Compositional Graph Pattern Matching

Malin Altenmüller
University of Edinburgh, UK *
malin.altenmuller@ed.ac.uk

Ross Duncan
Quantinuum, Japan
ross.duncan@quantinuum.com

Graph rewriting is the predominant technique used for reasoning with monoidal categories. Equivalences between morphisms are expressed as string diagrammatic equations and applied in form of graph rewrite rules. We study a class of graph categories in which substitution of subgraphs is defined as an instance of double-pushout rewriting and compare it to notions of substitution and pattern matching in term calculi like Standard ML. With substitution as primary operation, we show how to organise graphs into a multicategory with substitution as the composition operation. We then define the opposite construction to the graph multicategory and argue that it resembles a co-multicategory of graph *patterns*. As both graphs and patterns live in the same underlying category, we can study their interaction. This leads to a language of pattern matching for adhesive categories of graphs which we propose as a new construct for programming and reasoning with graphical calculi.

1 Introduction

Graphical languages are a common tool for the representation of computational processes, both in theoretical modelling such as Petri nets and in concrete descriptions of hardware such as circuits calculi. Of particular interest to us are string diagrams [36], a graphical syntax for morphisms in monoidal categories. String diagrams play a major role in the formalisation of a wide range of applications across theoretical computer science and physics, including concurrency theory [37, 8], functional programming [34, 35], economics [19], quantum computing [10, 11, 12], to name only a few. String diagrams are not useful as an explanatory tool, they are also mathematically rigid as equal diagrams represent equal morphisms in the monoidal theory [30]. Reasoning with diagrammatic languages consists of applying these diagrammatic equalities as rewrite rules.

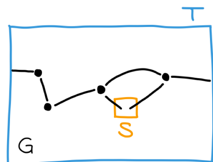
The use of diagrammatic languages has proven effective not only for pen and paper calculation, but also as the basis for automated systems [25, 3, 13]. This work contributes to transferring well known programming techniques to the realm of diagrammatic languages. We observe the connection between the substitution of subgraphs to the substitution of terms for variables in functional programming languages such as Standard ML. We then establish a similar connection with the notion of pattern matching.

We start from the well established formalism of double-pushout rewriting for implementing graph rewriting [15]. This two-step procedure specifies the removal of a subgraph from a larger graph followed by the insertion of a different subgraph into the hole. The double-pushout construction ensures the soundness of the rewrite by enforcing interface equality between subgraphs. Furthermore, it guarantees locality, ensuring that the rewriting procedure does not affect the context graph. Using the notion of boundary-preserving substitution as primitive operation, we define graphs as morphisms of a multicategory where objects represent the type of graph *variables* and morphisms encode graphs containing variables. An example graph is illustrated in Figure 1a. Composition corresponds to the substitution of a subgraph for a variable, implemented as one instance of DPO rewriting.

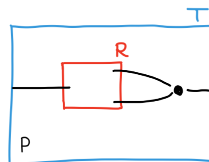
*Malin Altenmüller is supported by EPSRC grant no. EP/X025551/1.

Submitted to:
ACT 2026

© M.Altenmüller & R.Duncan
This work is licensed under the
Creative Commons Attribution License.



(a) Example of a graph with one variable $G : S \to T$.



(b) Example of a pattern with one variable $P : T \to R$.

The same principle motivates the study of a co-multicategory in which morphisms are graph *patterns*. Patterns specify the layout of pattern variables inside a graph structure, with composition expressing pattern refinement. An example of a pattern is shown in Figure 1b. As both graphs and patterns live in the same underlying category of graphs, their interaction can be expressed by graph morphisms. This leads to a notion of *pattern matching*, similar to the corresponding notion in functional programming.

Pattern Matching In functional programming languages, the use of pattern matching is a standard technique to define functions on algebraic data types. Having been introduced as a strategy for proving meta-theoretic properties of programs [9, 33], pattern matching for function arguments was first implemented as a programming construct in Standard ML [18]. By a *pattern* we understand an expression containing variables, defining the shape of a term polymorphically, for example on the left-hand side of a function definition. When the function is called with a concrete term, the term is matched against the pattern and the pattern variables are instantiated with concrete terms. An example of pattern matching on lists in Haskell is explained in Appendix A. Our notion of pattern matching for graphs describes the same operation on different data: given a graph pattern containing variables, we match a concrete graph against it by instantiating the pattern variables with concrete subgraphs. In term languages, substitution and pattern matching are complementary operations. We transfer this correspondence to graphs: with substitution of graphs being represented by DPO rewriting, we will implement pattern matching by constructing the corresponding pushout *complements*.

