges. We

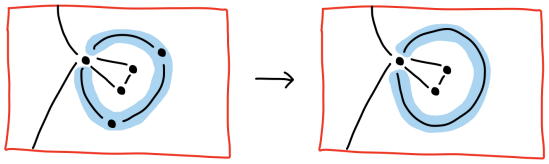


Figure 3.12: Example of contracting a non disc-like subgraph (shaded region), resulting in a non locally plane vertex.

cannot express the boundary of this subgraph by a single vertex because it is not embedded in a disc-like region of the face. Recall Section 2.3.3 on a discussion on how we may be able to represent the interface a non disc-like subgraph with multiple boundary vertices.

The following statements are true on the basis that a particular connected plane subgraph defines a disc-like region within its context graph. For each this region we can capture the edges which cross its interface and fix their cyclic ordering. Therefore, we can now not only assign rotations to individual vertices, but to whole subgraphs, too. This approach is analogous to the idea of contracting a region of a graph to a single special vertex which is the intuition for introducing boundary and dual boundaries in the first place.

Corollary 3.13. *For any connected subgraph H of G, if H can be contracted to a locally plane vertex then we can assign a unique cyclically ordered incidence list* $\mathsf{inc}(H)$ *of edges linking H*

and G/H. When contracting H to a single vertex v, the incidence list of v is equal to the one for the non-contracted subgraph, $\mathsf{inc}(H)$.

Definition 3.14. Let v be a vertex of a plane graph G, and let $r_0, \ldots, r_k$ be the regions of the plane obtained by removing the self-loops of v. The *interface* of v is the set of those r_i which contain a vertex of G. If H is a connected subgraph of G, then the interface of H is the interface of the vertex corresponding to H in the contracted graph G/H.

Proposition 3.15. *Let G be a plane graph with connected subgraph H; then to each element r of the interface of H, we can assign a unique, cyclically ordered incidence list* $\mathsf{inc}(r)$ *of edges linking H and $G \setminus H$ compatible with G.*

Proof. By the second half of Corollary 3.13 we can assume that H consists of a single vertex v. Let G_r denote the subgraph of G induced by the vertices lying in r; since r is bounded by the self-loops of v, every path from G_r to the rest of the graph passes through v. Hence we have $\mathsf{inc}(r) = \overleftarrow{\mathsf{inc}(v)|_r}$ where $\mathsf{inc}(v)|_r$ denotes the restriction of $\mathsf{inc}(v)$ to the edges of r. □

With these considerations on the planarity of embeddings and the contraction of plane subgraphs in mind, we can now move to defining open plane graphs as monoidal categories.

3.2 The PRO of open plane graphs

In this section we combine the planarity considerations of Section 3.1 with the categorical structure of surface-embedded graphs from Chapter 2 and define the PRO of open plane graphs.

We will represent open plane graphs as rotation systems which include a distinguished boundary vertex, see Figure 3.19a and define composition and substitution by the relevant constructions in the previous sections. After the construction of the category of open graphs in general, we consider the subcategory of open *plane* graphs.

Remark 3.16. Various algorithms exist which can check whether a given rotation system is plane [50, 72], so we will not explicitly do so in the following statements. Instead we will assume graphs with their rotation system to be plane and show that all operations we perform on them preserve planarity.

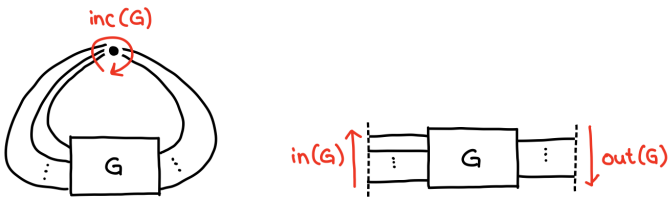
Because graphs are intended to represent morphisms of monoidal categories, we will introduce a distinction between their input and output edges. This distinction amounts to a splitting of the incidence list at the boundary vertex into two lists, one for the incoming

and one for the outgoing edges. This explicit splitting is necessary to defining composition of diagrams, both in parallel and in sequence.

Definition 3.17. An *open graph* consists of a rotation system $R \in \mathbf{R}$, a distinguished vertex $\partial G \in V(G)$, and a splitting of the rotation at ∂G into two lists of edges in and out, such that $\mathsf{inc}(\partial G) = \mathsf{in} +\!\!+ \mathsf{out}$. Additionally, $s(e) = \partial G$ for all $e \in \mathsf{in}$ and $t(e) = \partial G$ for all $e \in \mathsf{out}$. The *type* of G is $|\mathsf{in}| \to |\mathsf{out}|$.

Remark 3.18. Note that the splitting of rotation at the boundary vertex produces input and output lists of edges different orientations, see Figure 3.19b. Whilst this might be confusing when drawing graphs as actual open graphs, we prefer this representation due to the interpretation of the input and output edges as parts of a cyclic incidence list.

Even though we distinguish the inputs and outputs, open graphs are defined to contain a single boundary vertex ∂G: this vertex represents the outer face of the open graph and connects to all open edges, as shown in Figure 3.19a. This presentation suggests the interpretation of a graph embedding as a disc-like region of a surface, with the boundary storing one cyclic list of edges. Substitution of a graph into a bigger graph, or insertion of a graph into a hole of another graph are very natural to express in this framework (remember Section 2.1). We will use this particular presentation of graphs and substitution when we formulate graphs as operads in Section 3.3.



(a) one boundary (b) open graph

Figure 3.19: The boundary vertex connecting input and output edges of the graph.

Example 3.20. The following are basic examples of open graphs that we will use later. In all cases the boundary vertex is ∂. See Figure 3.21 for illustrations.

(a) Prime graph $m \to n$: $V = \{\partial, v\}$, $\mathsf{in} = [a_1, \ldots, a_m]$, $\mathsf{out} = [b_1, \ldots, b_n]$, $\mathsf{inc}(v) = \overleftarrow{\mathsf{in} +\!\!+ \mathsf{out}}$.

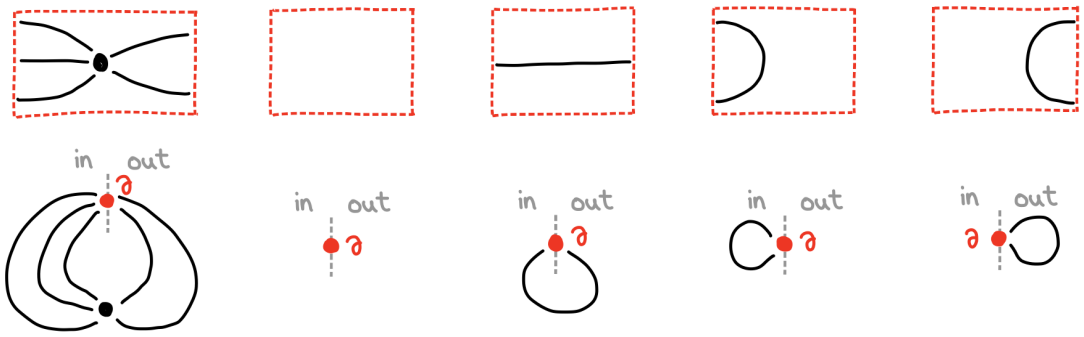
(b) Empty graph : $V = \{\partial\}$, $\mathsf{in} = []$, $\mathsf{out} = []$.

(c) Identity graph id_1 on one object: $V = \{\partial\}$, $\mathsf{in} = [e]$, $\mathsf{out} = [e]$.

(d) Cap : $V = \{\partial\}$, $\mathsf{in} = [e, e]$, $\mathsf{out} = []$.

(e) Cup : $V = \{\partial\}$, $\mathsf{in} = []$, $\mathsf{out} = [e, e]$.

Note that the "empty graph" is not actually empty but consists of a boundary vertex.



(a) Prime graph. (b) Empty graph. (c) Identity graph. (d) Cap. (e) Cup.

Figure 3.21: Illustration of the basic graphs from example 3.20.

When composing graphs in sequence, the structure of their interface edges changes: the edges at the composition boundary become "inner" edges and are not part of the interface anymore.

Definition 3.22. Given two open graphs $G : m \to n$ and $H : n \to p$ we construct their *sequential composite* (see Figure 3.24), $G \fatsemi H : m \to p$, as follows:

- vertices: $V_{G \fatsemi H} = (V_G \setminus \partial G) + (V_H \setminus \partial H) + \{\partial\}$,
- arcs: $A_{G \fatsemi H} = (A_G + A_H)/\sim$ where $\sim$ is the least equivalence relation such that $\mathsf{out}_G^{(i)} \sim \mathsf{in}_H^{(n-i)}$,
- boundary vertex: $\partial(G \fatsemi H) = \partial$,
- inputs and outputs: $\mathsf{in}_{G \fatsemi H} = \mathsf{in}_G$ and $\mathsf{out}_{G \fatsemi H} = \mathsf{out}_H$.

The incidence lists of all *vertices* (all but the boundary vertex) are inherited from the two source graphs G and H, modulo the quotient of the edges in the forming of the set of arcs.

Theorem 3.23. *Sequential composition of open graphs is associative.*

Proof. Given graphs $G : m \to n$, $H : n \to p$, and $K : p \to q$, we show that $(G \fatsemi H) \fatsemi K = G \fatsemi (H \fatsemi K)$. Because of some of the constructions not using the source graphs and also because of associativity of union of sets, the only case from Definition 3.22 requiring a more careful treatment is the construction of the new set of arcs.
We show that the resulting equivalence classes do not depend on the order of compositions. We distinguish two types of edges of H:

- Edges attached to at most one of the composition boundaries: These edges are only involved in one of the compositions and do not interfere with the other one. Therefore, the order of graph compositions does not affect them in the resulting graph.

- Edges which are connected to both composition boundaries, thus directly linking edges from G and K: Consider one of these edges $h = \mathsf{in}_H^{(n-i)} = \mathsf{out}_H^{(j)}$ which will connect edges $g = \mathsf{out}_G^{(i)}$ and $k = \mathsf{in}_K^{(p-j)}$ during composition. No matter which composition we calculate first, afterwards all three edges g, h, k will be a member of the same equivalence class: When composing graphs G and H, the edges g and h will end up in the same equivalence class as we have (g, h) in the relation. Composing with graph K afterwards, we get that k will be a member of the same equivalence class as h and therefore as g. When doing the right composition first, we get h and k being in the same equivalence class. Composition with G afterwards gives (g, h) in the relation, therefore g ends up in the same equivalence class as h and k.

□

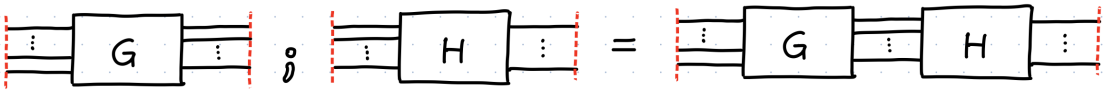


Figure 3.24: Sequential composition of two open graphs G and H.

3.2.1 Detour: extended open graphs

We can give an isomorphic characterisation of the category of open graphs, leading to a different but elegant way of defining composition and proving associativity. The idea of using a boundary vertex to represent the interface edges of a graph remains the same, except that in this version the boundary vertex is *split* into two separate vertices, an input boundary vertex ∂_{in} and an output boundary vertex ∂_{out}, connected by a boundary edge. We call this structure an *extended open graph*. Recall the schema for an open graph with one boundary vertex from Figure 3.19. The same graph as an extended open graph is shown in Figure 3.25. Topologically, these graphs embed in the same surface with the operation of vertex splitting and edge contraction (cf. Definition 1.38) translating between the two presentations. We can also show that they are isomorphic as PROs. For this extended definition of graphs with distinguished vertices for input and output edges, we propose an alternative way to constructing their composition *abstractly*,

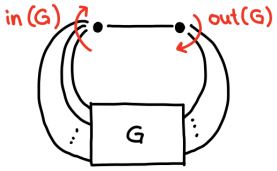


Figure 3.25: Schema of an extended open graph with two boundary vertices, capturing input and output edges, respectively.

by using a specific pushout in the underlying category of rotation systems. This construction demonstrates the fact that composition is a special case of a rewrite operation:

Proposition 3.26. *Sequential composition $G \fatsemi H$ of extended open graphs corresponds to the pushout of the span $G \leftarrow \partial\bar{\partial}_n \rightarrow H$ in the category of graphs with circles. $\partial\bar{\partial}_n$ defines the graph consisting of ∂G_{out} and ∂H_{in}, connected by $n+1$ edges: n edges for the composition boundary together with the special boundary edge. See Figure 3.27 for an illustration of the construction.*

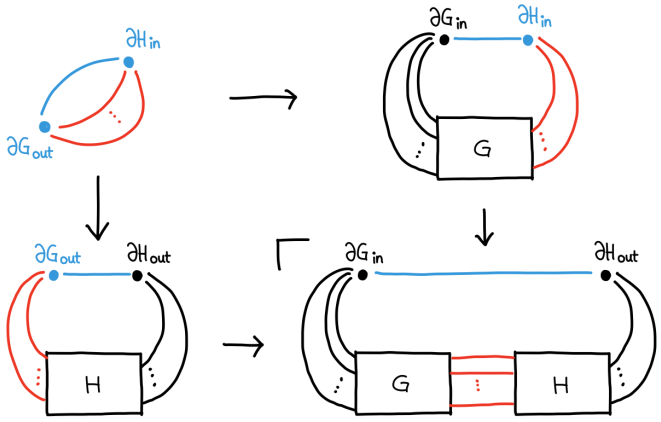


Figure 3.27: Composition of open graphs by pushout.

The fact that $G \leftarrow \partial\bar{\partial}_n \rightarrow H$ is a partitioning span supports the fact that we compose two separate graphs. One leg of the span introduces the left hand side graph G and the other one H while always ensuring that the edges between them as well as the special boundary edge are preserved. This definition of sequential composition is not just an elegant alternative to the direct one, but it also allows for a very short proof of associativity of composition, because we can use the corresponding property of pushouts:

Theorem 3.28. *Sequential composition of extended open graphs is associative up to isomorphism.*

Proof. We use Proposition 3.26 which constructs sequential composition as a pushout of rotation systems. We immediately get associativity by the fact that pushouts are associative. □

Now back to (normal) open graphs.

3.2.2 Monoidal structure of open plane graphs

In this section we construct the parallel composition of two open graphs. In contrast to the sequential composition where some of the interface edges are merged to become inner edges, the interface of a tensor product of open graphs is the concatenation of their individual input and output lists of edges.

Definition 3.29. Given two open graphs $G : m_1 \to n_1$ and $H : m_2 \to n_2$ we construct their *tensor product* (or *parallel composite*), see Figure 3.36, $G \otimes H : m_1 + m_2 \to n_1 + n_2$ as follows:

- vertices: $V_{G\otimes H} = (V_G \setminus \partial G) + (V_H \setminus \partial H) + \{\partial\}$,
- arcs: $A_{G\otimes H} = A_G + A_H$,
- boundary vertex: $\partial(G \otimes H) = \partial$,
- inputs and outputs: $\mathsf{in}_{G\otimes H} = \mathsf{in}_H +\!\!+ \mathsf{in}_G$ and $\mathsf{out}_{G\otimes H} = \mathsf{out}_G +\!\!+ \mathsf{out}_H$.

The incidence lists of all inner vertices are inherited from the individual graphs G and H.

Theorem 3.30. *The tensor product of open graphs is associative.*

Proof. Union of sets (+) and list concatenation (+$\!\!+$) are associative. □

Remark 3.31. We can construct the definition of parallel composition graphically by drawing an edge between the boundary vertices of the two graphs (exactly at the split between input and output lists), and then contracting this edge. The intuition for this operation is the same as for the definition of extended open graphs in Section 3.2.1. We will use the edge contraction operation later to show that parallel composition preserves planarity of graph embeddings in Theorem 3.39. See Figure 3.42 for a preview of the construction.

Theorem 3.32. *The empty graph is the unit of the tensor product of open graphs.*

Proof. The empty graph consists of a boundary vertex only, but for the construction of the composite we only need to consider the empty set of edges and empty lists representing no inputs or outputs. We then use that the empty set and the empty list are the unit of set union and list concatenation, respectively. □

Remark 3.33. Note that while we were able to express sequential composition in an elegant alternative way by a pushout in the underlying graph category, we cannot do this construction to model parallel composition. This is due to the definition of morphisms in the category of

open graphs: we demand graph morphisms to be flag-surjective, meaning that the number of edges attached to vertices must not increase. In contrast, for calculating the tensor product of open graphs, the lists of edges at the boundary vertices are the concatenation of the lists of the two original graphs.

Definition 3.34. Let id_0 denote the empty graph, as in Example 3.20; then for all $n \geq 1$ define $\mathrm{id}_n = \mathrm{id}_1 \otimes \mathrm{id}_{n-1}$.

Theorem 3.35. id_n *is the unit of sequential composition of open graphs.*

Proof. As the identity graph on n does not contain any vertices (except the boundary vertex), we only look at the set of edges of the composite. For calculating the set of edges of the composite, we observe that $\mathsf{in}_{\mathrm{id}_n}^{(n-i)} = \mathsf{out}_{\mathrm{id}_n}^{(i)}$. Therefore we have $E_{G \fatsemi \mathrm{id}_n} = ((E_G + E_{\mathrm{id}_n})/\sim) \simeq E_G$. The same argument holds for composing with the identity graph on the left. □

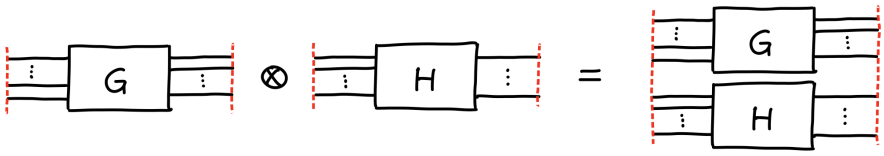


Figure 3.36: Parallel composition of two open graphs G and H.

Theorem 3.37. *The collection of all open graphs forms a PRO, denoted* **OG**, *with composition per Definition 3.22, tensor product per Definition 3.29 and identities per Definition 3.34.*

Proof. The laws hold per Theorems 3.23, 3.35, 3.30, and 3.32. □

Remark 3.38. In fact, due the presence of cups and caps (cf. Example 3.20), **OG** is *pivotal*: it is a rigid category (aka autonomous) with each object being its own left and right dual [87].

The PRO **OG** contains all open graphs and hence all surface-embedded graphs, not just the plane graphs. To capture only the plane graphs we require both horizontal and vertical composition to preserve planarity. We will give algorithms to calculate both compositions of graphs and then show that each step in the algorithm preserves planarity.

Theorem 3.39. *If G and H are open plane graphs then $G \otimes H$ is plane too; further if $G \fatsemi H$ is defined, then $G \fatsemi H$ is also plane.*

Proof. We provide constructions of the required graphs, where each step preserves planarity.

Parallel composition We provide an algorithm for computing the parallel composition of two open plane graphs. Assume $G : m \to n$ and $H : k \to l$.

1. Observe that H contains a unique face incident at ∂H, specified by the edges $\mathsf{in}_H^{(0)}$ and $\mathsf{out}_H^{(n)}$. Embed G in this face.

2. Add a temporary edge e between the two boundary vertices ∂G and ∂H. e occurs in the rotation of ∂G between $\mathsf{out}_G^{(n)}$ and $\mathsf{in}_G^{(0)}$ and in the rotation of ∂H between $\mathsf{in}_H^{(k)}$ and $\mathsf{out}_H^{(0)}$. Figure 3.42 illustrates this operation.

3. Contract the temporary edge e. This operation merges the two boundary vertices and concatenates their input and output lists. Set the resulting boundary vertex to ∂.

Lemma 3.40. *This algorithm produces a graph with the specifications in Definition 3.29.*

We show that each step involved preserved planarity:

1. Placing a plane graph into the face of another plane graphs produces a plane graph.

2. The insertion of the edge does not produce any crossings, because the two graphs G and H do not share any other edge. Therefore, the result is still a plane graph.

3. Contracting an edge does not change its genus, see Proposition 3.3. Therefore, the result is a plane graph.

Remark 3.41. We can observe the same result using the Euler Formula (cf. Theorem 1.53): For both graphs G and H the Euler formula is satisfied. When composing the two graphs, their boundary vertices are merged into one, therefore the number of vertices decreases by 1. At the same time, we identify the outside face of H and the inside face of G, thus the number of faces of the also decreases by 1. This means that the Euler Formula of the resulting graph is satisfied.