Remark 1. Our notion is related to but not the same as *subgraph matching*, specifying the position of a small graph inside a larger one. In our framework, patterns and graphs have the same interface structure and the pattern matching operation maps substructures of the graph onto variables inside the pattern.

Graph Category Our construction addresses categories of graphs with the following two properties: rewriting can be implemented by the double-pushout construction and the notion of boundary of a graph is well defined. We achieve the latter by working with *product categories* [27] (PROs) which describe processes as functions of their boundaries, and the former by considering categories with *adhesive* properties [29, 28] in which DPO rewriting is well defined. In this general framework, the definitions of substitution and pattern matching are pleasingly simple. Certain instances of graph categories may require a more careful treatment — we will demonstrate one such example — but this is merely due to the complexity of the graph category instead of the construction we present.

2 Substitution for Graphs

Equational theories for monoidal categories typically consist of a set of rewrite rules of subterms. Whenever a larger term contains a subterm which is the subject of a rewrite rule, we can replace it by the right-hand side subterm. This operation can be implemented directly on string diagrams. The standard notion of graph rewriting in the context of diagrammatic languages is *double-pushout rewriting* [15].

2.1 Double-Pushout Rewriting

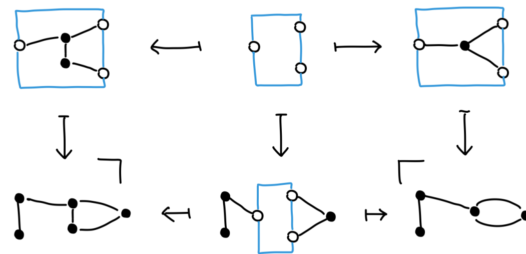
Assume a category of graphs **G** with graphs as objects and maps being injective graph homomorphisms. In double-pushout rewriting, the input information consists of a rewrite rule $L \Rightarrow R$ and a match $m : L \to G$ of the left-hand side (LHS) graph into a larger graph, indicating where to apply the rewrite rule inside a larger graph G. The overall operation is expressed by the commuting diagram in Figure 2a.

The boundary structures of the graphs L and R have to coincide for the rewrite rule to be well formed. To enforce this correspondence, a rule $L \Rightarrow R$ is represented as a span $L \leftarrow B \to R$, where B is a common subgraph of both L and R, specifying the interface of both. The other input to a DPO operation is the map $m : L \to G$ which specifies where to apply the rewrite rule within a larger graph G. Applying a rewrite rule consists of two stages: First we remove the LHS graph L from G by calculating the pushout complement of the composite arrow $B \to L \to G$. The result of removing L from G is a *context graph* $C = G \setminus L$ with a hole. The pushout complement construction ensures that the hole in C has the same boundary as L and R. The second step is the insertion of the RHS graph R into the hole in the context graph, calculated as the pushout of the span $C \leftarrow B \to R$.

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 $L \leftarrow B \to C$ (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. This is an elegant framework as it not only isolates the subgraph in question but also modularises rewriting. We are able to apply the rewrite in any other context graph in a straightforward manner, as long as the boundaries between hole and subgraphs match.

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

(a) Schema of DPO rewriting.



(b) A concrete example of DPO rewriting.

Figure 2b shows a concrete example of a DPO rewrite operation. 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 Adhesive categories [29, 28] are those in which rewriting by double-pushout makes sense. Formally, this is ensure by the existence of pushouts and pullbacks of monomorphisms and certain interactions between them. (For more details we refer to the literature.) We are particularly interested in the existence of pushouts in an adhesive category, and the uniqueness of pushouts when they exist. These properties ensure the sound rewriting of subgraphs. Most standard categories of graphs used for modelling string diagrams are adhesive.

2.2 Boundaries of Graphs

In the double-pushout rewriting formalism, the importance of the notion of boundary of a graph becomes apparent. A rewrite rule itself is well formed only if both subgraph have the same boundary, and inserting a subgraph into a context graph is possible only if the boundaries of the hole and the subgraph coincide. Boundary information in a graph G is represented by an injective graph morphism from the boundary graph B to G. The double-pushout construction relies on the boundary graph to guide the rewrite operation in order to compute a sound result.

Depending on the concrete category of graphs, the information constituting graph's boundary may differ. In this work we address with a whole class of categories, where we abstract over the concrete representation of boundary. For this, we will start from the notion of a PRO, a specific kind of monoidal category which we can interpret as general notion of graph structure. From a PRO we can derive the corresponding notion of boundary.

Definition 1. A *PRO* [27] ("product category") is a strict monoidal category whose objects are 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.

A morphism $m \to n$ in a PRO may be interpreted as a graph with n inputs and m outputs, with the tensor product being parallel composition of graphs. This notion of graph is the basis for our construction. Any graph defined in this way comes with a built-in notion of boundary with its domain and codomain.

Definition 2 (Graph Boundary)**.** The *boundary* of a graph $G : m \to n$ is defined as: $\partial(G) = \operatorname{dom} G \times \operatorname{cod} G$.

In addition to the objects of the corresponding PRO as the notion of boundary of a graph (i.e. a pair of natural numbers), we are interested in certain graphs themselves, encoding boundary information. The graph B in a DPO diagram is one such example. It contains the interface information of subgraphs and context graph, but nothing else. We define a *boundary graph* as the smallest graph for a given boundary.

Definition 3 (Boundary Graph)**.** Given a PRO $\mathbf{P}$. For each pair of objects $m,n \in \mathsf{Ob}(\mathbf{P})$, we define their *boundary graph* as a $\mathbf{P}$-morphism $G_\partial : m \to n$ such that for any other $\mathbf{P}$-morphism of the same type $G : m \to n$, there exists a morphism $G_\partial \to G$ of PROs.

Lemma 1. *A boundary graph $G_\partial(m,n)$ is unique for its boundary $m,n \in \mathsf{Ob}(\mathbf{P})$.*

Lemma 2. *Given a boundary graph $G_\partial(m,n)$, we have $(G_\partial \circ \partial)(G_\partial(m,n)) = G_\partial(m,n)$.*

We now define the class of graph categories that we work with for the rest of the paper. We define this category for any given PRO, ensuring that we can always compute the boundary for any graph.

Definition 4 (Category of Graphs)**.** Given a PRO $\mathbf{P}$, we define the category of graphs with boundaries, $\mathbf{G}(\mathbf{P})$, as follows:

- Objects are $\mathbf{P}$-morphisms $G : m \to n$.
- Morphisms $G \to G'$ are morphisms of PROs, preserving the boundary structure by definition.

In the following, we are interested in those categories $\mathbf{G}(\mathbf{P})$ which are *adhesive*, meaning that double-pushout rewriting is well defined and has similar properties to the category **Set**. Additionally, we require pushout complements to exist in the category $\mathbf{G}(\mathbf{P})$. Therefore, in the following, we assume these two properties on $\mathbf{G}(\mathbf{P})$, enabling access to rewriting by double pushout as well as the graphs and morphisms involved in a DPO diagram.

Remark 2. We introduce a drawing convention for graphs with boundaries, independent of the concrete category specification. As shown in Figure 3, we draw graphs as structures with vertices and a box around them, indicating their boundary. The box contains the graph only, with the edges crossing the border forming the interface of the graph. The actual information in a boundary graph depends on the specific instance; as we work at the meta level and as we use illustrations for explanatory purposes, this representation is sufficient.

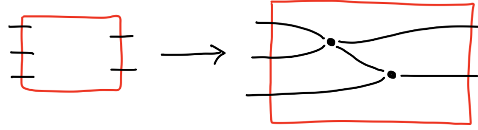


Figure 3: Illustration of a graph with boundary, independent of the concrete category instance.

2.3 Example Graph Instances

In this section We explain some examples of graph categories which are relevant as combinatorial representation of string diagrams. All of them form instances of the category of graphs $\mathbf{G}(\mathbf{P})$.

Directed Graphs Directed graphs are the most straightforward example of a graph category. With some careful constructing, they are a good candidate for representing morphisms in monoidal categories. A directed graph consists of a set of edges E and vertices V, together with source and target map for every edge, $s,t : E \to V$. Directed graphs and their homomorphisms form a category, called **Graph**. This category fits into our framework as rewriting is well defined:

Lemma 3. [29] *The category of graphs* **Graph** *is adhesive.*

As graphs are always closed (i.e. s and t are total function), the incorporation of boundary information requires a workaround. Typically, graphs with interfaces [17, 23] contain special kinds of vertices representing their interface connections. Any of these vertices have a single edge attached to them. A boundary graph in the category **Graph** consists of a set of interface vertices, as shown in Figure 4a.