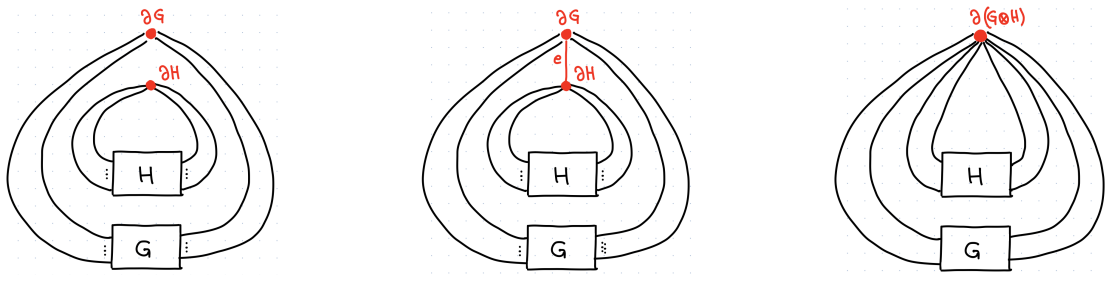


Figure 3.42: The tensor product of two open plane graphs is an open plane graph.

Sequential composition Assume $G : k \to m$ and $H : m \to n$ We compute their sequential composition $G \fatsemi H$ as follows (see Figure 3.43 for an illustration):

1. Placing the graphs G and H next to each other, insert a temporary edge e between the boundary vertices ∂G and ∂H. This edge is inserted into the rotation of ∂G between $\mathsf{in}_G^{(k)}$ and $\mathsf{out}_G^{(0)}$ and into the rotation of ∂H between $\mathsf{in}_H^{(m)}$ and $\mathsf{out}_H^{(0)}$.

2. Insert two temporary vertices v_{out_G} and v_{in_H} to disconnect the outputs of G and inputs of H from the respective boundary vertices. The vertex v_{out_G} has rotation $\overleftarrow{\mathsf{out}_G}$ and vertex v_{in_H} has rotation $\overleftarrow{\mathsf{in}_H}$.

3. Insert and contract an edge between the two temporary vertices. The resulting vertex has rotation $\overleftarrow{\mathsf{out}_G} +\!\!+ \overleftarrow{\mathsf{in}_H}$.

4. Letting i range over 0 to $m-1$, we identify edges $\mathsf{in}_H^{(i)}$ and $\mathsf{out}_G^{(n-i)}$ one by one, removing the temporary vertex at the end. This is well defined as the number of edges at the composition boundary matches.

5. We now contract the edge linking ∂G and ∂H and set the resulting vertex to ∂.

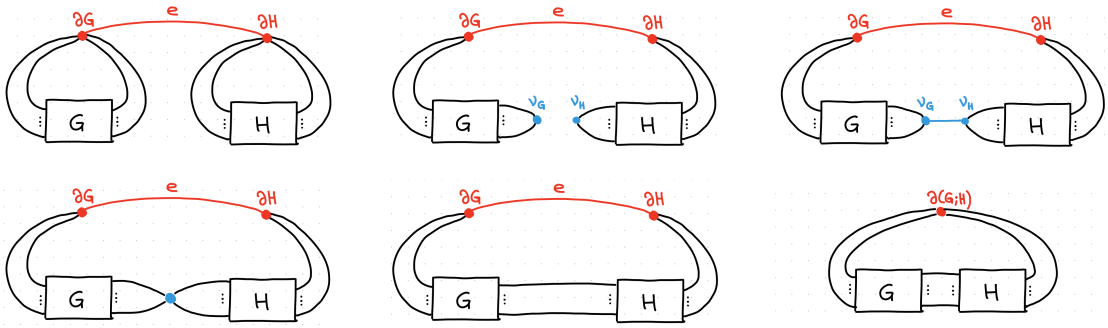


Figure 3.43: The sequential composition of two open plane graphs is an open plane graph.

Lemma 3.44. *This algorithm produces a graph with the specifications in Definition 3.22.*

The resulting graph is open plane as each step preserves this property:

1. Inserting an edge does not introduce any crossings as the graphs G and H do not share any other edge.

2. By inserting the temporary vertices we ensure that the order of output and input edges stay unchanged. Therefore, no edge crossings are introduced.

3 This operation ensures that the relation between the outputs of G and inputs of H stay unchanged. It produces a plane graph as there are no edges going in between vertices of G and H except the edge between the two boundary vertices. As the new edge does not intersect with the boundary edge, planarity is preserved.

4 When identifying edges we ensure that their order stays the same and therefore we do not introduce any edge crossings.

5 Contracting an edge does not change its genus, see Proposition 3.3, and because the graph is plane beforehand it is also plane after the operation.

Remark 3.45. We can observe the result on the Euler Formula, too. The number of vertices in the composition is the sum of both individual graphs minus 1 (as we merge the boundary vertices). Similarly, we merge the two inner faces of the graphs, thus the number of faces decreases by 1, too. As we identify edges at the composition boundary, we split faces of the embedding into two by adding an edge each. These two elements cancel each other out in the sum and thus the result still satisfies the Euler Formula.

□

By virtue of Theorem 3.39, the open plane graphs form a sub-PRO of $\mathbf{OG}$, which we denote $\mathbf{Plane}$. $\mathbf{Plane}$ admits an alternative characterisation.

Theorem 3.46. $\mathbf{Plane}$ *is generated by the open graphs of Examples 3.20 under composition and tensor product.*

Proof. Any plane-embedded graph is homeomorphic to one built from straight left-to-right segments, cups and caps. From here is is possible to tile the plane so that each tile contains exactly one of the generators [27]. □

Finally, having defined a category of open plane graphs, we now specify their rewrite theory and prove that a plane rewrite step inside an open plane graph results again in an open plane graph. We start by defining a plane rewrite step which takes into account closed curves in a graph, as discussed in Section 2.4.1.

Definition 3.47. A *plane rewrite step* of rotation systems takes as inputs a rewrite rule, presented as a partitioning span $L \leftarrow B \rightarrow R$, and a match m of the left hand side in a graph G, presented as boundary embedding $m : B \rightarrow L \rightarrow G$. It consists of two stages:

1. Calculate the pushout complement of the match m. If there is no plane solution of the re-pairing problem, deny the rewrite. If there are multiple plane solutions of the re-pairing problem, request the user to pick one.

2. Compute the pushout of the partitioning span $C \leftarrow B \rightarrow R$.

If the rewrite is successful (which is the case if the re-pairing problem has a solution), the result of a plane rewrite step is the graph $G[L/R]$.

We show that applying a plane rewrite step to an open plane graph preserves planarity:

Proposition 3.48. *Let L, B, R and G be open plane graphs, i.e. objects in the category $\mathbf{R}$ with a plane rotation system. Then rewriting L for R in G by a successful plane rewrite step (as defined in 3.47) preserves planarity, i.e. $G[R/L]$ is also an open plane graph.*

Proof. For any closed curves involved, planarity is ensured by the specification of the plane rewrite step (cf. Definition 3.47). To show the property for other shapes of graph, we will show that each map in the double pushout squares preserves planarity. We observe that each of the morphisms in a DPO diagram replaces one vertex (either the boundary or the dual boundary vertex) with a subgraph, while leaving the rest of the graph unchanged. Therefore, we prove that this replacement step preserves planarity. Take $l : B \rightarrow L$ which replaces $\bar{\partial} \in B$ by a subgraph. Because ∂ and $\bar{\partial}$ do not have self-loops in B, we know that all edges $e \in B$ we have that exactly one end (the one attached to $\bar{\partial}$) which gets replaced while the other one stays attached to ∂. In particular, the order of the edges $E_{\partial\bar{\partial}}$ around the new subgraph is fixed by the rotation around ∂, and by Equation 2.70.1 has to be preserved by l. This preserved order ensures the order in which the boundary edges E_B are mapped to the outside edges of L. Because the rotation at ∂ is part of a plane embedding, no edge crossings between ∂ and $L \setminus \partial$ can be introduced by the morphism, and because L is an open plane graph, the overall result is plane, too. □

3.2.3 Labelled graphs

One motivation for constructing the category $\mathbf{Plane}$ is to use string diagrams to reason about concrete algebraic theories in a non-symmetric setting. For this purpose, the category $\mathbf{Plane}$ is too general: we must restrict the language to the particular theory of interest. To achieve this we introduce the standard notion of labelling for a graph.

Definition 3.49. A *monoidal signature* consists of a set of morphism symbols Σ, and a pair of functions $\mathrm{dom}, \mathrm{cod} : \Sigma \to \mathbb{N}$, assigning to each symbol its input and output arity. We write $\sigma : m \to n$ for an element $\sigma \in \Sigma$ with $\mathrm{dom}(\sigma) = m$ and $\mathrm{cod}(\sigma) = n$.

Definition 3.50. Let Σ be a monoidal signature as above; a Σ-*labelled open graph* is an open graph G augmented with a function $\mathsf{lab} : V \setminus \partial G \to \Sigma$ such that $\mathsf{lab}(v) = g$ if and only $g : m \to n$ and $\deg(v) = n + m$.

Observe that the Σ-labelled open graphs form a PRO, denoted $\mathbf{OG}_\Sigma$ by the same construction as $\mathbf{OG}$ itself. Further, each generator $g : m \to n$ corresponds to a prime graph (cf. Examples 3.20) with a different labelling, which allows the signature Σ to be embedded in a set of Σ-labelled open *plane* graphs.

Proposition 3.51. *Let* $\mathbf{Plane}_\Sigma$ *denote the subcategory of* $\mathbf{OG}_\Sigma$ *whose underlying open graphs are plane; then* $\mathbf{Plane}_\Sigma$ *is the free PRO generated by* Σ.

3.3 The operad of open plane graphs

The definition of open graphs with boundary as a monoidal category highlights the operation of composing them with each other by placing them side by side, either in sequence or in parallel. Successive application of the two side-by-side compositions produces a graph that is a *tiling* of its subgraphs, see Theorem 3.46. We will now explore an alternative framework for defining open graphs and their composition: operads [65, 69]. Operads emphasise a different structural relation between a graph and its subgraphs. In contrast to a *tiling* approach for building bigger graphs from smaller ones, operad composition is implemented by *insertion* of one graph into another one, by substituting one of the outer graph's vertices by the inner graph. Because rewriting of subgraphs is a key feature in our graph framework, we already have the necessary machinery for characterising them as operads at hand. Our constructions for removing and inserting subgraphs and presentations of the outside and inside interfaces of a graph will fall into place nicely in the operad framework, as we will explain below.

In general, operads capture the idea of building structures from multiple substructures at once. They generalise categories: morphisms of operads take multiple arguments, they describe mappings from many disjoint objects to one. Because of their self-similar structure, operads are of interest both for theoretical models where the elements of a specific type can be built inductively from smaller elements of the same type, and physical applications, e.g. the

successive construction of circuits which works by soldering together smaller circuits. In the context of diagrammatic languages operads are convenient structures to talk about the layout of a diagram and precisely specify the relation between subdiagrams.

Using an operad structure to subdivide space on a surface is not a new concept: Tom Leinster presents the Little Discs Operad [65] as a specification of the layout of a finite number of disjoint closed discs inside a larger disc. The objects of this operad are the discs, and an n-ary morphism defines the arrangement of n small discs inside a larger disc. We have already seen that certain regions of a graph embedded on a surface are homeomorphic to a disc. Gluing together the discs creates the graph embedding and the surface. This intuition matches the information contained in the little discs operad.

The operad of wiring diagrams Another operadic structure that is very related to our construction is David Spivak's Wiring Diagrams Operad [90]. This operad defines wiring diagrams consisting of nodes and edges between them, with composition being the substitution of a node with a subdiagram. The structure we define here is very related to the operad of wiring diagrams, but in addition to the connectivity information between vertices, we also specify the topological arrangement of the subgraphs.

Definition 3.52. A small *coloured operad* $\mathcal{C}$ consists of:

- a set of *objects* (or *colours*) $\mathcal{C}_0$;
- for each $n \in \mathbb{N}$, and set $a_1, \ldots, a_n, a \in \mathcal{C}_0$ of objects, a set of *arrows* $\mathcal{C}(a_1, \ldots, a_n; a)$;
- for each a in $\mathcal{C}_0$ an *identity* arrow $\mathrm{id}_a \in \mathcal{C}(a; a)$
- for each $n, k_1, \ldots, k_n \in \mathbb{N}$, and each $a, a_i, a_i^j \in \mathcal{C}_0$ a *composition* function

$$\begin{aligned}\mathcal{C}(a_1, \ldots, a_n; a) \times \mathcal{C}(a_1^1, \ldots, a_1^{k_1}; a_1) \times \cdots \times \mathcal{C}(a_n^1, \ldots, a_n^{k_n}; a_n) \\ \to \mathcal{C}(a_1^1, \ldots, a_1^{k_1}, \ldots, a_n^1, \ldots, a_n^{k_n}; a)\end{aligned}$$

satisfying unit, associativity (and commutativity) laws that we will not reproduce here but refer to the literature [65] instead.

By default, composition of operads is defined for all inputs $a_1, \ldots, a_n$ at once, but we can express composition at one input at a time by using the identity map on all other inputs. This *indexed* composition is drawn schematically in Figure 3.54. For simplicity, we will use indexed

composition as the default version. We do not loose generality by doing so, especially as the inputs to the operad are disjoint elements.

Remark 3.53. When we use the word *operad* we mean a coloured operad. In the literature this is also sometimes called a multicategory.

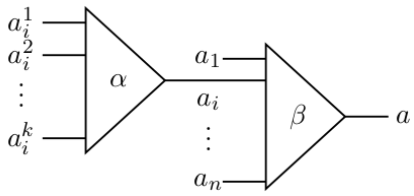


Figure 3.54: Schema of operad composition $\mathcal{C}(a_1, \ldots, a_n; a) \circ_i \mathcal{C}(a_i^1, \ldots, a_i^k; a_i)$.

3.3.1 The operad of surface-embedded graphs and substitution

We define an operad whose operations are open graphs and its objects are rotations. A graph may contain multiple holes (i.e. dual boundary vertices) which serve as the inputs of the operad map. The output is the rotation at the graph's boundary vertex. Composition of maps is given by the substitution of one graph G into the hole of another graph H, defined as an instance of DPO rewriting in the underlying category along the interface of G and the corresponding hole in H.

We will construct an operad whose arrows are open graphs, and whose objects are their types. Since **OG** is already a PRO, the operad structure effectively adds a third dimension of composition, in addition to the sequential and parallel versions defined in Section 3.2.

Remark 3.55. The operad structure emphasises the disc-like nature of the faces of a graph embedding. Therefore we use rotations at vertices as cyclic lists without a splitting into inputs and outputs.

We use a labelling per Definition 3.50. Let $\mathcal{V}$ be a monoidal signature containing for each $n \in \mathbb{N}$ a countable supply of *variables* $\bar{\partial}A, \bar{\partial}B, \bar{\partial}C, \ldots$ with rotation n, e.g. for a variable $\bar{\partial}X \in \mathcal{V}$ we have $\deg(\bar{\partial}X) = n$.

Proposition 3.56. *The PRO* $\mathbf{OG}_{\mathcal{V}}$ *forms a coloured operad with the following structure:*

- *The objects are rotations* $\mathsf{CList}(\mathbb{N})$.
- *A* k*-ary map is an open plane graph* G *with* k *holes, i.e. a morphisms of* $\mathbf{OG}_{\mathcal{V}}$*: the* k *inputs are given by distinct labels of dual boundary vertices* $\bar{\partial}X_i$ *occurring in the graph, the output*

is the boundary vertex of the graph. We write morphisms in sequent style:

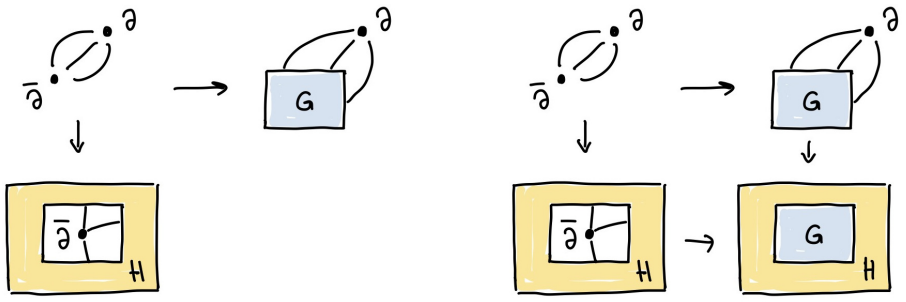
$$G : \bar{\partial} X_1, \ldots, \bar{\partial} X_k \vdash \partial G.$$

- *Composition at the ith input, $H \circ_i G$, of graphs*

$$G : \bar{\partial} X_1, \ldots, \bar{\partial} X_k \vdash \partial G \text{ and } H : \bar{\partial} X'_1, \ldots, \bar{\partial} X'_i, \ldots, \bar{\partial} X'_n \vdash \partial H$$

is well defined if the rotations at the composition boundary match: $\mathsf{inc}(\bar{\partial} X'_i) = \overleftarrow{\mathsf{inc}(\partial G)}$. *It is then defined by substitution of G for the dual boundary vertex $\bar{\partial} X'_i$ in H. This is calculated as the pushout of the partitioning span $G \leftarrow \partial\bar{\partial}_G \rightarrow H$ in the category of rotation systems, along the boundary graph $\partial\bar{\partial}_G$. The result is a graph of the form:*

$$H \circ_i G : \bar{\partial} X'_1, \ldots, \bar{\partial} X_1, \ldots, \bar{\partial} X_k, \ldots, \bar{\partial} X'_n \vdash \partial H.$$



(a) Given two graphs, (b) compose them by pushout.

Figure 3.57: Example schema of calculating graph composition by pushout.

Remark 3.59. Recall that the edges in these graphs are untyped, they only carry connectivity information. One can imagine a more general construction with wires carrying more interesting types which would be an interesting case to consider for future work.

Remark 3.60. For simplicity we discuss examples of operads with one input variable only. Graphs of this shape have at most one hole and at most one outside face. Operads do provide a suitable framework for more general structures that have multiple holes (but one outside face).

Figure 3.57 illustrates schematically composition of graphs by pushout, and Figures 3.58 shows a concrete example of this process.

Corollary 3.61. *Since by Proposition 3.48 rewriting preserves planarity,* $\mathbf{Plane}_{\mathcal{V}}$ *is the operad*

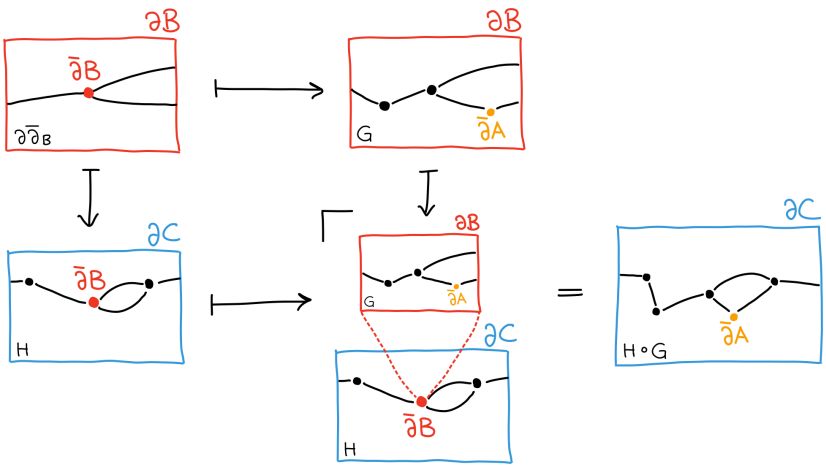


Figure 3.58: A concrete example of operad composition of $G : \bar{\partial}A \vdash \partial B$ and $H : \bar{\partial}B \vdash \partial C$ along $\partial\bar{\partial}_B$.

of plane graphs with inherited operad structure of $\mathbf{OG}_{\mathcal{V}}$*. The same holds for* $\mathbf{Plane}_{\Sigma+\mathcal{V}}$ *for any monoidal signature* Σ*.*

3.3.2 The cooperad of graph patterns and substitution

With the operad of plane graphs we have developed a structure that constructs graphs by wiring together subgraphs. In this section we will explore the dual of building graphs: taking them apart. An operad morphism expects multiple subgraphs as its inputs and produces a larger graph containing these subgraphs. The structure we define now takes a large graph as its argument and calculates how to *split* it into multiple subgraphs. We suggest interpreting this operation as the un-wiring of a diagram, arising from taking the dual of a wiring diagram. We call the elements of this dual construction *graph patterns* and organise them in a *co*operad [38].

Proposition 3.62. *The PRO* $\mathbf{OG}_{\mathcal{V}}$ *forms a coloured cooperad with the following structure:*

- *The objects are rotations* $\mathsf{CList}(\mathbb{N})$.
- *A* k*-ary map is an open plane graph* P *with* k *holes, called a* pattern*: the input is the boundary vertex of the graph and the* k *outputs are distinct labels of dual boundary vertices. We write maps in sequent style:*

$$P : \partial G \dashv \bar{\partial}X_1, \ldots, \bar{\partial}X_k.$$

- *Composition* $P \circ_i Q$ *is defined as the pushout in the category of rotation systems of the span* $P \leftarrow \partial\bar{\partial}_P \rightarrow Q$*, analogous to operad composition in Proposition 3.56.*

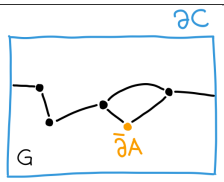
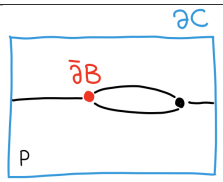
Graph	Pattern
$\bar{\partial} A \vdash \partial C$	$\partial C \dashv \bar{\partial} B$
	

Table 3.1: Comparison of graphs and patterns as operad and cooperad.

Patterns contain the same information as graphs, but the *flow* of this information is different. Graphs combine multiple elements $\bar{\partial}_i$ into one larger structure of type ∂G, whereas patterns take an overall structure of type ∂P and return multiple substructures $\bar{\partial}_i$. Thus, patterns can extract certain subregions from a larger graph.

Graphs versus patterns The operad of open plane graphs defines a structure to define how to build bigger graphs from smaller ones, it provides a way of *constructing* graphs by wiring together its subgraphs. Cooperads on the other hand tell us in which way we can take a graph apart by extracting its subgraphs. Patterns specify the un-wiring of a diagram, therefore they are the eliminator instead of the constructor of a diagram. To highlight the dual nature of graphs and patterns and to clarify drawing conventions, Table 3.1 shows examples of a graph and a pattern.

We want to use the language of graph patterns to guide the extraction of certain region of a given graph. The pattern specifies where this particular region is to be found inside a graph by providing the context graph around the *dual boundary*. Therefore, the output type of a pattern is the representative dual boundary vertex of the subregion.

So far, patterns can be composed with each other to further specify the location of their variables, by cooperad composition. As its variables are never instantiated, patterns form an *expression* language in which they do not reduce. Patterns can only act (and thus reduce) when we match a graph against them. As both graph and patterns are encoded as morphisms in $\mathbf{OG}_{\mathcal{V}}$, we can study this interaction.

3.3.3 Operad-cooperad interaction

Because the elements of operad and cooperad live in the same underlying category of rotation systems, we can compose morphisms of either structure with *each other*. The boundary (and

dual boundary) vertex structure of graphs and patterns guide which compositions between graphs and patterns are possible. We will focus on precomposing a graph to a pattern. We define how to pass a value (the graph) to an expression (the pattern) and then *match* the value against the pattern, a well known construct from programming. We start by specifying what we mean by a *match* and then how we can evaluate it.

Remark 3.63. For simplicity of the explanation we focus on graphs and patterns that have at most one variable. For example, a pattern might look like $P : \partial C \dashv \bar{\partial} B$. Because of the operad-cooperad structure, the framework can be extended straight-forwardly. We believe that the more general example includes interesting applications with graph programming.

Definition 3.64. Given a graph $G : \bar{\partial} A \vdash \partial C$, and a pattern $P : \partial C \dashv \bar{\partial} B$, a *match* is given by a morphism in the category of rotation systems $\mathbf{R}$, $m : P \to G$, such that $\partial\bar{\partial}_B \longrightarrow P \xrightarrow{m} G$ is an opposite boundary embedding. A match can *fail*, if there exists no such map m.

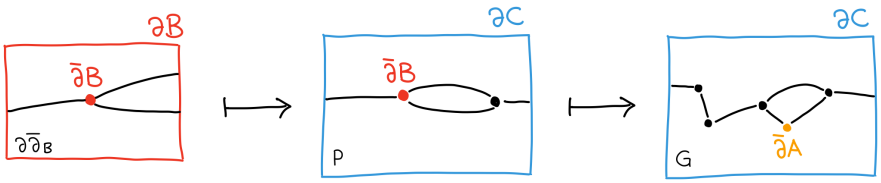


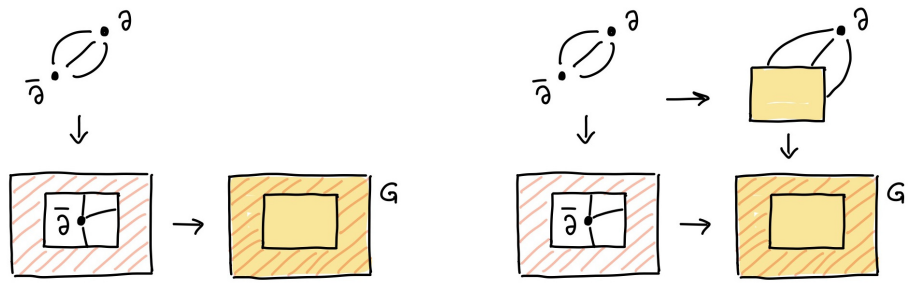
Figure 3.65: Example of a match of a graph $G : \bar{\partial} A \vdash \partial C$ against a pattern $P : \partial C \dashv \bar{\partial} B$, and the resulting opposite boundary embedding $\partial\bar{\partial}_B \to P \to G$.

Figure 3.67a depicts the schema of a match and Figure 3.65 shows a concrete example.

Mapping a pattern onto a graph highlights the parts which both structures share. In addition to these *shared* parts, a graph may contain other elements which are going to be exposed by a "pattern matching" operation. We can show that there exists at most one match for any graph and pattern:

Lemma 3.66. *If a match $m : P \to G$ exists, it is unique for a pattern P and a graph G.*

Proof. We assume a match $m : P \to G$. The pattern $P : \partial G \dashv \bar{\partial} B$ is a plane graph containing vertices ∂G as its boundary and $\bar{\partial} B$. Similarly, the graph $G : \bar{\partial} A \vdash \partial G$ is a plane graph which contains vertices $\bar{\partial} A$ and ∂G. First, we observe that because m is part of an opposite boundary embedding (cf. Definition 2.64), it is defined on all vertices $v \in V(P)$ except the dual boundary vertex $\bar{\partial} B$. Second, by Equation 2.70.1, m has to preserve the rotations of all the vertices it is defined on. Therefore, a match only exists, if all rotations in P and G coincide. Any other match $m' : P \to G$ has to satisfy these two properties, and m' has the same vertex map as m by Equation 2.70.1. Therefore m' describes the same map as m. □



(a) Given a match $\partial\bar{\partial} \to P \to G$, (b) calculate matching G against P.

Figure 3.67: Example schema of calculating a pattern match by taking the pushout complement of a match.

As a match has the structure of an opposite boundary embedding, we can calculate its pushout complement. This operation is precisely the calculation of the pattern match.

Definition 3.68. Given a graph G, a pattern P, and a match $m : P \to G$. The pushout complement of the opposite boundary embedding $\partial\bar{\partial}_G \to P \to G$ is performing the *pattern match* of G against P which we write $G \bowtie P$.

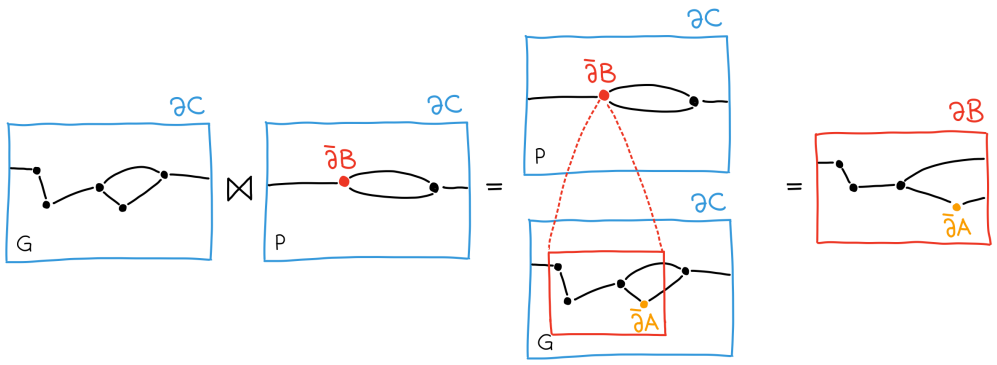


Figure 3.69: A concrete example of a pattern matching operation: Given the match from Figure 3.65, the result of the pattern match is a graph $A \vdash B$.

The pattern match returns the subgraphs of G that instantiate the pattern's variables, thus replacing the pattern's dual boundary vertices in the match m.

Lemma 3.70. *Given a match $m : P \to G$, the calculation of the result of the pattern matching operation always exists (and is unique up to circles).*

Proof. This follows from the fact that m is a morphism in the category of rotation systems $\mathbf{R}$, and from the fact that $\mathbf{R}$ has pushout complements that are unique up to the forming of circles, see Proposition 2.73. □

Figure 3.67b shows the schema of the pattern matching operation by completing the pushout square from the given match in Figure 3.67a. A concrete example of a pattern match is illustrated in Figure 3.69.

From the types in the pushout square we can read off that the result of the composition of graph and pattern is a *graph*. This observations fits into the interpretation of patterns as an expression language. Calculating the match amounts to apply a pattern (i.e. an expression) to a graph (i.e. a value) which results in a graph (i.e. a value).

Summary

In this chapter we have defined the special case of plane graphs as an instance of surface-embedded graphs. We have shown that composition and tensor product as well as substitution for these graphs preserves their planarity. With this structure we have created a framework in which we can reason about plane graphs and the rewriting theory. This fulfills our initial goal: a combinatorial theory for encoding string diagrams of monoidal categories that are non-symmetric.

Furthermore, with the specification of plane graphs as operads we suggest a framework where substitution of subgraphs is a first class operation. Together with graph patterns and the matching operation between the two we have presented a new framework for programming with graphs, with operations and term manipulation similar to those in functional programming and metaprogramming. Our framework has applications where graph matching is a key operation, for example in query languages for database management [36]. As future work, we plan to extend our framework with notions from functional programming, like the handling of overlapping patterns as well as an explicit distinction between a language of expressions and a (smaller one) of patterns.

3.3.4 Related and future work

Plane graphs are not only interesting in the context of graphical languages for certain monoidal categories. They have an interesting connecting with logic: rooted plane graphs correspond to plane lambda terms in which the order of abstraction and application must be planar [97, 96]. We would like to establish this connection with our framework in the future.

Additionally, we want to compare our work on boundary vertices with other notions graph contexts [86, 33].

There are various interesting questions to be explored based on the operad and cooperad notion of graphs:

Firstly, the formulation of graphs as operads and their patterns as the dual concept motivates to think about graphs from a programmer's point of view. With graphs representing the graphical calculus for various mathematical theories, programming with them as a way of reasoning in the theory is an active research area. Especially in the context of data base systems, graph pattern matching is a common technique [36].

Secondly, so far we have only considered one-hole structures in our definitions. Considering graphs and patterns with multiple variables would be an interesting extension of the framework.

Additionally, a potential operation on graphs and patterns is their composition with exchanged positions. So far we have precomposed a pattern to a graph which exposed pattern matching. But what does it mean to postcompose a graph with a pattern? Because graphs and patterns live in the same underlying category, we can certainly calculate this particular composition. We can think about it as applying a eliminator that isolates certain parts of a value first and then feeding the result into a constructor which builds a different value. To judge whether this composition yields an interesting operation on graphs is to be determined after more details have been worked out.

Furthermore, the duality between graphs and patterns shows some similarities to the language of the $\mu\tilde{\mu}$-calculus [25, 93]. In this framework, terms and coterms exist side-by-side. Both kinds can be constructed individually, but neither can compute by itself. Only when the two kinds interact a reduction can happen. We see some parallels to our graphs and patterns language, where we can compose both separately but also trigger some interaction by composing them *with each other*.

Finally, the theory of operads and cooperads is very rich and well studied. An interesting direction for future work would be to study the monad arising from the graph operad, and the comonad arising from the pattern cooperad, and to establish whether the interaction between the two can be formalised, e.g. in a distributive law [84].

Part II

A Data Type of Surface-Embedded Graphs

Chapter 4

Introduction

In this part we develop data structures and operations representing plane graphs and their rewriting systems 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.

Defining plane graphs in a functional programming language like Agda comes with various challenges. Firstly, graphs are cyclic structures which is not straight-forward to represent in a functional programming language. We aim to avoid unrolling cycles in a graph instead aim for a finite, inductive representation. Secondly, a direct implementation of graphs together with their embedding as described in the previous sections is not suitable in the environment of Agda. In the categorical considerations we modelled graphs from sets of edges and vertices together with some surface embedding information. In the Agda development we present plane graph embeddings as the native elements instead. Collections of unordered elements are not easily encoded in Agda. With inductive definitions of data types, structured sets are much more natural to represent. Therefore, instead of starting from unordered graphs and adding an order (e.g. in form of a rotation system) to impose a certain structure afterwards, we use the ordered structure in the first place and implement it as an Agda data type.

In Chapter 5 we construct such a data type of plane graphs. We solve the challenge of presenting the cyclic and over-connected characteristic of a graph by taking one of its *spanning trees* as a scaffold and adding the remaining edges on top of it. Figure 4.1 shows an example of a graph and its splitting into a spanning tree and the remaining edges. With this classification of a graph's edges and the structure of its spanning tree we are able to implement it as an inductive structure. Overall, we will be representing a plane graph by a traversal of its spanning tree where at each step we insert a finite number of additional edges at the current position.

(a) An example graph.

(b) A choice of spanning tree (drawn as thicker, blue edges) and root (drawn as a black square).

Figure 4.1: Example of a graph's spanning tree.

Using the definition of plane graphs as decorated spanning trees, we implement an operation which focuses on a particular regions in the graph which can then be targeted by a rewrite rule in Chapter 6. This operation specifies how to separate a subgraph from its context and thus ensures that a rewrite rule is applied locally. After the rewrite rules is applied, subgraph and context can be reassembled to form the overall result of the rewriting operation.

We introduce graphs with a focus as an instance of a zipper [51] for the graph's spanning tree. This zipper structure is configured to record the structure of additional edges alongside a path to the focus inside the spanning tree. The additional edges provide an interface for each region inside the graph. When applying a plane rewrite rule inside a plane graph, the preservation of the interface edges ensures that the result is a plane graph again.

In the context of graphical languages, focussing is an important operation for isolating a subtree in order to apply a rewrite rule. But focussing and navigating inside a data structure are interesting operations in other contexts, too. Another example of a system in which these operations are useful comes from the area of metaprogramming. In metaprogramming, programs are implemented as terms of a language and reasoning about programs as term transformations, for example for performing a type-checking procedure. In this context, the notion of focus typically means isolating a certain subterm which can be analysed or manipulated. The notion of interface between a subterm and its environment consists of free variables that are known in the context of a subterm.

We will explore the particular case of focussing inside plane graphs in depth in Chapter 6 and discuss a potential abstraction over a larger class of structures with a focus in Chapter 7.

4.1 Programming in Agda

Agda [79] is a functional programming language that implements an intensional dependent type theory similar to Martin-Löf Type Theory [68]. The Agda compiler is written in Haskell and the style of Agda programs is similar to Haskell, too. Additionally, Agda serves as a proof assistant as equivalence between two programs can be encoded directly in the system. We give a brief summary of the features of Agda that we use in this work. For a more thorough introduction, we refer to the literature [80, 91, 60].

Literate Agda Agda has a literate mode with which we can insert active (i.e. type checked) code into LaTeX(or html) documents. We will make use of this feature throughout this document: all code blocks are copied 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.

Typographical conventions We use the "Conor" colour scheme for syntax highlighting of Agda code in this document. This highlighting colours data and record types in blue, constructors and record fields in red, and function types in green. Additionally, keywords are displayed in black, and *bound variables* as well as comments (which we use for pseudocode) in grey.

Type universes Agda implements a universe hierarchy of types. The hierarchy starts with the type of small types, called Set. Types grow larger (meaning they may *contain* types of a smaller size) as we go up the hierarchy, Set_1, Set_2, and so on. Functions in Agda may be defined polymorphically in the universe of types in which case they abstract over the *level type*: a natural number indicating at which level of Set the function is operating at. We will make occasional but not extensive use of this feature in this work.

Data types and functions Data types in Agda are similar to generalised algebraic data types (GADTs) in Haskell. They are specified by the data keyword and contain the type's kind as well as a list of constructors expressing the possible ways of forming values of the type. Here is our first example, the type of Boolean values:

```
data Two : Set where
  ff : Two
```

```
  tt : Two
```

This code snippet defines a type Two of kind Set (i.e. a small type) with two constructors, tt and ff.

Functions in Agda are typically defined by pattern matching on their input arguments. Pattern matching splits an input argument into the possible constructors of the corresponding type, for each of which the function may be implemented differently. For example, we can define the function not which inverts the values of Booleans by pattern matching on the constructors of Two:

```
not : Two → Two
not tt = ff
not ff = tt
```

Another example of an Agda data types is the empty type Zero:

```
data Zero : Set where
```

This type does not have any constructors, therefore it is impossible to generate a value of type Zero. Defining functions on inputs of the empty type is trivial which is expressed in Agda by the *absurd* pattern:

```
magic : { A : Set } → Zero → A
magic ()
```

We can observe two further features of Agda in this example: Firstly, the function is defined *polymorphically* in the type A. This means that different instances of the magic function (one for each instance of A) have a single representation. Secondly, the type argument A is passed to the function *implicitly*, indicated by the curly brackets around it. When calling a function with an implicit argument, this argument becomes an unknown subterm that will be solved by unification. Sometimes we are interested in explicitly passing an implicit argument (for example if it cannot be solved by unification) which do by placing it inside a set of curly brackets again, e.g. magic { Two }.

Remark 4.2. A central concept in type theory is the connection with logic, expressed by the *Curry-Howard isomorphism* and often referred by the slogan "propositions as types". This concept identifies a data type with a proposition and the terms of the type as proofs of this

proposition. We can apply this isomorphism to our previous examples: There are two proofs of the type Two, interpreted as a proposition, namely ff and tt. The type Zero represents the empty proposition, i.e. falsity. This proposition cannot be proven which, in the type theory, corresponds to the fact that there are no values of type Zero.

Data types in Agda may be defined inductively. In this case, at least one of the constructors takes a term of the same type as an argument. A simple example are the natural numbers:

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat
```

Here, the constructor suc forms a term of type Nat when called with an already existing term of the same type.

Functions on inductive data types are typically defined by recursion. For example, addition for natural numbers Nat contains a recursive call in the inductive constructor case for suc:

```
_+_ : Nat → Nat → Nat
zero + n = n
(suc m) + n = suc (m + n)
```

This function is well formed as the argument of the recursive call gets structurally smaller at every step, a fact that is used by Agda's termination checker.

We introduce two more features with the help of the example of natural numbers: Firstly, _+_ is our first example of a mixfix operator [26] in Agda which is indicated by the underscores in the positions of the arguments. Secondly, in some cases we will be interested in using the original input argument to a function like _+_ without splitting it by the pattern it matches. We can do so by adding a variable name on the left hand side together with an @ symbol ahead of the term the variable stands for: m'@(suc m) + n = _. Whenever we use m' on the RHS of the function, it addresses the entire expression (suc m). This feature in Agda is called *as-pattern*.

Similarly to functions, data types can be defined polymorphically over another type. A simple example is the type of lists with elements of type X:

```
data List′ (X : Set) : Set where
  nil : List′ X
  cons : X → List′ X → List′ X
```

nil is the constructor for the empty list, and cons constructs a list by appending an element of type X to the front of a tail of type List′ X. (Remark: We will be using an alternative, but isomorphic, definition of lists throughout this work which we will introduce below.)

Indexed data types Data types in Agda may depend not only on other types but also on *values* of other types. These types are actually type families [32] and are a generalisation of generalised algebraic data types (GADTs) in Haskell. For example, the following is the type of finite sets of a certain size, indicated by a natural number n as the type's index:

```
data Fin : Nat → Set where
  zero : ∀ { n } → Fin (suc n)
  suc : ∀ { n } → (i : Fin n) → Fin (suc n)
```

Observe in this example that the value of the natural number varies across the constructors, it does not have to be known beforehand. This is the most general way of defining an indexed data type. The specification of the type Fin also shows that the names of constructors of different data types may be overloaded in Agda. Both the type of natural numbers Nat and the type of finite sets Fin have the same constructor names. Note that there is no instance of type Fin zero, both constructors produce terms of type Fin suc n.

For the implementation of functions on *indexed* data types we can make use of the very powerful tool of dependent pattern matching [71]. In addition to a case split into the different constructors, the pattern matching algorithm performs unification on the indices involved. This may solve some constraints on other input arguments or introduce constraints on the implementation of the function. This is particularly interesting when pattern matching on an element of the type of propositional equality which we introduce below.

To simplify working in systems with type families we define some syntactic sugar for the quantification of an index value. If an expression is indexed by a universally quantified value, we may wrap the expression in a set of square brackets:

```
[_] : { I : Set } → (I → Set) → Set
[_] { I } X = { i : I } → X i
```

Additionally, when specifying the type of a function that respects the index of a type family, we may write the function type with a different kind of arrow:

```
_→i_ : { I : Set } → (X Y : I → Set) → I → Set
(X →i Y) i = X i → Y i
```

Records types and copattern matching Records in Agda implement tuples with (potentially) multiple fields that can depend on previously introduced fields. The simplest example of a record is the type with one element:

```
record One : Set where
  constructor ⟨⟩
```

This record does not contain any fields at all, but we are nevertheless able to construct its values by using the constructor ⟨⟩ provided.

An important property in Agda is that the η-conversion rule holds for record types. This means a record is fully defined when giving a value for all its fields and equality between two instances of the same record type can be translated into equalities of their respective fields. One is defined as a record (as opposed to a data type with one constructor), because we want to consider any two values of type One to be equal without the need for inspecting them first. As the record type One does not contain any fields, η-conversion ensures this property immediately.

Another important example of a record is the type of dependent pairs, also called the *sigma type*:

```
record Σ { l } (S : Set l) (T : S → Set l) : Set l where
  constructor _,_
  field fst : S
        snd : T fst
```

This type has two fields, the first one of type S and the second one of type T fst. Note that the type of the second field depends on the value of the first.

There are various ways of specifying a term of a record type. As an example, let us consider three functions p, q, and r of the same type which takes two arguments a and b arranges them as a dependent pair:

```
p q r : { A : Set } { B : A → Set } → (a : A) → (b : (x : A) → B x) → Σ A B
```

Constructing a value of type Σ A B is done by giving a value for all its fields. We may do so by using the record keyword and providing a value for each (named) field:

```
p a b = record{ fst = a ; snd = b a }
```

Alternatively, we may use the constructor _,_ which is provided as part of the information in the type of dependent pairs:

```
q a b = a , (b a)
```

Finally, we may use copatterns [3] which introduce a different case in the function definition for each field of the record:

```
fst (r a b) = a
snd (r a b) = b a
```

Copatterns are very useful when the fields of the record take different additional arguments as these can be matched on individually in the corresponding line of definition.

Note that the three functions p, q, and r (i.e. the different methods of creating an element of a sigma type) are isomorphic.

An example instance of the dependent pair type are coproducts:

```
_+_ : Set → Set → Set
A + B = Σ Two λ { ff → A ; tt → B }
```

The first element of this pair is a Boolean value which encodes the number of options for the second field. The two values, encoded by ff and tt in the first component correspond to the left and right hand side of the coproduct. The implementation uses extended lambda-abstraction to encode the function in the second component of the sigma type. With this construction we can define an in-line function with different cases, such as the pattern matching on the Boolean value in this example.

An instance of a coproduct is the statement that a type X is "decidable":

```
Dec : Set → Set
Dec X = (X → Zero) + X
```

Decidability for a type means we can either construct a value of that type (expressed by the right hand side of the coproduct) or we can deduce falsity when assuming that a value of the type exists (expressed by the function $(X \to \mathsf{Zero})$ on the left hand side of the coproduct). Decidability is an important property of equality types, but because of the Curry-Howard correspondence the statement is meaningful for any type.

Program equivalence and equational reasoning In Agda, equality between two elements of a data type is itself declared a type. This makes it possible to state and work with proofs directly in the language. The type of equality of elements of type X is defined as follows:

```
data _≡_ { l } { X : Set l } (x : X) : X → Set where
  refl : x ≡ x
```

The equality expressed by this type is propositional as there is only one way to construct a value of $x \equiv y$. This is possible whenever x and y are equal by their *definition* in which case we can construct the proof of equality as a term of the equality type using the reflexivity constructor.

In turn, when we have a proof $x \equiv y$ as an input type, the only option of a successful pattern match on the proof is the constructor refl. This means that y has to be definitionally equal to x and any instance of y in the expression can be replaced by x. This strategy is used extensively by dependent pattern matching and making it such a powerful tool.

Example 4.3. A slightly more involved example of inductive data types and functions, which we will make extensive use of later, are lists and backwards lists. This example also includes an instance of the equality type _≡_.

We have seen the standard definition of the type of lists earlier. Instead of the direct definition, here we are using a layer of abstraction to represent lists and backwards lists as instances of the same data type. The following type describes free categories on a given relation:

```
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
```

Given a relation R, this function calculates R's reflexive transitive closure for two elements x and y. The function also takes an argument d which encodes the direction in which instances of the relation are sequenced together. The type Dir has two constructors, cons and snoc indicating whether to add new instances to the front or the back of the sequence. Note that in the type declaration of FCat the two indices of type X appear in different positions, one to the left of the colon of the data declaration, and one on the right. Both are indices of the same type, but x must have the same value across all constructors whereas the value of y can vary.

When instantiating an FCat with a particular direction d we get the notion of Stars and Rats. The naming is in analogy with the Kleene star operation (which precisely defines the reflexive transitive closure of a relation).

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

```
pattern [] = nil' refl
pattern _,-_ x xs = cons refl x xs
pattern _-,_ xz x = snoc refl xz x
```

In addition to the types Star and Rats themselves, we use Agda *patterns* synonyms to introduce syntactic sugar for the application one of their constructors. Firstly, we introduce the pattern [] to encode the empty list constructor which contains the proof that two elements x and y are equal. Additionally, for each direction we introduce a pattern encoding the addition of an element to an existing list: "cons" _,-_ in the forwards case and "snoc" _-,_ in the backwards case. These patterns are non-symmetric so we can distinguish between the two directions more easily.

Standard lists and backwards lists are implemented as instances of Star and Rats in our framework where we instantiate the types with the trivial relation. Elements of a list do not have to satisfy any condition on their values, therefore it makes sense to add a new element into a list without any further requirement introduced by the relation.

```
List : Set → Set
List A = Star { One } (\ _ _ → A) ⟨⟩ ⟨⟩
```

```
BList : Set → Set
BList A = Rats { One } (\ _ _ → A) ⟨⟩ ⟨⟩
```

We are interested in a number of operations on forwards and backwards structures, especially in transforming one into the other.

First of all, we are able to rotate a backwards list into a forward list. This operation reverses the order of elements, therefore the head of the backward input list becomes the head of the forward output list:

```
rotate : { X : Set } → BList X → List X
rotate [] = []
rotate (lz -, l) = l ,- rotate lz
```

Another operation is reverse which reverses the order of elements in a list, but keeps its forward orientation. This function uses ++ which computes the concatenation of two (forward) lists.

```
reverse : { X : Set } → List X → List X
reverse [] = []
reverse (x ,- xs) = reverse xs ++ (x ,- [])
```

Fish and Chips In addition to forwards and backwards versions of certain functions we will need operations that combine elements of both with each other. These operations are defined more generally on Star and Rats, but may be instantiated to lists. A pair of operations involving both forwards and backwards sequences are called "fish" and "chips" [43]. The fish operator _⟨*⟩⟨_ takes a backwards and a forwards sequence as inputs and computes a backwards sequence with the elements in the output in the "same" order as in the input:

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

Similarly, the chips operator _⟨*⟩⟩_ takes a backwards and a forwards sequence as inputs but returns a forwards sequence:

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

Fish and chips are defined on two non-empty sequences. We can create some special cases by calling the functions with the empty list as one of their inputs. These cases implement reversing the orientation of a list while keeping their order the same. For example, the function reverse *bz* corresponds to calling chips with an empty list as second argument: *bz* ⟨*⟩⟩ [].

Underscores In addition to indicating the fixity of operators, underscores in Agda can mean two things, depending on where they appear in an equation. When used as an argument of a function definition, an underscore stand for a variable without a name. This is useful when the variable is not used in the computation. For example, the type of non-dependent pairs is an instance of a Σ-type that does not use its first argument in the type of the second, and therefore looks like this:

```
_×_ : forall{ l } → Set l → Set l → Set l
S × T = Σ S λ _ → T
```

Note that this example contains two different types of underscores as × is defined as an infix operator.

If an underscore appears in the definition of a function (on the right hand side of an arrow), it stands for an implicit value that may be solved by Agda's unification algorithm.

Module system and variable blocks Agda implements a module system which creates a name space for the functions that are defined in it and the variables it binds. Similarly, with a variable block we can bring more variables into the scope of the program. Throughout this implementation, we will make use of modules extensively. If we do not explicitly quantify over a bound variable in a function declaration, this variable is a module parameter of the current module or declared as a variable and therefore in scope of the function.

4.2 Related work

Graphs in functional languages Representing cyclic and shared structures in a functional programming language has been the subject of various research. Some of these structures use an inductive spanning structure (e.g. a spine for cyclic lists) and add additional structure on top [11], similar to what will we use in our implementation of graphs. Other works focus on detecting cycles when defining operations on graphs, for example by marking nodes during an operation [35] or by using the features of a lazy functional language to unroll cycles a finite number of times only [56]. Another approach suggests entirely different algebraic structures to represent graphs with a small set of generators and combinators on them [78] on which graph properties and algorithms can be expressed.

Plane graphs The particular case of a representation of plane graphs is an important feature in the Rocq [10] proof of the Four Colour Theorem [41, 40]. This proof uses *hypermaps* in which the corners of a graph become the vertices and are related by different functions, depending on their positioning to each other and the edges in the original graph. This presentation specifies an spatial arrangement of the faces of the graph embedding, similar to the information in a rotation system, and its planarity can be checked by using an generalised Euler formula (recall Theorem 1.53). Graph operations are expressed by the different maps between the graph's corners. 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, meaning that the planarity property never has to be checked explicitly but holds by definition.

Contour categories In our work we store graphs as a traversal of their spanning tree. The order of the traversal ensures their topological property by enforcing an order on the non-tree edges. Categorical contours [74] follow a similar idea and have been developed for

context-free grammars. The trees represent generators of a context free language. A contour defines a particular traversal of the tree that can be used to check that a word is well formed in the language and properties about context-free languages can be expressed as categorical statements on their contour category representation.

Chapter 5

Plane Graphs in Agda

In this chapter we develop suitable data types and constructions in Agda for planarly embedded graphs and their rewriting. We start from the same specification and required properties for graphs as before, but because of the different framework of a functional programming language, the solution to the specification will be different from the rotation system approach in Part I. We will explain the design choices in the development of a data type of plane graphs in Agda and discuss a translation between this type and the presentation as rotation systems. For the implementation in Agda, we focus on the special case of plane graphs. Similar considerations to Chapter 3 apply: defining plane graph embeddings already covers the key constructions necessary for implementing more general surface embeddings. We discuss potential extensions of our development to higher-genus surfaces in Section 5.3. The data structure for graph embeddings in Agda has to incorporate both connectivity and topology information. We will discuss these requirements and define the data type of plane graphs in Section 5.1. We will show how to translate from this particular type of plane graphs to their representation as rotation systems as seen in Part I in Section 5.2.

Remark 5.1. Throughout the formalisation we consider all graphs to be *closed*, meaning that both ends of every edge are connected to a vertex. The resulting data structure serves as a representation of open graphs by using boundary vertices to represent input and output edges of a graph, such as discussed in detail in Section 2.1.

5.1 Graphs in Agda

When defining graphs in the framework of a monoidal category in Chapter 2, we started from a presentation of graphs with *sets* of edges and vertices. Afterwards we added rotation systems to

the definition to encode surface-embeddings of graphs. In an environment where we can create and transform sets easily, graphs are a much simpler structure than their embeddings, and we make use of this fact in the categorical setting. In programming though, the manipulation of sets is not as straight-forward. A common combinatorial representation of graphs in programming languages are adjacency matrices which store the connectivity relation between vertices and edges. In Agda, matrices are not very easy to work with, as they do not have an efficient representation as an inductive type. Both graphs and their matrix representation contain a lot of implicit equivalences, for example the order of edges around vertices for graphs. To avoid working with equivalence classes in Agda (at least for the moment), we would need to pick a representative for each of them. Furthermore, as we are interested ultimately in particular orderings of edges, identifying different orderings along the way does not make much sense.

Firstly, instead of encoding graphs as matrices we will use a direct representation of graphs which is more suggestive in the environment of Agda and provides a canonical order of storing and inspecting them. Secondly, instead of implementing graphs first and adding embedding information afterwards, we will implement graph embeddings directly. Both forgetting about the embedding information to create equivalence classes and adding it back in afterwards are laborious and (in our case, unnecessary) operations in Agda.

With Agda's rich type system at hand, we can be very precise about the data structures we describe by encoding their properties as part of their type. We use this feature to specify graphs which carry not only connectivity but also topology information. The type system ensures that any operation on an intrinsically plane graphs produces a plane graph by the preservation of the graph's type only. Moving the complexity into carefully designing the data type of graphs means that the proofs of planarity come for free.

5.1.1 Graphs are cyclic structures

One of the main challenges for implementing graphs in Agda is their *cyclic* nature. A path in a graph (i.e. an enumeration of adjacent edges) may form a cycle and iterating a graph may unroll a cycle and continue indefinitely. In finite graphs (which we cover in this implementation) we may encounter cycles, too, but we will eventually revisit nodes and edges in a traversal. Thus, finite graphs are not true infinite structures, but finite structures which may contain cycles. Infinite data type which have a finite representation have been studied as rational fixpoints of functors [75]. Other works on inductive definitions of cyclic data structures use strategies of splitting a graph into a non-cyclic and a cyclic part [46] or introducing backpointers in an

inductive structure [39]. This is similar to graphs which are implemented to express terms in syntaxes with binders, e.g. the lambda calculus [97]. Alternative approaches use the laziness of Haskell to only ever unroll as little of the cyclic structure as possible [11]. The aim is always to define algorithms on cyclic structures on the inductive part of the structure only. For our implementation of plane graphs we follow this approach to construct a finite, inductive representation of plane graphs. Crucially, the backpointer structure will have to be defined in order to express the planarity property.

We will split a graph into two parts by choosing a finite, cycle-free, connected, maximal subgraph as the first component. This is the inductive part of the graph which we can easily represent and manipulate in Agda. The second component contains the remaining structure of the graph which amounts to a subset of the graph's edges. These additional edges form the graph's cycles.

Observation 5.2. *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.*

Therefore, the choice of a suitable non-cyclic subgraph of a graph G amounts to a choice of one of G's spanning trees. Any spanning tree includes all of G's vertices, therefore the remaining information contains edges of G only. Our representation of graphs will therefore encode an *over-connected* tree. Trees are very straight-forward to represent as an inductive type in Agda. We take the spanning tree of a graph as a frame to which we add the additional edges at the relevant positions. Remember Figure 4.1 for an example.

Remark 5.3. The choice of spanning tree in a graph does not matter at this point. In the particular case of graphs as a representation of string diagrams, we can always compute a canonical spanning tree: As we know the splitting of the rotation at the boundary vertex into inputs and outputs, we can pick the corner between last input and first output edge to be the root of the spanning tree, and compute the tree by an (existing) algorithm of our choice (for example a depth-first traversal). This approach ensures that equal string diagrams have equal representations as decorated trees.

For representing plane graph more generally, we may be interested in considering equivalence classes of all its spanning trees. We will discuss an operation on graphs in Section 6.3 that moves the root of the spanning tree to a different corner, which is related to this question. This operation keeps the edge structure of the graph, but changes the order in which to traverse the

spanning tree, according to where we move the root to. We will not attempt representing the entire equivalence class of different spanning trees for the same graph embedding here, but consider it an interesting future project.

Remark 5.4. In the formalisation we are working with *undirected* graphs. Our main goal is to find a suitable encoding of graphs embedded in the plane. Directed edges add a layer of complexity to the implementation of a graph without affecting topological considerations. The approach of splitting a graph into one of its spanning trees and additional edges works equally in the case of undirected and directed graphs. We will not introduce the unnecessary bookkeeping of edge directions and instead focus on our main goal: capturing a graph's embedding information in a suitable way as part of its type.

Remark 5.5. For the same reason as discussed in Remark 1.64, we consider connected graphs only.

5.1.2 The order of edges matters

Even though we will not implement plane graphs by using rotation systems directly in the format of lists of edges, considering the ordering of edges in a surface-embedded graph is still crucial to our development. We begin by making some general observations about surface-embedded graphs and their plane subgraphs.

Lemma 5.6. *Let G be a surface-embedded graph; then:*

- *For any edge e, the edge contraction $G - e$ embeds in the same surface. (This is a recall from Proposition 3.3.)*
- *The edge contraction of any plane subgraph of G embeds in the same surface.*
- *A tree is trivially a plane graph.*
- *Contracting a graph's spanning tree does not change its genus.*

As the graph's spanning tree does not hold any of its embedding information, the additional edges which are left after the contracting the spanning tree carry all of the graph's topological information.

Corollary 5.7. *The structure of edges that are not included in a graph's spanning tree alone determine its genus.*

Contracting an edge merges its source and target vertices. Because a graph's spanning tree contains *all* vertices of a graph, contracting it results in a graph with just one single vertex, with all remaining edges attached to it. Figure 5.8 shows the contraction operation step-by-step with each stage highlighting the edge that will be contracted next.

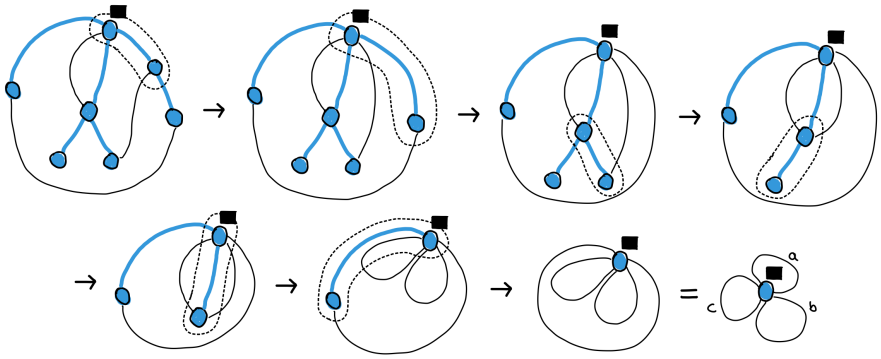


Figure 5.8: Step-by-step contraction of the spanning tree of a graph.

The resulting graph of the contraction in Figure 5.8 is a bouquet graph. Bouquet graphs have already made an appearance in Section 3.1 and now we define them properly.

Definition 5.9. 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 of edges in which each $e \in E$ appears exactly twice.

As the structure of non-tree edges is key for specifying a graph's surface embedding, we have to store them in a certain order. This order is encoded by the rotation at the central vertex of the bouquet graph resulting from contracting the graph's spanning tree. The traversal of the graph's spanning tree in a clockwise determines an order of the non-tree edges. Therefore, it is sufficient to record their occurrence in the tree traversal. The clockwise traversal of our example graph is illustrated in Figure 5.12a.

The traversal of the graph moves through the entire structure (in a certain order) 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*:

Definition 5.10. 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.

Remark 5.11. In a surface-embedded graph a corner is either the entire neighbourhood of a vertex (in case it does not have any incident edges) or 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.

(a) Traversal of the spanning tree.

(b) Corners (including the root) of the graph.

Figure 5.12: Spanning tree representation of the example graph.

An example of the corners of a graph embedding is illustrated in Figure 5.12b. As the set of corners all possible positions inside a graph, we can use it to root the graph's spanning tree by marking one of its corners.

Definition 5.13. The *root* of a graph is a distinguished corner.

Using corners as the notion of positions inside a graph, we now specify the graph traversal operation as the following algorithm:

Algorithm 5.14. 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 taking one of the following three steps:

1. Following along a tree edge e: this operation progresses the traversal from vertex $v1$ to vertex $v2$ which is adjacent to $v1$ by e. The traversal continues recursively from this position.

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 non-tree edge: this operation passes a non-tree 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.

Note that two of the steps record additional information such as a corner or a non-tree edge, and only the first step moves the traversal along the spanning tree structure.

In the Agda implementation of plane graph embeddings we will introduce a data type of Steps which describes precisely the choice of one of the operations to progress a traversal. The

three steps specified in Algorithm 5.14 correspond to constructors span, push/pop, and corner of the Step data type. The full traversal is implemented as a sequence of these Steps. More details are explained in Section 5.1.3.

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 5.15. *In a traversal of a graph's spanning tree, edges and corners always alternate.*

Proof. Every corner is defined by a vertex and two neighbouring edges at that vertex.

- The only possible way progress the traversal after a corner has been visited is to inspect one of its two defining edges (whether it is a tree edge or a non-tree edge).
- When recording a non-tree 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 tree edge (by the definition of spanning tree), therefore, in addition to e the vertex v has at least one more edge attached to it. 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 along a tree edge t from v to a different vertex v', there is at least one corner located at vertex v' with one of its defining edges being t and located next to t at v' in a clockwise order. This is the next element that will be visited in a traversal.

□

5.1.3 The graph data type

We will explain the implementation of the data type of plane graphs in various stages. We will start by defining one Step of the traversal. Afterwards we show how to combine multiple steps in sequence which form the full traversal operation. Finally, we add some degenerate cases of plane graphs which completes the data type.

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 5.15 that the traversal of a tree follows a strict pattern of alternating edges and corners. We use this information to guide the implementation of a Step in the traversal. Furthermore, in the specification of a graph traversal in Algorithm 5.14 we assume a separate data structure to store non-tree edges. We incorporate this data structure into the specification of an indexing type for Steps.

Indexing type A traversal type contains two components:

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

The first set in TravTy is a list of edges. This list records the non-tree edges throughout the traversal. Whenever we encounter one of these edges, the list changes in a *stack-like* fashion: the first time we visit a non-tree edge we add it to the front of the, and the second time we remove it from the front of the list. The stack-like property means that edges which have been added most recently have to be removed first. We will explain the implication of this behaviour in Subsection 5.1.4. Every step in the traversal is indexed by its effect on the stack of non-tree edges. When implementing the traversal of an entire graph, we will ask for this stack to be empty at the start and the end of operation.