Proposition 1. *Directed graphs are graphs with boundaries in the sense of Definition 4: Given two natural numbers $m,n \in \mathbb{N}$, a graph $G : m \to n$ consists of a set of edges E, a set of internal vertices V, and sets of external vertices I,O, together with a source function $s : E \to V + I$ and a target function $t : E \to V + O$. Further, we have $|I| = m$ and $|O| = n$. Sequential composition of graphs G and G' is well defined if $O_G = I_H$ and results in a graph $G \fatsemi H$ with $V = V_G + V_H$, $I = I_G$, $O = O_H$, $E = E_G + E_H / \sim$ where $e \sim e'$ whenever $t(e) = s(e')$. The source and target functions of $G \fatsemi H$ are defined as:*

$$s_{G\fatsemi H}(e) = \begin{cases} s_G(e) & e \in E_G \\ s_H(e) & \text{else} \end{cases} \qquad t_{G\fatsemi H}(e) = \begin{cases} t_H(e) & e \in E_H \\ t_G(e) & \text{else} \end{cases}$$

The tensor product of two graphs, $G \otimes H$, is defined as parallel composition: $E = E_G + E_H$, $V = V_G + V_H$, $I = I_G + I_H$, $O = O_G + O_H$, $s = s_G + s_H$, $t = t_G + t_H$.

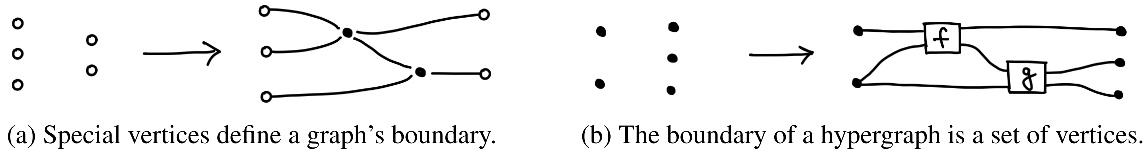
A boundary graph $G_\partial : m \to n$ consists of the two sets of external vertices I,O only.

Graphs are commonly used as combinatorial representation of string diagrams, for example in the underlying definition in tools like PyXZ [23] for reasoning with zx-diagrams and Quantomatic [25] for automated graph rewriting.

Hypergraphs An alternative representation of string diagrams are hypergraphs. A hypergraph consists of a set of vertices V and a set of hyperedges $E : \mathsf{List}\ V \to \mathsf{List}\ V$. Wires of a string diagram (corresponding to objects in the monoidal category) are translated to vertices of the hypergraph, and boxes (i.e. morphisms in the category) are represented as hyperedges.

Hypergraphs form a category **HGraph** for which rewriting is well defined:

Lemma 4. [16] *The category of hypergraphs* **HGraph** *is adhesive.*



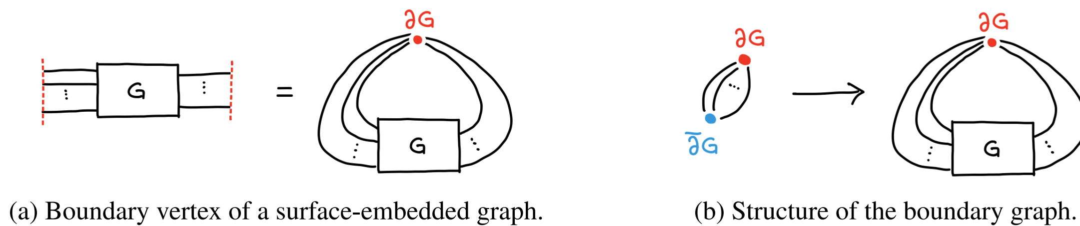
(a) Special vertices define a graph's boundary.
(b) The boundary of a hypergraph is a set of vertices.

Figure 4: The boundary structures of different types of graphs.

In their representation as hypergraphs, inputs and outputs of a string diagram are represented as a vertex. Therefore, the discrete graph consisting of vertices only serves as the notion of boundary graph, as shown in Figure 4b. Hypergraphs with boundaries have been studied in various lines of research [6, 7] and serve as the underlying data structure for the interactive theorem prover Chyp [26] for reasoning with symmetric monoidal categories.