```
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 5.16: 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 (i.e. a version of the Boolean type) with two constructors, corner and edge, see Figure 5.16. Throughout the traversal we use elements of Next as a "tag" structure, specifying what kind of data in the graph is currently visited. The lifting of a tag to the data it labels is implemented by the function What (see Figure 5.16) which maps 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 5.15: in a graph traversal, edges and corner always occur alternating. We enforce this alternation by using the tagging system and a function after (see Figure 5.16) which changes the tag from an edge to a corner, and vice versa. Both for the clockwise traversal of an entire graphs as well as for inserting smaller subgraphs at certain positions, elements of the traversal type ensure that the data fits together in the right way, by construction.

With the indexing structure for a graph traversal in place, we now specify what a traversal Step looks like.

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.

```
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)
```

Each step in the traversal is formed from one of three scenarios, all of which are illustrated in Figure 5.17.

Firstly, we may encounter a corner c. This Step does not change the stack of edges, but it enforces the next element in the traversal to be an edge. Secondly, we may encounter an edge e which is not part of the spanning tree. The first time we do so, we push it onto the stack es of non-tree edges, and the second time it is popped from the stack. We will see shortly (in Theorem 5.22) 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 will have to be a corner. The remaining constructor describes the situation in which the next edge e is an element of the spanning tree. In this situation we follow along the edge to reach a subtree, rooted at vertex v. This subtree is represented by a *sequence* of steps around the child node v. This sequence is defined similar to a list while ensuring that the indexing types for every two neighbouring elements match. We implement sequences of Steps by using Stars, as introduced in Example 4.3. A Star of Steps specifies multiple steps in a sequence while ensuring the alternation of corners and edges and while keeping track of the list of non-tree edges. In the constructor span we ask for an explicit edge e connecting to the vertex v which is the root vertex of the particular 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

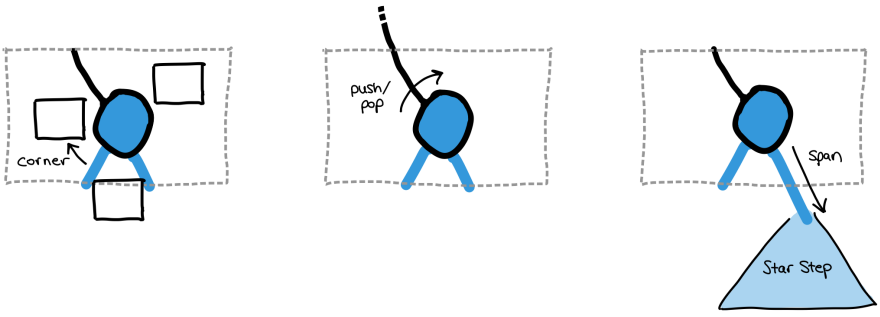


Figure 5.17: The different cases of constructing a Step of the traversal.

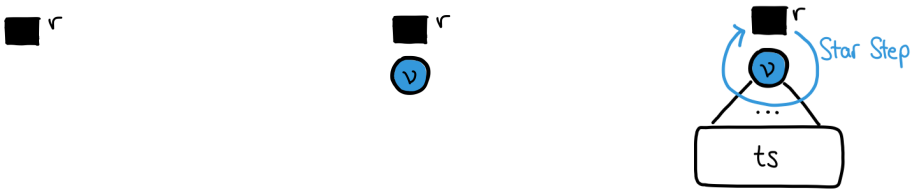
we have entered the subtree. The subtree must contain at least a corner, thus the Star Step is non-empty (which encoded by the fact that its indexing type changes from corner to edge). Overall, span describes a Step, progressing from the edge e to expecting a corner as the next element in the traversal. As the traversal of a subtree may add or remove edge from the stack, the span constructor passes the current 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, thus it is another instance of a Star Step.

The type of plane graphs We now 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 5.18. 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 : (r : C) → Graph
  vertex : (r : C) → (v : V) → Graph
  tree   : (r : C) → (v : V) → Star Step ([] , edge) ([] , corner) → Graph
```



(a) Empty graph, empty r. (b) Discrete vertex, vertex r v. (c) General case, tree r v ts.

Figure 5.19: The different ways to construct an element of type Graph.

First of all, a graph may be empty in which case it contains neither a vertex nor an edge, but a root corner r. Another degenerate case is a graph containing a single vertex v with no edges attached, see Figure 5.19b. 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. For an entire Graph we ask for the stack of non-tree 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, see Remark 5.1.

The tree constructor for a Graph specifies a graph with a non-empty set of subgraphs, as the other two constructors already cover the base cases. We achieve this by explicitly 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.

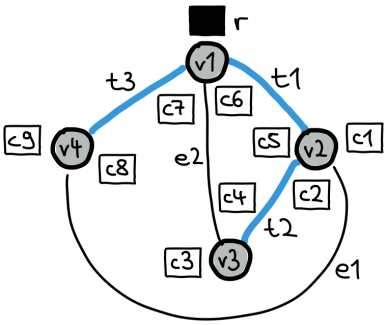


Figure 5.20: Example graph with vertices v, corners c, tree edges t, and non-tree edges e.

Example 5.21. Assume variables $(v1, \ldots, v4 : V)$ for vertices, $(t1, t2, t3 : E)$ for tree edges, $(e1, e2 : E)$ for non-tree edges, and $(c1, \ldots, c9 : C)$ for corners. The example graph in Figure 5.20 is implemented by the following term:

```
ex-graph : Graph
ex-graph
  = 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 ,- []) ,- [])
```

We observe in particular the alternation of edge operations (span,push, and pop) with corners (as discussed in Proposition 5.15).

5.1.4 Planarity

In Lemma 5.6 we have observed that the non-tree edges alone determine the surface a graph is embedded in. Therefore, the data structure we use to organise non-tree 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, compare Definition 3.5. 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 graphs.

Theorem 5.22. *A graph is plane if and only if it can be expressed as a term of the type Graph.*

Proof. Firstly, 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 (see Remark 5.25). Therefore, the constructor tree implements a plane graph.

Secondly, 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 get a corresponding bouquet graph, and then show that using a stack is precisely the data structure that ensures planarity of the bouquet graph:

1. Assume a spanning tree 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 cannot 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 non-tree edges as before the contraction. We repeat the process for all edges in the spanning tree. Because a spanning tree contains all vertices in the graph, we get a bouquet graph with edges being the non-tree edges.

The rotation of the central vertex in the bouquet amounts to the order of non-tree 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 5.9 of bouquet graph, 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.

□

Remark 5.23. The data type Graph defines a structure similar to *blossoming trees* [4] which are a trees additionally equipped with half edges at certain vertices. Connecting the half edges in a suitable (i.e. non-crossing) way constructs faces of a plane surface embedding.

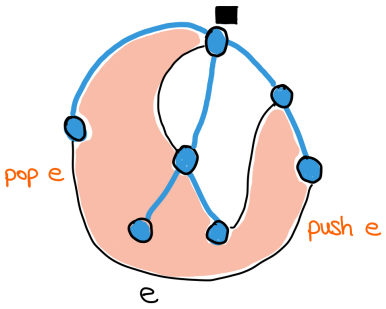


Figure 5.24: A non-tree edge enclosing a region (shaded) of the embedding.

Remark 5.25. The statement of Theorem 5.22 can be interpreted in an alternative way: Every non-tree edge e *completes* a face of a graph's surface embedding. This is illustrated in Figure 5.24. Inserting a non-tree edge into a spanning tree amounts to splitting the surface into two regions, an "inner one" containing the newly completed face, and an "outer". 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 5.24 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 the illustration in Figure 5.24.

5.2 Translation to rotation systems

In Chapter 3 we used rotation systems to represent a graph's surface embedding. In this chapter we have used the particular traversal order of a graph's spanning to define its (plane) embedding. Both representations determine a graph's surface embedding, but for technical reasons we prefer to use a different representation in the implementation than in the formulation of categorical rewriting for graphs. For the construction of a category of graphs we were aiming to keep the structures as closely related to graphs as possible to be able to use well known techniques on graphs, such as double-pushout rewriting. Having constructed a category of total graphs, the switch from sets to cyclic lists of edges is a straight-forward extension of the framework. For the environment of Agda we have chosen a different notion of graphs by expressing them as spanning tree traversals. This approach provides an ordering of the elements in a graph which we encode as the inductive data structure Graph. Importantly, both notions of plane surface embeddings contain the same information: a graph embedding is defined by its *faces* which we can compute from either representation. Given a rotation system, we can trace the faces of the corresponding graph embedding by following the next edge in a vertex' rotation until we get back to the edge we started with. In the spanning tree representation, every non-tree edge splits a face into two (cf. Remark 5.25) which we can use to construct all faces of the embedding.

In this section we show how to translate directly from the data type representation to rotation systems. We take an an element of type Graph and show how to calculate its rotation system. Every vertex in the graph carries a Star Step of its subtrees (as illustrated in Figure 5.26) from which we can directly "read off" the vertex' rotation:

```
starRotation : Star Step k l → (One + E) → List E
starRotation [] (inl ⟨⟩) = []
starRotation [] (inr e) = e ,- []
starRotation (corner c   ,- s) oe =     starRotation s oe
starRotation (push e     ,- s) oe = e ,- starRotation s oe
starRotation (pop e      ,- s) oe = e ,- starRotation s oe
starRotation (span e _ _ ,- s) oe = e ,- starRotation s oe
```

(a) Top level node. (b) Inner node.

Figure 5.26: The rotation around a vertex in a Graph.

starRotation inspects the elements in a vertex' Star Step. If the Star Step is empty, we have to distinguish whether the vertex is the root of the graph or it is located further inside the graph and accessed via an edge. This distinction is encoded by the sum (One + E) where (inl $\langle\rangle$) indicates that we are at the top level and v is the root of the graph and (inr e) encodes that we are in a subtree, accessed by an edge e. (NB inl and inr are pattern synonyms defined for the type _+_ of coproducts.) In the case of a subtree, the edge by which it is visited has to be included in the vertex' rotation, too. In the case of a non-empty Star Step, starRotation inspects each of its elements. corners can be ignored as only edges occur in a rotation system. In the case of a pushed or popped edge, this edge is added to the rotation. If an edge spans a subtree, it also is recorded. The function starToRotation inspects each element of the vertex v subtrees. But as we are only interested in the rotation at v (and not v's children nodes), considering the top level of each element in the Star Step is enough. This means that we do not recursively compute the rotations of a subtree, but only recording the edge by which it is connected to v.

To compute the rotation system for the whole graph, we traverse its spanning tree structure and compute the rotation for every vertex we encounter. To avoid duplication of rotations, this operation requires two mutually defined functions:

```
collect    : Star Step k l → (One + E) → List (List E)
topLevel : Star Step k l → (One + E) → List (List E)

topLevel ss oe = starRotation ss oe ,- collect ss oe

collect [] oe = []
collect (span e vs sts ,- ss) oe = topLevel sts (inr e) ++ collect ss oe
collect (_ ,- ss) oe = collect ss oe
```

The function topLevel is called whenever we encounter a new vertex in the traversal, starting with the root vertex. It computes the rotation for the vertex by calling starRotation, and then invokes collect for the same Star Step, in order to compute the rotations of any

potential subtrees of that vertex. The function collect traverses every element of the given spanning tree. Any pushes, pops or corners can be ignored in this traversal as these elements are taken care of by the operation starToRotation. The only interesting case is a span e v sts. This is precisely the case when we encounter a new vertex v in the traversal. collect now calls the topLevel function, with the v at the top level. As this vertex is visited via the edge e in the traversal, topLevel is called with the parameter (inr , e), indicating that e has to be an element of the rotation at v. The distinction between the two functions topLevel and collect is important, because the calculation of the rotation systems has to only be performed *once* per vertex. On the contrary, a vertex might have multiple spanned subtrees and collect has to be called once per subtree. To avoid duplication of the vertex' rotation, the two functions have to be kept separate from each other.

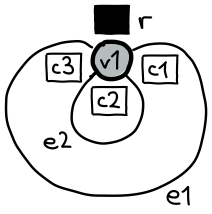
To compute the entire rotation system for any graph we can now call topLevel on the graph's traversal Star Step:

```
graphToRotations : Graph → List (List E)
graphToRotations (empty _) = []
graphToRotations (vertex r v) = [] ,- []
graphToRotations (tree r v st) = topLevel st (inl ⟨⟩)
```

This function distinguishes two degenerate cases of a Graph: An empty graph has an empty rotation system. The graph with a single vertex v consists of the rotation at v but because there one rotation (at the vertex) but as there are no edges in this graph, the rotation is empty. Therefore the overall rotation system is a singleton list with its element being the empty rotation. In the general case of a tree traversal, this function calls topLevel to collect the rotations of all vertices in the graph. The last parameter of this function call is (inl , ⟨⟩), indicating that v is the root of the overall graph.

Example 5.27. 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 below and implemented in Agda like this:

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

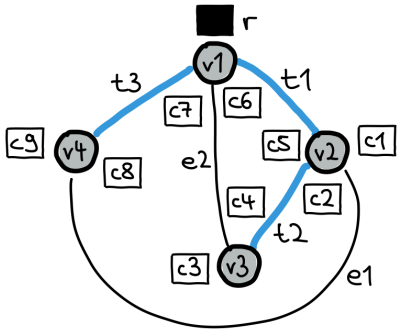


We can calculate the rotations for this graph using the function graphToRotations as follows. We state the result as a propositional equality to display the expanded term which shows the rotation at the only vertex *v1*.

```
rot-bouquet : graphToRotations ex-bouquet ≡ (e1 ,- e2 ,- e2 ,- e1 ,- []) ,- []
rot-bouquet = refl
```

Example 5.28. As another example, remember the graph from Example 5.21. Computing the rotation system for this graph results in the following list of lists:

```
ex-rot-graph : graphToRotations ex-graph
  ≡ (t1 ,- e2 ,- t3 ,- [])
  ,- (e1 ,- t2 ,- t1 ,- [])
  ,- (e2 ,- t2 ,- [])
  ,- (e1 ,- t3 ,- [])
  ,- []
ex-rot-graph = refl
```



The converse direction of translating between a Graph and its rotation system is not as straight-forward. This is because the spanning tree representation includes a choice of spanning tree and root. Each of these choices results in a different term of type Graph but all of them have the same rotation system. We may be interested in equivalence classes of spanning trees in the future, as we discuss in Section 5.3. A translation from a rotation system to a Graph would require to choose a spanning tree for a graph which is an operation we have deliberately left out of the development.

5.3 Future work

Equivalent graphs So far in our framework, two equal graphs embeddings with different spanning trees are not represented as equal structures. In the applications of graphs as representations of string diagrams this might not be a large drawback. As we know the first input edge for any string diagram, we can use a standard algorithm to calculate the spanning tree starting at this particular edge, as discussed in Remark 5.3. In general though, graphs are non-directed structures and by introducing a spanning tree we choose a certain direction. This is crucial for the representation of graphs as inductive structures because we need a

deterministic way of inspecting substructures and define operations. Specifying equivalence classes of spanning trees is an interesting extension as it would allow us to talk about undirected structures, independent of the choice of spanning tree.

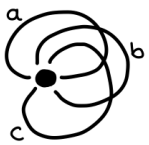
Higher genus graphs In Lemma 5.6 we have discussed that the structure of the non-tree edges alone determining the genus of the surface a graph can be embedded in. In particular, if this structure is a stack, the graph is plane, as shown in Theorem 5.22. In general, we are interested in a data type of graphs that are embedded in a higher genus surface. Instead of a stack for the non-tree edges we will need a different data structure to restrict the order of edge access.

Consider a torus, the surface one genus above the sphere. As we are only interested in edges that are not in the spanning tree, we can consider a bouquet graph which only contains this type of edges. As a generalisation of a stack of non-tree edges, the most obvious candidate for the torus are two stacks. Unfortunately, this data structure is too expressive and at the same time not expressive enough for toroidal embeddings. We consider two examples of bouquet graphs and their embeddings:

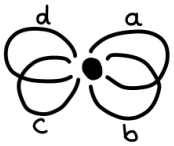
- A graph embedded on the torus may contain edges which are nested up to level 3, meaning that in the traversal we can move past the first end of three edges and then encounter their opposite ends in the same order. The smallest example of this type of graph is pictured in Figure 5.29a, with rotation at the central vertex $[a, b, c, a, b, c]$. This is a valid graph embedding on the torus, and a forbidden minor on the sphere (meaning that any graph containing it as a subgraph cannot be plane), thus it is an important special case distinguishing plane and toroidal graph embeddings. Unfortunately, for representing this graph's traversal, two stacks are not sufficient. With two stacks, maximally two edges can cross each other at any point, and the nesting of this example graph is too complex.

- On the other hand, consider the graph with two pairs of crossing self-loops, as illustrated in Figure 5.29b. The rotation at the central vertex is $[a, b, a, b, c, d, c, d]$. The traversal of this graph *can* be realised with two stacks. But this graph is not embeddable in the torus, in fact, is it a forbidden minor for toroidal embeddings. Thus, for this example, two stacks are not restrictive enough to represent toroidal embeddings.

Researching data structures to express higher genus surfaces which generalise our implementation with stacks would be a very interesting future work and extension of our framework.



(a) Forbidden minor on the sphere.



(b) Forbidden minor on the torus.

Figure 5.29: Special cases of graph embeddings, distinguishing graphs of different genus.

Over-connected inductive structures Graphs are one example of an over-connected structure which we represent as an inductive type (the spanning tree) with some additional structure on top of it (the non-tree edges), stored in a particular order (stacks) to represent the properties of the surface embedding which we are interested in. Infinite structures that have a finite (inductive) representation are described by rational fixpoints of endofunctors [75]. Any data structure that has back-pointers in an otherwise inductive structures can be described by such a rational fixpoint. Exploring this connection further and developing a syntax for over-connected data types is an interesting future project.

Chapter 6

Focussing Inside Plane Graphs

For using graphs as representations of string diagrams, we need to be able to express their rewriting theory as rewriting a subgraph is an important operation for modelling diagrammatic reasoning. The implementation of rewriting for graphs requires a notion of partitioning a graph into a subgraph in focus and its context graph. This partition ensures that a rewrite rule is targeting a certain subgraph only and does not affect the context. Recall from Section 1.3 that, in a DPO rewriting step, the construction of the pushout complement calculates the subtraction of the graph in focus from the overall graph, returning just the context graph. For the notion of plane graphs in Agda, we will construct a number of intermediate structures capture these important operations used in the application of a rewrite rule. The main development is the notion of *focussing* on a substructure inside a larger Graph. This involves both a notion of highlighting of a subgraph in question and the calculation of the context when the subgraph is removed.

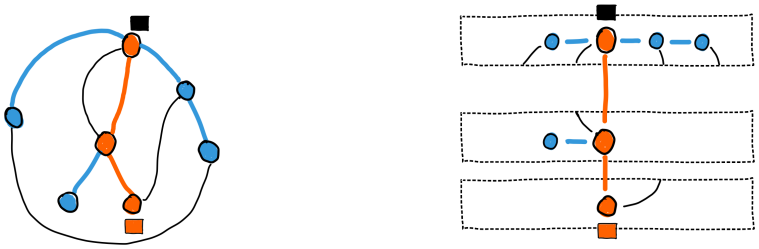
We define partitioning of graphs into a context and a focus as an instance of a zipper [51] for its spanning tree.

6.1 Zippers

Zippers define a data structures to focus on a certain substructure inside an overall inductive type like a list or a tree. They split the overall data structure into a context and the substructure in focus. Contexts are defined as paths between the root of the structure and the its focus. At every step in the path, the sibling substructures to the left and right are recorded. Importantly, paths are stored bottom-up, starting at the focus and growing upwards towards the root. This allows for fast access to the immediate neighbour structures from the position of the focus.

We can apply a local modification easily without the need to recalculate the entire structure. Additionally, zippers come with an operation to move up and down the path between root and focus. This is useful for inspecting sibling substructures, but also to calculate between the structure with a focus and the overall global structure. A zipper together with the operation to move along the path step-by-step defines a *cursor* which, similar to a cursor in a text buffer, can move onto its immediate neighbours in all directions quickly.

Remark 6.1. For now we will define zippers to focus on a particular *corner* in a Graph, rather than a whole subgraph. When we explain rewriting of subgraphs in Section 6.4, we will discuss how to choose a larger region of a graph as the focus. For constructing the path structure and building the zipper from it, focussing on an individual corner is sufficient.



(a) One corner in focus.

(b) The layers of the zipper.

Figure 6.2: Example of a graph's zipper.

A context for an individual corner is the rest of the graph. Therefore, a path to a corner is a path in the spanning tree to that particular corner. An illustration of a path and a corner in focus is shown in Figure 6.2a. At every step in the path, the neighbouring subgraphs to the left and right are recorded. Because we do not just store trees, but also additional non-tree edges, we have to extend the notion of zipper to include the stacks of those non-tree edges. In a standard clockwise graph traversal, the stack of non-tree edges may be non-empty at the position of the corner in focus of the zipper. Therefore, the zipper must record this state of the stack as part of its focus information. Additionally, each of the path's layers does not only store a list of sibling subgraphs, but also a potential change of the stack of non-tree edges. The layer structure of the example zipper (including half edges to indicate the change of stack) is shown in Figure 6.2b.

Each layer of the path is located around a vertex of the graph, and contains a number of subgraphs to the left and right of this central vertex. The subgraphs may contain tree and non-tree edges as well as other vertices. Multiple layers are connected with each other via spanning tree edges between their central vertices. A full path will be implemented as a list of

layers, starting at the focussed corner and finishing at the root of the graph.

We achieve the recording of non-tree edges in a zipper by introducing an indexing type for the layers in its path. The indexing type contains of two edge stacks which represent the state of the stack to the left and the right of the path. Each layer is indexed by a pair of elements of the indexing type (thus fourth stacks in total), one indexing the "root-side" of the layer and one indexing the "focus-side". The traversal of the subtrees inside a layer influences the changes in stack structure. The overall path records the change of non-tree edges between the stack at the focus of the zipper and the empty stack at the graph's root. Encoding the stacks of non-tree edges as part of the path's index ensures that the planarity property is preserved when a subset of the layers is replaced by a new one, as long as the change in edge stack for the new layers (aka their index) is the same as for the old layers.

We will start the development by the explanation of this indexing type followed by the definition of the layers in a zipper's path, before we specify the type of zippers for Graphs.

6.1.1 Indexing type

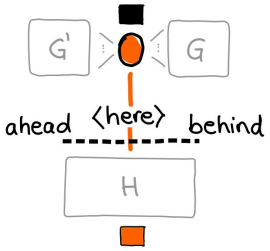
To keep track of the stack of non-tree edges and preserve the surface of the embedding, each layer in the path will be relating elements of a *zipper type*:

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

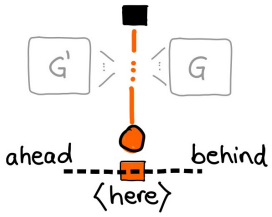
To explain the elements of a zipper type, we imagine a horizontal cut through a path, as illustrated in Figure 6.3. The indexing type describes the state either in between two layers (Figure 6.3a) or at one end of the path (Figure 6.3b).

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 every other position on the path, the ZipTy describes the state of a tree edge.

The other two fields in a ZipTy are recording the state of the non-tree 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



(a) Indexing type ZipTy in between two layers where here = edge.



(b) Zipper type at the position of the focus where here = corner.

Figure 6.3: Illustration of a zipper type as a horizontal cut through a path.

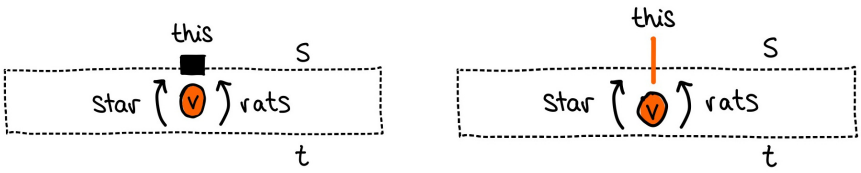
already pushed onto the stack (by the subgraph G in Figure 6.3). The state of the stack is stored in the field behind, emphasising that it represents the information in the past traversal of here. To continue the traversal from here, we move further down into the structure. In case of here = edge this means traversing the subtree H that the edge is leading to, in case of here = corner this subtree is empty and we immediately return. When the traversal returns from the 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.

6.1.2 Path structure

Every step in the path is associated with a vertex v and consists of v's subgraphs that are located to the left and right of the path. Additionally, it stores the edge via which the path continues to the next vertex located 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 *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. The following information is store in each Reyal:

```
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)
```

An illustration of a Reyal s t is shown in Figure 6.4.



(a) s = _ ⟨ corner ⟩ _ (b) s = _ ⟨ edge ⟩ _

Figure 6.4: Illustration of the information in a Reyal s t.

Let us have a look at the fields of a Reyal in detail:

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 vertex it is linked to, a Reyal also contains a field this which stores here of s towards the root. This element is either the root corner (see Figure 6.4a) or an edge (Figure 6.4b). The sibling subtrees alongside the path in a Reyal are stored in the fields stars 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. The orientation of stars and rats allow for easy access to the sibling subtrees closest to the focus of the zipper:

Rats **and** Pets To resemble the cursor-like nature of the original definition of Huet's Zipper, we mimic fast access to the immediate neighbours of the corner in focus. Therefore, the sibling subgraphs at each Reyal are stored with a particular direction: on the left of the vertex we store a standard Star Step of subtrees, with the head of the sequence being located at the focus-side. For the right hand side of the vertex we use a reverse version of a Star Step to ensure the same property of fast access to the subgraph closest to the focus. This means that the subtrees on the right of each vertex on the path are store in an *anticlockwise* orientation.

The backwards version of a Step is 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)
```

and we can sequence Pets together by using the reverse relation composition Rats (recall

Example 4.3). Importantly, the information in a Rats Pets is the same than in a Star Step. The only difference is their orientation.

Sequences of Reyals

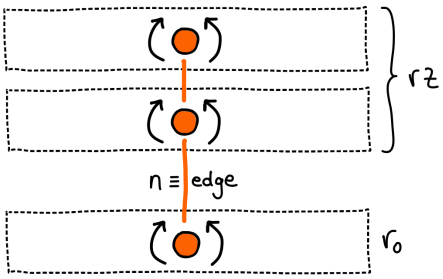


Figure 6.5: Illustration of the connectivity property of a Rats Reyal.

An entire path is defined as the concatenation of multiple Reyals, or, more precisely, a Rats Reyal. To construct it, we can use standard relation composition with one additional requirement: concatenation of two Reyals is well defined only if they share an edge between them. We are not allowed to compose a Reyal if its index has shape (_ ⟨ corner ⟩ _) which indicates that it is located at one of the ends of the path. This property of a Rats Reyal 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 the path consists of two Reyals or more. We encode this in the first case of the definition of ConReyals by asking the inputs Rats Reyal to be a snoc as well as its tail *rz* being 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 6.6. A Reyal's distinguished element this is located at its root-side. Therefore a Rats Reyal does not include the corner that is actually in focus, but rather leaves a *hole* in its position. Thus, a path stores the *context* for a particular corner in the graph. Constructing the zipper contains adding the corner in focus which will fit into this hole.

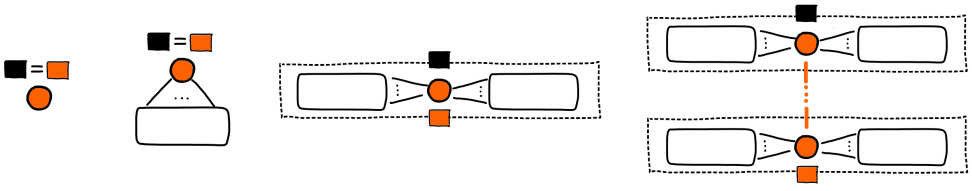
6.1.3 The type of zippers for Graphs

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 type Zipper is indexed by a list of edges. This list represents the state of the stack of non-tree 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. Because the focus consists of a corner only, the state of the stack is the same to either side of it, therefore we can express it as a single list.

All constructors for a Zipper ask for a element of type C which is the corner in focus. The remaining input information encodes the context of the 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 6.7 and we will now describe each of them.



(a) vertex (b) root (c) root-vtx (d) Reyals

Figure 6.7: The different constructors of a Zipper.

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 6.7a. Because there are no edges present in the graph, the list of edges in the zipper's index is empty.

Root corner A Zipper may focus on the root corner of a more complex graph, see Figure 6.7b. In this case, the entire graph (without its root) constitutes the context. We do not need a layer structure to encode this case, but a single Star Step is sufficient. Additionally, as the path from the root to the focus is empty, so is the stack of non-tree edges at the focus of the Zipper (and hence its index is []).

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. 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 that we do not 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 6.8.

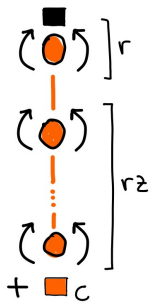


Figure 6.8: reyals.

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

Proposition 6.9. *A Zipper defines a plane graph context together with a corner in focus.*

Proof. The stack of the non-tree 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 6.11. As a first example, here is a zipper of a graph consisting of one vertex $v1$ and one self-loop $e1$. The focus of the zipper is the corner $c1$ which is the only other corner apart

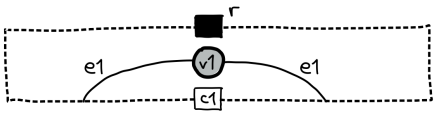


Figure 6.10: Example of a zipper focussed on the root vertex.

from the root *r*. The zipper is illustrated in Figure 6.10, and implemented using the root-vtx constructor as only one reyal is needed 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
```

Example 6.12. This larger example of a zipper assumes vertices *v1*, *v2*, *v3*, tree edges *t1* and *t2*, a non-tree edge *e1*, a root corner *r*, and further corners *c1*,...,*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 in Figure 6.13 and implemented as follows:

```
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
```

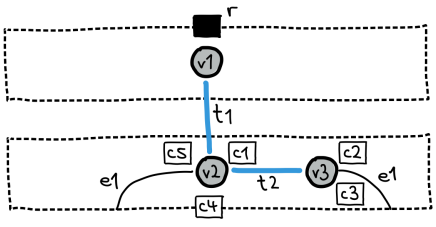


Figure 6.13: Example of a zipper with two reyals.

6.2 Computing the original tree

As discussed above, a zipper partitions a graph into a context and a corner in focus. In this section we explain how to re-calculate the original graph from one of its zippers. Given the data of a zipper — a context graph and a corner in focus — this operation combines the two elements and computes the overall clockwise traversal of the graph's spanning tree, starting at its original root.

This operation is particularly important when we implement rewriting of subgraphs. The substitution of a subgraph operates entirely locally, leaving the context graph unchanged. In the zipper representation of the graph, this means that the path does not change, only the elements in the focus of the zipper. After a rewrite has been applied, we are able to compute the overall result graph combining the context and the focus into a standard spanning tree traversal. The translation operation is graphically depicted in Figure 6.14.

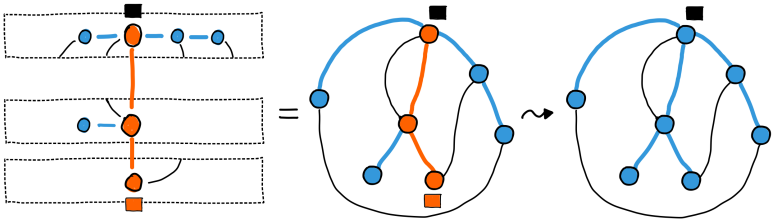


Figure 6.14: Calculating the original tree from a zipper.

As the focus of a zipper is one of the graph's corner only in our implementation, this operation merely inserts the corner in focus at the correct position in the context graph. Whereas this seems like a fairly simple operation, we then also have to translate from the layered structure of the path to the clockwise traversal representation of the overall graph. This translation is as complex for a corner in focus as it would be if the focus contained a larger subgraph. The reorganisation of the data in the path back to a clockwise traversal requires care as the stack property of the non-tree edges has to be maintained at all times in order to preserve planarity of the graph.

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 a the graph straight away.

In case of a root-vtx the zipper focuses on a corner at the root vertex that is not 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*. Remember that these two fields store subtrees in reverse orientations. We therefore use a version of the "chips" operator (recall Section 4.3) which combines a Star Step and a Rats Pets:

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

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 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 and a single Reyal. The case structure in the definition of function toLayers is implemented to match this behaviour. 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 their sibling subtrees in 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 takes it as 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.

The transformation from Reyals to Layers exposes an important intermediate state which is shown in Figure 6.15. 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

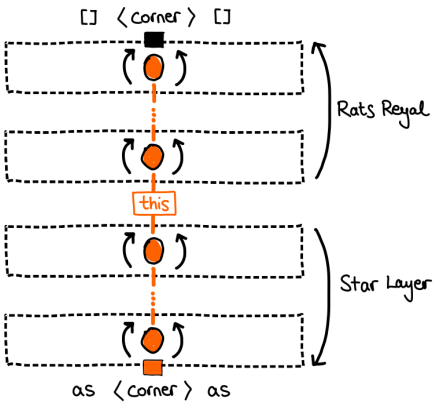


Figure 6.15: Intermediate state of a toLayers operation.

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 connectedness 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 relevant 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 and insert the focus (corner *this*) at the right place.

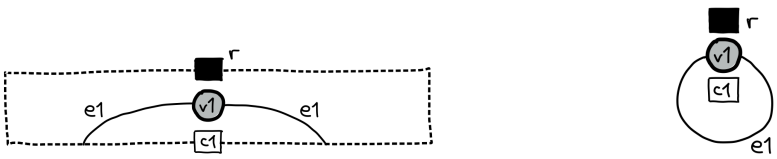
At any layer further towards the root, layersToSteps concatenates the subtrees in a clockwise manner: the right hand side *rats* is added to the beginning of the Star Step and the left hand side *star* is added to its end. For this case, the property that the layers are connect with each other, expressed 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 (topmost) 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.

Putting a lot of work into carefully making planarity an intrinsic property of the type of Graphs now pays back, as the proof of the following property is trivial.

Proposition 6.16. *Calculating the original graph using starToGraph and toLayers from a zipper returns a plane graph.*

Proof. This is immediate from the type of ziToOldRoot. □



(a) The zipper is defined by one Reyal.

(b) The original graph.

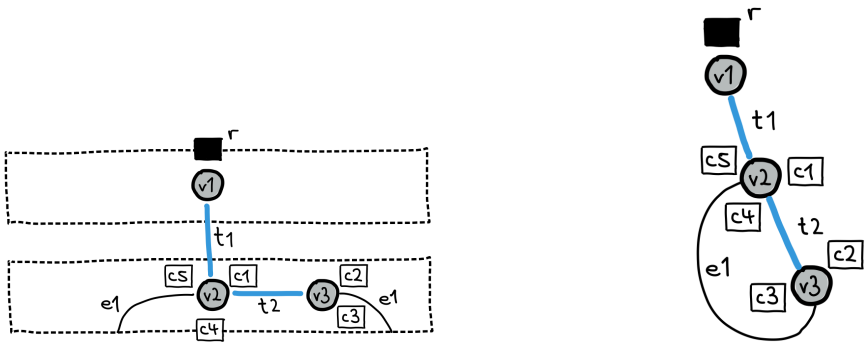
Figure 6.17: Example of recalculating the original graph from a zipper.

Example 6.18. The original tree of the zipper from Example 6.11 is calculated as follows. The zipper structure together with the original tree is depicted in Figure 6.17.

```
root-vtx-oldTree : ziToOldRoot zi-root-vtx
                 ≡ tree r v1 (push e1 ,- corner c1 ,- pop e1 ,- [])
```

Example 6.19. Similarly, the original tree of the zipper from Example 6.12 is calculated by the following function and depicted in Figure 6.20.

```
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 ,- [])
                      ,- [])
```



(a) The zipper's path is encoded as two reyals.

(b) The original graph.

Figure 6.20: Example of recalculating the original graph from a zipper.

The calculation of the original graph from a zipper is one way of translating between the layered representation of a graph and the clockwise traversal. The way we defined the operations involved ensures that the non-tree edges on the stack are accessed in the same order.

We will now present an alternative translation from a zipper to a plane graph. The information in the graph remains the same, but we change the perspective on it.

6.3 Rerooting the Tree

We now specify an alternative operation for computing a plane graph from one of its zippers. This operation changes the perspective onto the graph by moving the root of its spanning tree to the corner in the zipper's focus. The spanning tree remains the same during this operation, but it is *rerooted*. This operation corresponds to choosing a different face of a surface-embedded graph to be its outside face (remember Remark 2.4). The information in the graph embedding remains the same when rerooting, in particular the connectivity information between vertices and edges stays unchanged. Therefore, the rerooting operation constructs an equivalent graph

embedding up to the choice of the outside face, defined by the position of the root of the spanning tree.

The different choice of root means that the graph's spanning tree is traversed in a different order. Computing the new traversal order of the spanning tree alone is not a particularly complex operation. However, the new traversal order of the non-tree edges in the graph is more involved. The stack discipline to store these edges has to be preserved when rerooting, but in the new traversal order we may visit some of them in a different order. Therefore we have to *turn* some of the non-tree edges around. In the new traversal, we may encounter the two ends of an edge in reverse order, thus we have to swap its push and pop operations when calculating the new graph.

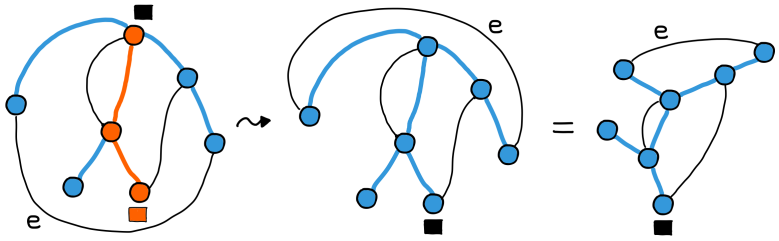


Figure 6.21: An example of moving the root of a graph to the zipper's focus.

An example of rerooting is illustrated in Figure 6.21: the non-tree edge e has to be turned when the root is moved to the zipper's focus as the traversal order now visits its end in reverse order. Therefore e appears at the top of the illustration after the rerooting operation.

Overview of the operation We give a summary of the rerooting algorithm before going into more details about the operations involved. The rerooting operation consists of two steps. First we turn the path of the zipper which is of the form

Rats Reyal ([] ⟨ corner ⟩ []) (*as* ⟨ corner ⟩ *as*)

into a

Star Layer ((reverse *as*) ⟨ corner ⟩ (reverse *as*)) ([] ⟨ corner ⟩ [])

This operation moves the root of the graph to the focus and turns around the path such that the original root becomes the focus of the zipper. The stack *as* at the focus of the original zipper is turned into a stack reverse *as* (where reverse calculates the reverse of a list) at the new focus. The overall operation is similar to the toLayers as it will step through one Reyal at a time, turning it into a Layer, except that here some non-tree edges have to be turned.

From the resulting Star Layer structure we then compute the new graph traversal in a second step. This new traversal will start from the focus of the zipper and move clockwise as before. As all relevant edges are already turned around, this operation is simply reorganising the information, taking the layer structure and calculating the clockwise order, similar to the layersToSteps function in the previous section:

```
layersToNewGraph : (t : ZipTy) → CarryingR t
  → (s : Star Layer t ([] ⟨ corner ⟩ [])) → ConLayers s
  → Graph
```

The argument CarryingR t stores a running total of the traversal of the graph, as the function steps through each Layer of the path. With every function call to layersToNewGraph, the left and right subtrees of the current layer are added to the running total, in the correct order. The base case of this tail-recursive function simply returns the running total which at that point contains the entire traversal. As this function is merely a bookkeeping operation and because of its similarity with the one in Section 6.2, we will not explain any other details here. Instead, we focus on the first operation which turns some edges around and implements the choice of new outside face.

6.3.1 Turning of edges

Before we will dive into the details of the turning operation, we explain some high level intuition. We will do this by considering which non-tree edges have to be turned around at each step when moving the root of the spanning tree to the focus of the zipper.

Recall that every non-tree edge encloses one face of the graph's surface embedding, as discussed in Remark 5.25. The face in which the focus of the zipper is located is enclosed by a number of edges. These edges are exactly those on the stack as in the index of the corner in the focus of the original zipper. In the original graph traversal, these edges have been pushed before reaching the focus, and are popped afterwards. When moving the root to the corner in focus, they are also the edges that have to be turned around. In the new traversal of the spanning tree, their order is reversed: the order of pushing and popping the individual edges is exchanged as well as the order of edges on the stack. After the turning operation, the stack at the original zipper's focus is empty [], and the stack at the original root is the reverse of the stack at the focus, reverse as. In Figure 6.21, this stack contains the edge e only.

Let us have a look at the type of the operation implementing the turning of the non-

tree edges:

```
turnLayers : (t@(es ⟨ n ⟩ zs) : ZipTy)
           → (rz : Rats Reyal ([] ⟨ corner ⟩ []) t)
           → ConReyals rz
           → What n
           → (pa : Partition as es) → (pb : Partition as zs)
           → (ss : Star Layer (turned pa ⟨ n ⟩ turned pb) ([] ⟨ corner ⟩ []))
           → ConLayers ss
           → ConM n rz ss
           → Graph
```

This function turns a Rats Reyal into a Star Layer, by stepping through the path and transforming every Reyal into a Layer individually, starting at the focus of the zipper.

The very first and very last calls to this tail-recursive function are the most straight-forward ones: When being called for the first time, the Reyal structure *rz* contains the entire graph and the Layer structure *ss* is empty. In the very last function call, it is the converse situation: *rz* is empty because all the Reyals have been turned into Layers, and *ss* contains the entire graph. (This code snippet of the left-hand side is written in pseudo-code in form of an Agda comment, as we want to highlight the shape of *t* and *rz* but omit refinements of other arguments.)

```
{-
 turnLayers ([] ⟨ corner ⟩ []) [] conR this pa pb ss conS conM
 -}
```

This function call happens at the position of the original root of the spanning tree. Thus, the ZipTy *t* is located at a corner with empty lists to either side and the Rats Reyal *rz* is empty.

This base case of turnLayers is implemented to immediately call the function layersToNewGraph on the Star Layer *ss*, hereby already turning the layer structure into a tree traversal. This is useful because turnLayers has access to the precise new stack structure at the position of the old root which is required for a call to layersToNewGraph. Therefore, the return type of turnLayers is a Graph.

Any intermediate function call to turnLayers contains both a Rats Reyal and a Star Layer that split the graph into two parts. The function call is "located" at some position *n* on the path, with a number of Reyals located towards the original root of the tree and some Layers located

towards the zipper's focus. Here, the function removes the topmost Reyal, turns it into a Layer and adds it to the sequence of Layers.

Everything is connected During the repeated function calls to turnLayers we have to provide some guarantees about the structures involved: First of all, both the Rats Reyal and the Star Layer have to be connected sequences. Recall that this conditions requires an *edge* between each two neighbouring elements the sequence. Thus, turnLayers expects the proofs of ConReyals rz and a ConLayers ss as inputs. Secondly, in addition to the connectivity of the two structures individually, we have to ensure that they are connected in the *middle*:

```
ConM : (n : Next)
     → (rs : Rats Reyal ([] ⟨ corner ⟩ []) (as ⟨ n ⟩ bs))
     → (ss : Star Layer (as′ ⟨ n ⟩ bs′) ([] ⟨ corner ⟩ []))
     → Set
```

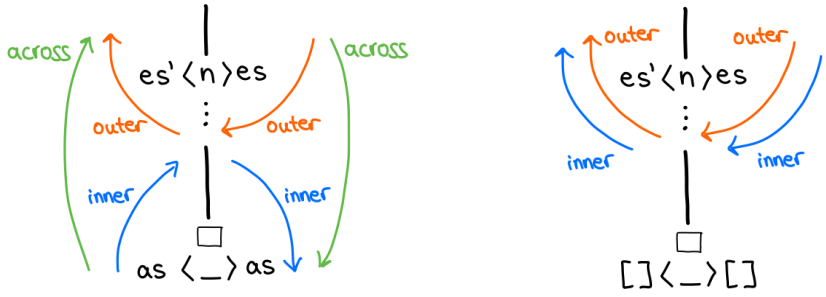
This is similar to the properties of individual Rats Reyal and Star Layer being connected: it expresses the fact that the element n in between the two structures is an edge. Altogether, the connectivity information from the individual structures ensures that the resulting Star Layer of the overall operation is connected, too.

Partitioning the edge stack At every step in the turning operation, we have to decide for the current Reyal which edges have to be turned around and which stay unchanged. Overall, we know that we have to turn all edges in the stack as at the focus of the zipper. For every individual layer though, the calculation is more complex. We introduce use auxiliary structure which segments the stack into three parts and thus classifies the edges stored on it. This auxiliary structure is called a *partition*:

```
record Partition (as es : List E) : Set where
  constructor part
  field inner  : BList E
        across : List E
        outer  : List E
```

(This record contains more fields but we will explain them later.) A Partition spans a certain region of the graph traversal and splits the edges stack into different sections, depending on when they are pushed and popped during the traversal relative to the region. From this

distinction we will be able to calculate which edges need to be turned. A Partition *as es* keeps track of the subtrees on one side of a zipper's path, between the focus (with edge stack *as*) and a position further towards the root (with stack *es*). There are three different categories of non-tree edges that can occur within this region which are illustrated in Figure 6.22a.



(a) Sections of the stack.

(b) Edge stack after turning.

Figure 6.22: Stack sections of a pair of Partition *as es* and Partition *as es*′.

- Some edges are already pushed to the stack before we enter the region and they are not popped until after the traversal has left the region again. Both ends of these edges lie outside of the region in focus, so they span across it. Even though no subtree in the region is attached to these edges, we still have to mention them in a Partition explicitly, as they are located at the bottom of the stack for the entire traversal of this region.
- Some edges are connected to the region with one of their ends and their other end is connected somewhere further towards the root of the tree. In particular, they are not on the stack at the position of the focus, because in the traversal they are popped beforehand or pushed afterwards. These edges are stored in the field outer.
- The third category characterises inner edges. These are edges that are pushed onto the stack on the right hand side of the path or popped on the left hand side. They are located on the top of the stack when the traversal reaches the focus.

In addition to the three stack segments, a Partition comes with rules on how these three segments appear in the two indexing stacks:

$$\begin{aligned} &\mathsf{qpop} \;\; : as \equiv \mathsf{inner} \langle\rangle \mathsf{across} \\ &\mathsf{qpush} : es \equiv \mathsf{outer} \mathbin{+\!\!+} \mathsf{across} \end{aligned}$$

The stack at the focus *as* contains the edges which span across the region as well as the inner edges that were pushed inside the region. The stack *es* contains the across edges as well,

but this time with the outer edges on top of the stack. These latter outer edges will be popped before the traversal reaches the focus.

Remark 6.23. Note that the different kinds of edges never interleave each other when forming the index stacks of a Partition. This is due to planarity: we are not allowed to alternate the push and pop operations of two different edges.

The careful partition of the edge stack contains exactly the right information on the edges that have to be turned around and those which stay the same. We can calculate the state of the stack after the relevant edges have been turned. This is stored in a computed field of the record which is calculated for each record individually, depending on the values of its fields.

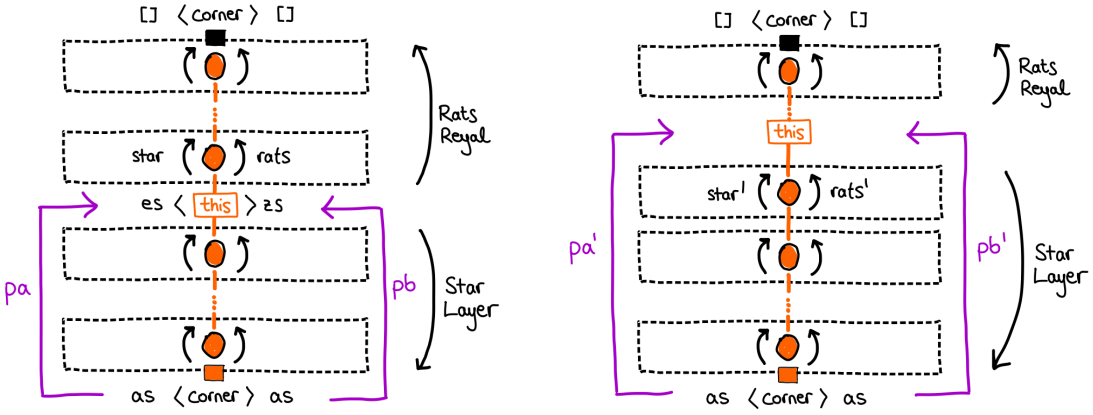
```
turned = outer ++ (rotate inner)
```

Figure 6.22b illustrates the state of the edge stack after the turning operation. The resulting stack at the root end of the region are the outer as well as the inner edges. Because the outer edges were not on the stack at the position of the focus, they are visited in the same order in the new traversal as they were before. The inner edges, on the other hand, have to be turned around. In the original traversal these edges were pushed before the traversal reached the focus (on the right hand side of the path), and popped afterwards (on the left of the path). As the new traversal starts at the focus and visits the left hand side of the path first, the orientation of the inner edges has to change. We notice that the across edges do not appear in the updated edge stack. These edges will eventually also be turned during the overall operation, but not inside this current region of interest. After the rerooting operation, the across edges do not occur in the subregion anymore.

Turning of edges The turning of edges in each individual Reyal is implemented as the application of a Partition to the reyal's subtrees, which are stored as either a Rats Pets or a Star Step. Turning is implemented for the two sides of the path individually:

```
turnStar : {n m : Next}
         → (p : Partition as es) → Star Step (es , n) (es′ , m)
         → Σ (Partition as es′) λ p′ → Star Step (turned p , n) (turned p′ , m)

turnRats : {n m : Next}
         → Rats Pets (es , n) (es′ , m) → (p : Partition as es′)
         → Σ (Partition as es) λ p′ → Rats Pets (turned p′ , n) (turned p , m)
```

These operations turn the relevant edges in the Rats Pets as well as updating the Partition, as the stacks in the partition change with each turning operation. As we work our way through the Reyals towards the old root with turnLayers, more and more edges are turned the right way around, therefore the updated Partition includes less and less non-trivial information. Figure 6.24 shows the development of the edge stacks and partitions as one Reyal is turned into a Layer.



(a) Before turning edges of *star* and *rats*.

(b) Turning changes the lists to *star′* and *rats′*.

Figure 6.24: Schema of the intermediate state during a call to turnLayers.

Back to the implementation of the function turnLayers: in the most general case, the operation uses a Partition and functions to turn the relevant elements:

```
turnLayers t (rz -, r@(reyal v this star rats)) cR this′ pa pb ss cS cM
  = let (pa′ , star′) = turnStar pa star
        (pb′ , rats′) = turnRats rats pb
        (cR′ , cM′)   = connect rz r ss cR cM cS
        s             = layer v rats′ this′ star′
        (c , c′)      = cM′ s
    in turnLayers _ rz cR′ this pa′ pb′ (s ,- ss) c′ c
```

The operation takes the topmost Reyal *r* and the two partitions *pa* and *pb*, as shown in Figure 6.24a. It then turns the relevant edges to either side of *r* which results in the updated fields *rats′* and *star′* as well as updated partitions *pa′* and *pb′*. These new fields are used to form a new Layer called *s* which is added to the Star Layer in the recursive function call. This is illustrated in Figure 6.24b. From the connectivity property of the original Rats Reyal and Star Layer, together with the connectivity property in the middle, we get the corresponding

properties for the updated structures. Finally, the function issues a recursive call with the updated arguments.

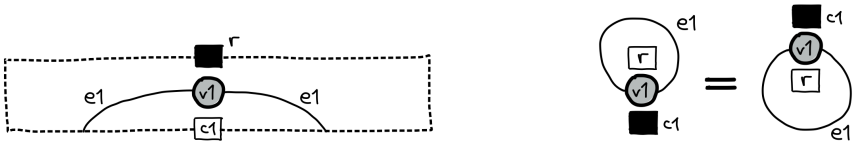
Putting it all together Overall, we can now define the operation ziToNewRoot which takes an arbitrary zipper structure for a graph, and reroots it to its focus, returning a plane graph again:

```
ziToNewRoot : {as : List E} → Zipper as → Graph
ziToNewRoot (empty r) = empty r
ziToNewRoot (vertex r v) = vertex r v
ziToNewRoot (root r v st) = tree r v st

ziToNewRoot { as } (root-vtx r@(reyal v this star rats) c)
  = turnLayers {as} (as ⟨ corner ⟩ as) ([] -, r) _ c
                (part [] as [] refl refl)
                (part [] as [] refl refl) [] _ _

ziToNewRoot { as } (reyals r rz cR c)
  = turnLayers {as} (as ⟨ corner ⟩ as) (catRats ([] -, r) rz) (conr r rz cR) c
                (part [] as [] refl refl)
                (part [] as [] refl refl) [] ⟨⟩ (conm r rz)
```

Initially, turnLayers is called with a trivial Partition and an empty Star Layer. No edges have been turned yet, so all edges on the stack at the zipper's focus are left to turn. The corresponding partition is (part [] as [] refl refl) where the only non-empty field contains across edges. The initial segment of path is empty (it only includes the focus itself), therefore all edges on the stack encircle it. As we work our way up the path, the Partitions may contain edges in their other fields, too.

We have taken great care in setting up the data type of graphs and zippers to include the planarity property. During the rerooting operation we used the stack discipline as a helpful guidance, but, more importantly, it forced us to also preserve planarity. Therefore, the proof of the following theorem is now for free.

Proposition 6.25. *Rerooting the zipper of a plane Graph to its focus results in a plane graph.*



(a) The zipper is defined by one Reyal. (b) The re-rooted graph.

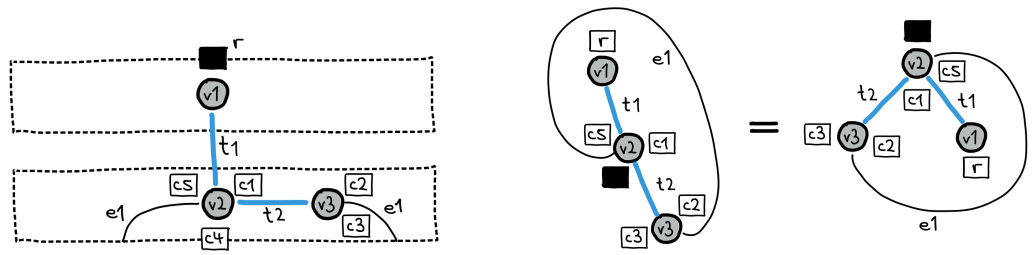
Figure 6.26: Example of rerooting a zipper to the corner in its focus.

Example 6.27. When calculating the re-rooted tree from the zipper in Example 6.11 we get the following graph, depicted in Figure 6.26. Note that this graph is almost the same than in Example 6.18 where we computed the original tree of the zipper, except that the two corners are exchanged, with *c1* acting as the new root.

```
root-vtx-newTree : ziToNewRoot zi-root-vtx
                 ≡ tree c1 v1 (push e1 ,- corner r ,- pop e1 ,- [])
```

Example 6.28. Here is the rerooting of the zipper from Example 6.19 which uses the Reyals constructor. Figure 6.29 provides the corresponding illustration. Observe how the shape of the spanning tree changes from a single spine to two sibling subtrees when rerooting.

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



(a) The zipper focusses on *c4* with two Reyals. (b) The graph re-rooted to the focus *c4*.

Figure 6.29: Example of recalculating the original graph from a zipper.

6.4 Rewriting

As we have discussed before (e.g. in Section 1.3), the operation of graph rewriting consists of two steps: Firstly, identifying the subgraph which is targeted by the rewrite rule. This amounts to computing a match of the subgraph inside a larger graph. The second step consists of replacing the subgraph by the right hand side of the given rewrite rule. In the DPO rewriting framework, these two steps are defined as certain graph morphisms. To define rewriting for the Agda implementation of plane Graphs, we need similar operations and structures. The advantage of the intrinsically plane graphs and operations in the implementation is that we get the preservation of planarity during a rewriting step for free.

The matching of a smaller graph inside a larger one is represented by an instance of the graph's zipper, focussing on that particular subgraph. Fortunately, a lot of the work on the data type of plane has gone into defining plane graphs with a *focus*. In the current implementation, the focus of a zipper is an individual corner. The data we have about this corner is its location inside the overall graph as well as the state of the edge stack at its position in the graph traversal. Rewriting a corner with a subgraph amounts to inserting the graph into the context of the corner. Therefore, the rewriting operation we present here looks more like an insertion operation. But with zippers focussing on corners only, this makes sense. The operation amounts to placing the subgraph into an enclosed face of the context graph and connecting certain edges to vertices, without introducing any crossings.

As a corner itself has no influence of the edge stack, we can rewrite it by a graph which take an empty stack to an empty stack. The rewrite operation inserts the new graph at the same vertex where the corner in focus is located, positioned right next to this corner:

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

We implement this function by matching on the shape of both the zipper and the graph. As an example, we look at the case where the zipper is focussed at the root corner and the graph we are inserting is a tree.

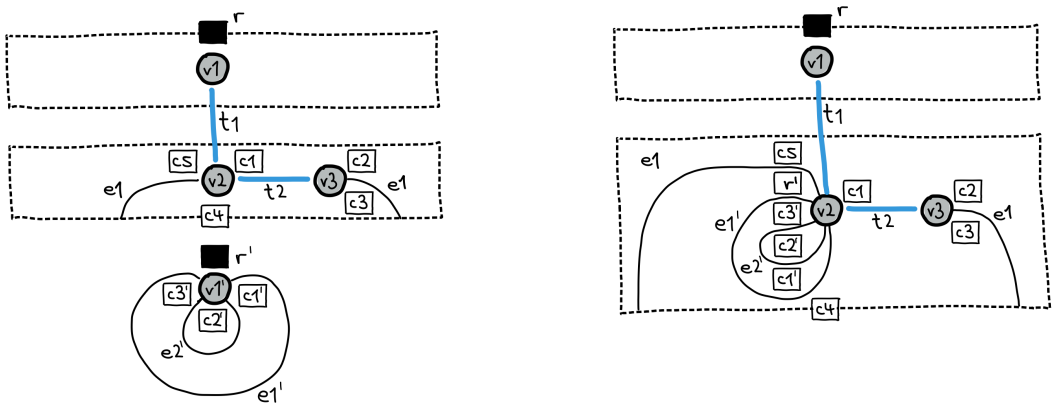
```
rewriteFocus (root r v ss) (tree r′ v′ ss′) = root r v (concat ss′ (corner r′ ,- ss))
```

In this case, we can insert the subtrees ss' of the graph at the beginning of the traversal around the root vertex v in the zipper which is done by concatenating the two Star Steps of subtrees. The result is a zipper which is still focussed at its root corner r, but now the context graph includes both sets of subtrees ss' and ss, concatenated via the corner r'.

Whenever the zipper is focussed at the root corner, its index is the empty list of edges, making the inserting of a list of subtrees straight-forward. In the case where the focus lies further inside the tree, the stack *cs* may be non-empty. The graph we insert does not interact with the edges of *cs*, but to be able to insert it at the position of the focus, we have to perform a type cast. To make the types match, we observe that we can lift any Star Step taking a stack *as* to *bs* to taking the stacks *concatenated* with a list *cs*. The Star Step only acts on the top of the stack and never uses any edges from the segment *cs*. This lifting amounts to placing the subgraph into a certain face of another graph which is enclosed by exactly the edges in *cs*.

```
castStep : { as bs cs : List E } { c d : Next }
         → Step (as , c) (bs , d)
         → Step (concat as cs , c) (concat bs cs , d)
```

We will omit the details of the other cases of the implementation of rewriteFocus and refer to the Agda files, because the idea is the same as in the case shown: the input graph is inserted into the zipper structure at the position of the corner in focus. 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 6.30: Example of inserting a graph at the focus of a zipper.

Example 6.31. As an example, we use the zipper from Example 6.12 and insert the graph with two self-loops from Example 5.27 at the position of the focus. The graphs involved and the result of the operation are shown in Figure 6.30.

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)
                           (ex-bouquet 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
```

Challenge 1: Rewriting larger graphs So far, our implementation covers the case of zippers focussing on a single corner. For more general graph rewriting, we are interested in focussing on and replacing larger subgraphs. We make the following observation: As the Reyal structure starts at the focus, we always have easy access to the region of the graph embedding that immediately surrounds the focus. With every Reyal, this regions grows, but, when growing from the focus "upwards", it always forms a disc-like part of the surface. Therefore it can be removed as a whole without invalidating the planarity property by removing the corresponding number of Reyals from the path. Overall, even with a Zipper type that focuses on a corner only, we now have a strategy for addressing a disc-like region of the graph's surface embedding. This region can be rewritten by attaching a number of Reyals to the front of the remaining path. The indexing type ZipTy ensures that the graph inserted has the same effect on the edge stack than the previous sequence of Reyals.

Challenge 2: Finding a graph match So far, this description of rewriting assumes a match already by providing a certain zipper focussed on the root where a rewrite rule will be applied. For implementing a full rewriter we will need an operation which matches the left hand side of a rewrite rule to a subgraph inside a Graph. In particular, this operation has to compute a zipper structure in a way 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 5.3). Additionally, the two subgraphs in a rewrite rule may not have the same spanning tree, and thus the paths to the focus would be very different. 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 non-tree edge to be part of the spanning tree and recomputes which edge will then has to be removed from the tree in order to keep its non-cyclic property. Crucially, any function that manipulating a Graph, whether by changing the spanning tree, or replacing a subgraph, will always be guaranteed to preserve the planarity.

6.5 Future Work

The implementation of graphs and their zippers in Agda motivate a number of ideas which would be interesting to develop further:

Implementation of a graph rewriter The operations described in this Part II are implemented towards the goal of a full graph rewriter in Agda. As discussed in the previous section, at the current stage rewriting can be realised by inserting a graph at the position of the zipper's focus or by replacing the part of the path nearest to the focus. A more general framework would require needs additional operations to turn a graph's spanning tree into a certain shape to allow for rewriting one of its subgraphs. The aim for future work is a program which takes a graph and a rewrite rule and computes the rewrite automatically.