Surface-Embedded Graphs Of particular interest to us are surface-embedded graphs which serve as a presentation of a certain class of string diagrams [2, 1]. Specifically, *plane* graphs encode diagrams for non-symmetric monoidal categories, encoding theories with a non-trivial topology e.g. quantum circuits [11]. We represent surface-embedded graphs as *rotation systems*: Each vertex is equipped with an ordering of its incident edges. This combinatorial data uniquely determines the topological embedding of the graph into a surface [20, 14].

To encode the inputs and outputs of a diagram, every surface-embedded graph is equipped with a special *boundary vertex*. Similar to the interface vertices for graphs, this is an auxiliary element in the graph, but here we only consider a single boundary vertex at the interface. All input and output edges of a graph are connected to this boundary vertex, illustrated in Figure 5a. This structure enables us to add embedding information to the interface edges to a graph, in form of a rotation at its boundary vertices. With a single vertex representing a graph's boundary we can ensures the preservation of the topology of a graph embedding when rewriting.



(a) Boundary vertex of a surface-embedded graph.
(b) Structure of the boundary graph.

Figure 5: The boundary structure of surface-embedded graphs.

Additionally, boundary vertices serve as auxiliary structure for specific regions of the surface-embedded graph. This may be the outside region, representing the graphs inputs and outputs, but we also use these special vertices to stand for any hole in a graph. For holes, the vertices are called *dual* boundary vertices $\bar{\partial}$. When inserting a subgraph into a hole, the structure of boundary and dual boundary vertices ensures the preservation of the rotation system and thus the surface embedding.

Boundary graphs in this category consists of two boundary vertices, ∂ and $\bar{\partial}$, connected by a number of edges. The rotation at these vertices contain as many edges as the graph has inputs and outputs, $n+m$. Mapping this boundary structure onto a graph consists of replacing the vertex $\bar{\partial}$ by a graph G while keeping the boundary vertex ∂ and all edges attached to it unchanged. To insert a graph into a hole,

we replace the boundary vertex which stands for the hole with a subgraph. Therefore, morphisms in the category of rotation systems have a partial vertex component [2]:

- Objects in the category of rotation systems **Rot** are total graphs (in which every edge is connected to a vertex) where for each vertex a cyclic list of edges is stored.
- Morphisms are graph morphisms with a partial vertex component. Crucially, if a morphisms is defined on a vertex, it has to preserve the order of edges around it. (For details, we refer to our previous work.) Further conditions define the notion of graph *embedding*.

The category of rotation systems is not adhesive (as the notion of monomorphisms is not useful for encoding graph embeddings), but we have shown that for specific kinds of spans, pushouts and pushout complements exist. These spans provide enough structure to express graph reasoning via an equational theory. In a *partitioning span*, one leg indicates the subgraph and one leg the hole into which we insert it. An example is shown in Figure 6. Crucially, all edges are always connected at both of their ends, ensuring the surface properties when rewriting.

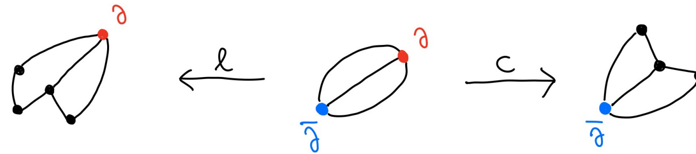


Figure 6: Example of a partitioning span.

Definition 5. A *partitioning span* is a span $L \xleftarrow{l} B \xrightarrow{c} C$ in **Rot**, 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.

Partitioning spans capture precisely the situations in which we can perform a rewrite operation. We have shown that the category **Rot** rewriting of partitioning spans produces well formed results. These adhesive properties are enough for the category of rotation systems to fit into our framework of graph pattern matching.

Lemma 5. [2] *In the category of rotation systems* **R***, pushouts of partitioning spans exist and pushout complements exist and are unique (up to scalars).*

3 Graphs as Multicategories

Typically, categories of graphs are built from a number of generators together with operations to compose graphs in sequence or in parallel, according to the structure of the monoidal category whose diagrams they represent. Derived from these operations is a notion of graph substitution, for example as a pushout. We now change perspective and define a framework in which graph *substitution* is a primitive operation. This framework consists of multicategories whose arrows take multiple input objects to a single output object. We will show that our definition of substitution for graphs can be used as the central operation of a *multicategory*. In the context of diagrammatic languages, multicategories are convenient structures to talk about the layout of a diagram and precisely specify the relation between subdiagrams.

Remark 3. Multicategories are also known as *coloured operads*, or sometimes merely *operads*. We prefer the term multicategory to remain consistent when we consider the opposite construction in Section 4.