Equivalence classes of spanning trees We have made two choices when implementing graphs: Firstly, we have picked one of the faces of a graph embedding to be its *outside* face, as we have discussed in Remark 2.4. Secondly, we choose a spanning and a root for a graph. In future work, we would like to consider equivalence classes of graph embeddings, independent of the choice of outside face or spanning tree. The refocussing operation presented in Section 6.3 is one step towards this goal as it calculates a different perspective on the same structure by choosing a different root for the same spanning tree.

More general structures than graphs Programming with graphs with a focus motivates to think about contextual structures more generally. Zippers provide a convenient framework to talk about substructures inside larger structures, and with operations such as refocussing we can change the perspective on the structure in a principled way. Contextual programming is not only interesting for graphs laid out on a surface, but also for frameworks in which the notion of context is more abstract, e.g. the free variables for a lambda term in focus. We propose a project towards contextual programming for a larger class of data types in Chapter 7.

Chapter 7

Contextual Programming

This chapter contains a short outline of a proposal for working on a framework implementing programming with data structures with a context generically, based on our work on plane graphs and their rewriting theories.

Data types with a focus Specifying the rewriting theory for plane graphs motivates to think about the notion of focussing for a bigger class of inductive data types. Focussing defines the separation of a substructure from its context. It allows operations to act on the substructure alone without affecting the context. We can distinguish between the part to which the operation can be applied *locally* and the part which is invariant under it. To ensure that a subgraph can still be embedded into its context after applying a function to it, we need to store the interface between focus and context and preserve it whenever the focussed structure changes. In the context of plane graphs, focussing is used to calculate the match of a subgraph inside a larger graph. This match describes where to find the subtree. The local operation that we apply to the subgraph is a rewrite rule. In our categorical rewriting framework, the separation between substructure and context is defined as certain spans in the DPO diagram as discussed in Section 1.3, and the guarantee that the result of a rewrite rule fits into the hole again by the commutation of both squares involved.

Contextual information can come in different shapes. Typical examples include input arguments to a function which have to be supplied by the environment for the function to compute, or some memory resources that have to be available before a computation can run. For this proposal, we focus on a *spatial* notion of context which consists of a series of constructors of a data type. A subterm in focus is located "behind" these constructors, somewhere further inside the overall term. In a traversal of the term, once we have passed the constructors

specified in the context, we find the subterm in focus. As an example, a sublist of a list can be found by passing a certain number of list elements, i.e. passing a number of applications of the $cons$ constructors. The spatial location of a subterm is interesting both when manipulating the actual topology of a term, but also in cases where we have to keep track of the order of resources in the context. This is the case when describing terms in a metaprogramming setting: the location of a certain subterm determines which resources can be used to infer information about it (for example typing information).

We propose to develop the notion of structures with a focus for a certain general class of inductive data types: those that are described by containers.

Containers describe data types Containers have been developed as a generic representation of a class of data types. They provide a syntax for storing data in a structured way. Operations that act on the structure of a whole class of data types can be defined *generically* on containers and then be instantiated to any type in the class.

Definition 7.1. The data of a container has two parts: a set of shapes, and, for each shape, a set of positions where data can be stored:

```
record Container : Set₁ where
  constructor _◁_
  field Sh  : Set
        Pos : Sh → Set
```

Shapes determine the structure of the data type which the container encodes. Depending on the shape, the positions then define the places where data can be stored. Containers are closed under structural operations such as pairing, products, and fixpoints. We will define these operations explicitly on our notion of containers later in this chapter. Containers represent the class of strictly positive data types [1] which arise from forming fixpoints of polynomial functors. Initial algebras for containers always exists (because they represent strictly positive functors), and correspond to W-types which themselves can describe any tree-shaped type in an intutionistic type theory. We can use trees as the intuition for illustration containers schematically, such as in Figure 7.2 for the definition of a container with annotated shape and positions.

Containers typically come with an interpretation operation which extends the notion of container to an endofunctor on types.

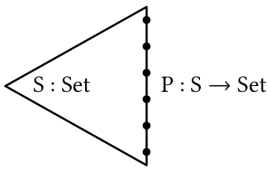


Figure 7.2: Schema of a container.

Definition 7.3. The *extension of a container* is defined as the following endofunctor on sets:

$$[\![\_]\!]\mathsf{c} : \mathsf{Container} \to \mathsf{Set} \to \mathsf{Set}$$
$$[\![\ Sh \lhd Pos\ ]\!]\mathsf{c}\ X = \Sigma\ Sh\ \lambda\ sh \to Pos\ sh \to X$$

Concretely, this operation implements the instantiation of a container to a particular data type. Given an element type X, the extension of a container at X consists of a shape $sh : Sh$, and a function from positions $Pos\ sh$ of that shape to elements of X.

Example 7.4. An example of a container is a description of the type of lists:

$$\mathsf{ListC} : \mathsf{Container}$$
$$\mathsf{ListC} = \mathsf{Nat} \lhd \mathsf{Fin}$$

$$\mathsf{List} : (A : \mathsf{Set}) \to \mathsf{Set}$$
$$\mathsf{List}\ A = [\![\ \mathsf{ListC}\ ]\!]\mathsf{c}\ A$$

Figure 7.5: Lists as an instance of container.

The list container ListC provides a specification of the list type, with natural numbers Nat as the shape (representing the list's length), and a finite set of the corresponding size as the positions. We get the actual data type of lists of element type A by taking the extension of the list container at A.

We would like to develop a calculus of programming with context-aware structures, using containers as the representation of the class of inductive data types. This is related to derivatives of data types [70, 2, 1] and also to the framework of context logic [17] to which we aim to establish a formal connection.

A comonads to describe context-awareness After developing suitable definitions for the partition of any inductive structure into a context and a focus, we aim to show that annotating every element inside the structure with its context admits a context comonad.

Comonads provide a potential framework for implementing contextual information [92]. They stand in contrast to *monads* which are a construction to add effects to pure functions in

functional programming [76]. An effect describes an additional output of a pure function, such as a string that can be written to an input-output system. On the contrary, a *coeffect* describes an additional *input* to a pure function. This is precisely the information a context can provide, some information that is known before a function computes. Similar to the use of monads for effects, these coeffects can be defined in the structure of a comonad.

Our aim is to establish a spatial notion of context which admits this comonad structure. We believe that programming with data types comonadically has a lot of potential with numerous applications beyond the instance of plane graphs and their rewriting.

Conclusion

We finish by highlighting some of the aspects that we have explored in this work.

Combinatorial presentations for graphs embedded on surfaces typically look quite different from the encoding of graphs themselves. Adding topological information to an other purely relational data structure requires considerable effort in the data structures involved. In this work, we have investigated two of those representations.

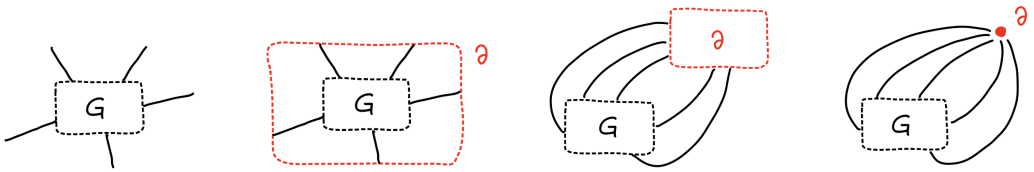


Figure 8.1: Recall: Introduction of a boundary vertex to represent the outside face of a graph.

In Part I we have used rotation systems on top of a graph representation to encode their embedding. We have introduced auxiliary structures to represent the boundary of a graph for this notion to be appropriate to represent graph embeddings (see Figure 8.1). In this notion, representing graphs on different surfaces is relatively straight-forward as all the embedding information is encoded in the rotation system and can be extracted from there.

(a) A graph, represented by its spanning tree.

(b) The layer structure of the graph's zipper.

Figure 8.2: Recall: Representation of a graph and its zipper in the Agda implementation.

In Part II we have used an inductive representation of graphs by using their spanning tree as a skeleton. For the implementation of graphs in a functional programming language, this was a suitable structure as we are able to manipulate graphs in a straight-forward way

using recursive functions (see Figure 8.2a for an example). The embedding property of the graphs is entirely encoded by the representation of the non-tree edges. This encoding is fairly straight-forward with a stack for the case of plane graphs. In theory, the framework is easily extendable to higher genus graphs, but in practice it is not known which data type can encode the structure non-tree edges in this case.

Both representations of graph embeddings highlight the importance of local rewriting. When working with an equational theory of diagrammatic languages, being able to separate the target of a rewrite rule from its context is crucial. It guarantees that the rewrite is acting on a certain subgraph only, leaving the context unchanged. This behaviour is formalised as double-pushout rewriting in the categorical language, described by the native composition operation in a diagram operad (e.g. see Figure 8.3), and as an instance of a zipper in the Agda implementation of graphs (see Figure 8.2b). We have additionally presented an opposite operation to rewriting by defining a framework for pattern matching for graphs.

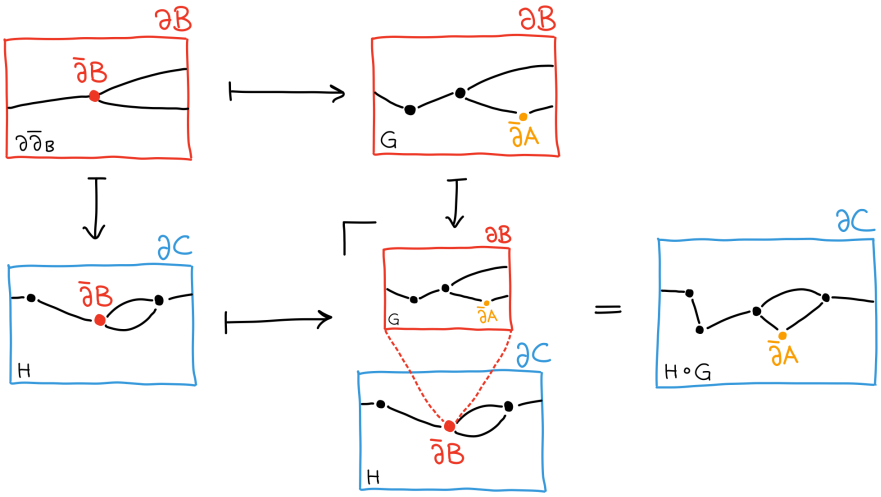


Figure 8.3: Recall: An example of operad composition of $G : \bar{\partial} A \vdash \partial B$ and $H : \bar{\partial} B \vdash \partial C$.

All of the concepts which we introduced and studied in this work motivate to think about contextual programming more generally which is not only interesting in connection with surface-embedded graphs but for a large number of applications in programming languages and process theories.

List of Figures

Index of Definitions

Bibliography

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 2005.

[2] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2701, 2003.

[3] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 2013.

[4] Marie Albenque and Dominique Poulalhon. A generic method for bijections between blossoming trees and planar maps. *Electronic Journal of Combinatorics*, 22, 2015.

[5] Malin Altenmüller and Ross Duncan. A category of surface-embedded graphs. In *Electronic Proceedings in Theoretical Computer Science, EPTCS*, volume 380, 2023.

[6] John C. Baez and Kenny Courser. Structured cospans. *Theory and Applications of Categories*, 35, 2020.

[7] John C. Baez, Kenny Courser, and Christina Vasilakopoulou. Structured versus decorated cospans. *Compositionality*, 4, 2022.

[8] John C. Baez and Jason Erbele. Categories in control. *Theory and Applications of Categories*, 30, 2015.

[9] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: An online proof assistant for higher-dimensional rewriting. *Logical Methods in Computer Science*, 14, 2018.

[10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. 2004.

[11] Richard S. Bird. On building cyclic and shared structures in haskell. *Formal Aspects of Computing*, 24, 2012.

[12] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory ii: Rewriting with symmetric monoidal structure. *Mathematical Structures in Computer Science*, 32, 2022.

[13] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. Rewriting modulo symmetric monoidal structure. pages 710–719. Association for Computing Machinery (ACM), 10 2016.

[14] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. Confluence of graph rewriting with interfaces. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10201 LNCS, 2017.

[15] Filippo Bonchi, Joshua Holland, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Diagrammatic algebra: From linear to concurrent systems. *Proceedings of the ACM on Programming Languages*, 3, 2019.

[16] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Full abstraction for signal flow graphs. *ACM SIGPLAN Notices*, 50, 2015.

[17] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 2005.

[18] Davide Castelnovo, Fabio Gadducci, and Marino Miculan. A new criterion for m, n -adhesivity, with an application to hierarchical graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 13242 LNCS, 2022.

[19] Bob Coecke and Ross Duncan. Interacting quantum observables: Categorical algebra and diagrammatics. *New Journal of Physics*, 13, 2011.

[20] Bob Coecke, Ross Duncan, Aleks Kissinger, and Quanlong Wang. Generalised compositional theories and diagrammatic reasoning. 2015.

[21] Bob Coecke and Aleks Kissinger. Picturing quantum processes: A first course on quantum theory and diagrammatic reasoning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10871 LNAI, 2018.

[22] Bob Coecke, Mehrnoosh Sadrzadeh, and Stephen Clark. Mathematical foundations for a compositional distributional model of meaning. 2010.

[23] Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru, and Jamie Vicary. homotopy.io: a proof assistant for finitely-presented globular n-categories. 2 2024.

[24] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the qubit routing problem. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 135, 2019.

[25] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. *ACM SIGPLAN Notices*, 35, 2000.

[26] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5836 LNCS, 2011.

[27] Robert Dawson and Robert Paré. General associativity and general composition for double categories. *Cahiers de topologie et géométrie différentielle catégoriques*, 34:57–79, 1993.

[28] Lucas Dixon and Ross Duncan. Graphical reasoning in compact closed categories for quantum computation. *Annals of Mathematics and Artificial Intelligence*, 56, 2009.

[29] Lucas Dixon, Ross Duncan, and Aleks Kissinger. Open graphs and computational reasoning. *Electronic Proceedings in Theoretical Computer Science*, 26, 2010.

[30] Lucas Dixon and Aleks Kissinger. Open-graphs and monoidal theories. In *Mathematical Structures in Computer Science*, volume 23, 2013.

[31] Ross Duncan. *Types for Quantum Computing*. PhD thesis, University of Oxford, 2006.

[32] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6, 1994.

[33] Matt Earnshaw, James Hefford, and Mario Román. The produoidal algebra of process decomposition. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 288, 2024.

[34] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. 2006.

[35] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11, 2001.

[36] Wenfei Fan. Graph pattern matching revised for social network analysis. In *ACM International Conference Proceeding Series*, 2012.

[37] Brendan Fong. Decorated cospans. 2 2015.

[38] Ezra Getzler and John .D.S. Jones. Operads, homotopy algebra and iterated integrals for double loop spaces. *Arxiv preprint hep-th/9403055*, 1994.

[39] Neil Ghani, Makoto Hamana, Tarmo Uustalu, and Varmo Vene. Representing cyclic structures as nested datatypes. *Nihon Sofutowea Kagakukai Taikai Koen Ronbunshu CDROM*, 2006, 2006.

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

[41] Georges Gonthier. *The Four Colour Theorem: Engineering of a Formal Proof*, pages 333–333. 2008.

[42] Jonathan Gross and Thomas Tucker. *Topological Graph Theory*. Courier Corporation, 2001.

[43] Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2010.

[44] Annegret Habel and Detlef Plump. $\mathcal{M},\mathcal{N}$ *-Adhesive Transformation Systems*, pages 218–233. 2012.

[45] Amar Hadzihasanovic and Diana Kessler. Data structures for topologically sound higher-dimensional diagram rewriting. In *Electronic Proceedings in Theoretical Computer Science, EPTCS*, volume 380, 2023.

[46] Makoto Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6, 2010.

[47] A. Hatcher and W. Thurston. A presentation for the mapping class group of a closed orientable surface. *Topology*, 19, 1980.

[48] Jules Hedges, Evguenia Shprits, Viktor Winschel, and Philipp Zahn. Compositionality and string diagrams for game theory. 2016.

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

[50] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 10 1974.

[51] Gerard Huet. The zipper, 1989.

[52] John Robert Edmonds Jr. *A Combinatorial Representation for Oriented Polyhedral Surfaces*. PhD thesis, University of Maryland, 1960.

[53] Dimitri Kartsaklis and Mehrnoosh Sadrzadeh. A frobenius model of information structure in categorical compositional distributional semantics. In *MoL 2015 - 14th Meeting on the Mathematics of Language, Proceedings*, 2015.

[54] Yasuyuki Kawahigashi. Conformal field theory, tensor categories and operator algebras. *Journal of Physics A: Mathematical and Theoretical*, 48, 2015.

[55] Gregory Maxwell Kelly, M Laplaza, Geoffrey Lewis, and Saunders Mac Lane. *Coherence in Categories*. Springer, 1972.

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

[57] Aleks Kissinger and Vladimir Zamdzhiev. Equational reasoning with context-free families of string diagrams. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9151, 2015.

[58] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9195, 2015.

[59] Alex Kissinger. Chyp: An interactive theorem prover for string diagrams, 2023.

[60] Wen Kokke, Jeremy G. Siek, and Philip Wadler. Programming language foundations in agda. *Science of Computer Programming*, 194, 2020.

[61] Stephen Lack. Composing props. *Theory and Applications of Categories*, 13, 2004.

[62] Stephen Lack and Pawel Sobociński. Adhesive categories. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2987, 2004.

[63] Stephen Lack and Pawel Sobociński. Adhesive and quasiadhesive categories. In *RAIRO - Theoretical Informatics and Applications*, volume 39, 2005.

[64] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5. Springer New York, 1978.

[65] Tom Leinster. *Higher Operads, Higher Categories*. 2004.

[66] Tom Leinster. *Basic category theory*. 2014.

[67] Saunders MacLane. Categorical algebra. *Bulletin of the American Mathematical Society*, 71:40 – 106, 1965.

[68] Per Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104, 1982.

[69] J. P. May. *The Geometry of Iterated Loop Spaces*, volume 271. Springer Berlin Heidelberg, 1972.

[70] Conor McBride. The derivative of a regular type is its type of one-hole contexts. Technical report, 2001.

[71] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14:69–111, 1 2004.

[72] K. Mehlhorn and P. Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica (New York)*, 16, 1996.

[73] Paul André Melliès. Local states in string diagrams. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8560 LNCS, 2014.

[74] Paul-André Melliès and Noam Zeilberger. The categorical contours of the chomsky-schützenberger representation theorem. 12 2023.

[75] Stefan Milius, Lutz Schröder, and Thorsten Wißmann. Regular behaviours with names: On rational fixpoints of endofunctors on nominal sets. *Applied Categorical Structures*, 24, 2016.

[76] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93, 1991.

[77] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. 2001.

[78] Andrey Mokhov. Algebraic graphs with class (functional pearl). *ACM SIGPLAN Notices*, 52, 2017.

[79] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

[80] Ulf Norell. Dependently typed programming in agda. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5832 LNCS, 2009.

[81] Dusko Pavlovic. Monoidal computer i: Basic computability by string diagrams. *Information and Computation*, 226, 2013.

[82] Dusko Pavlovic. Monoidal computer ii: Normal complexity by string diagrams. 2014.

[83] Maciej Piróg and Nicolas Wu. String diagrams for free monads (functional pearl). *ACM SIGPLAN Notices*, 51, 2016.

[84] John Power and Hiroshi Watanabe. Distributivity for a monad and a comonad. In *Electronic Notes in Theoretical Computer Science*, volume 19, 1999.

[85] Mario Román. Open diagrams via coend calculus. In *Electronic Proceedings in Theoretical Computer Science, EPTCS*, volume 333, 2021.

[86] Mario Román. *Monoidal Context Theory*. PhD thesis, Tallinn University of Technology, 2023.

[87] Peter Selinger. A survey of graphical languages for monoidal categories, 2011.

[88] Paweł Sobociński. Nets, relations and linking diagrams. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8089 LNCS, 2013.

[89] Paweł Sobociński, Paul W. Wilson, and Fabio Zanasi. Cartographer: A tool for string diagrammatic reasoning. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 139, 2019.

[90] David I. Spivak. The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits. 5 2013.

[91] Aaron Stump. *Verified Functional Programming in Agda*. 2016.

[92] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electronic Notes in Theoretical Computer Science*, 203, 2008.

[93] Rik van Toor. *MMH: High-level programming with the Mu-Mu-Tilde-calculus*. PhD thesis, Universiteit Utrecht, 2020.

[94] Hassler Whitney. *On Regular Closed Curves in the Plane*. 1992.

[95] Fabio Zanasi. Rewriting in free hypergraph categories. *Electronic Proceedings in Theoretical Computer Science*, 263:16–30, 12 2017.

[96] Noam Zeilberger. Linear lambda terms as invariants of rooted trivalent maps. *Journal of Functional Programming*, 26, 11 2016.

[97] Noam Zeilberger and Alain Giorgetti. A correspondence between rooted planar maps and normal planar lambda terms. *Logical Methods in Computer Science*, 11, 9 2015.