The multicategory which we will define here is inspired by the following two examples:

Example 1 (The Little Disks Operad [32, 5, 31])**.** In surface embedded graphs, every part of the graph takes up some of the surface the graph is embedded in. Gluing together the faces of the graph embedding forms the entire surface. A similar topological motive is the basis of the Little Disks Operad: This operad specifies the layout of a finite number of disjoint closed disks inside a larger disk. The objects are disks, with an n-ary map defining the spatial arrangement of n small disks inside a larger disk.

Example 2 (The Operad of Wiring Diagrams [38])**.** David Spivak's Wiring Diagrams Operad is an important inspiration for this work. This operad defines wiring diagrams which are formed from nodes and edges, together with special vertices called *stars*. Composition of wiring diagrams is defined by the substitution of a diagram for a star of the same type inside a larger one. We will have a similar notion of graph variables (and later, pattern variables) in our framework which define the positions in a graph where we can insert a subgraph.

Definition 6. A small *multicategory* $\mathbf{C}$ consists of:

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

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

satisfying unit and associativity laws that we will not reproduce here but refer to the literature [31].

By default, composition of multicategories is defined for all inputs $a_1, \ldots, a_n$ at once, but we can express composition at one input by using the identity map on all other inputs (illustrated in Figure 13). For simplicity, we will use indexed composition as the default version. We do not lose generality by doing so as we consider the subgraphs as inputs to a multicategory map to be disjoint.

3.1 The Multicategory of Surface-Embedded Graphs and Substitution

We define a multicategory whose objects are boundaries of graphs and whose morphisms are graphs with boundaries. A graph may contain multiple holes, whose types serve as inputs of the multicategory map. We call the inputs graph *variables*. The output type is the boundary of the overall graph. Composition of maps is given by the substitution of one graph H into the hole of another graph G, defined as an instance of DPO rewriting along the interface of H and the corresponding hole in G.

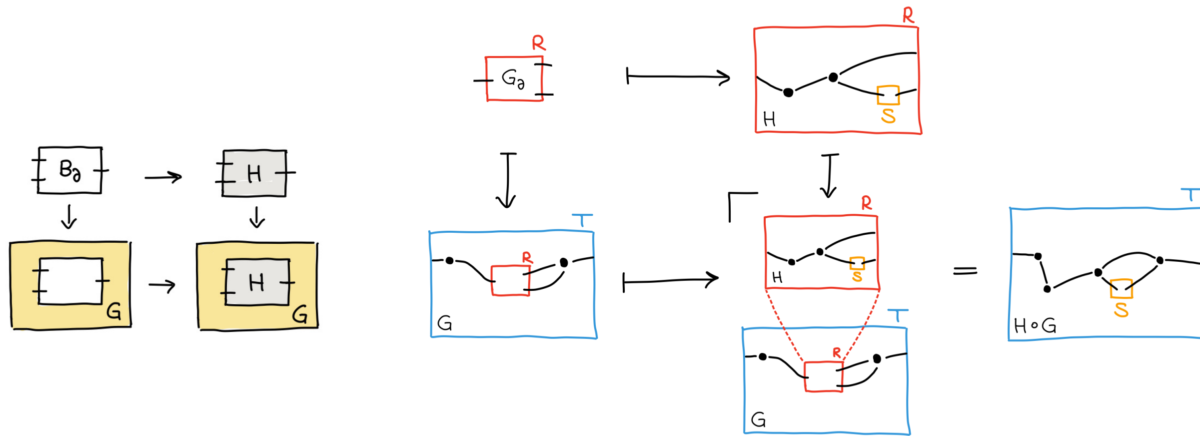
Proposition 2. *A category of graphs* $\mathbf{G}(\mathbf{P})$ *forms a multicategory as follows:*

- *The objects are pairs of objects in* $\mathbf{P}$*, i.e. pairs of natural numbers, representing* boundaries of graphs. *We call the objects* graph variables.
- *An n-ary map* $T_1, \ldots, T_n \to T_G$ *is a graph* G *in the category* $\mathbf{G}(\mathbf{P})$ *of type* $\partial(G) = T_G$*, containing subgraphs of the input types, i.e. there are* $\mathbf{G}(\mathbf{P})$*-morphisms* $G_\partial(T_1) \to G, \ldots, G_\partial(T_n) \to G$*.*
- *Composition* $H \circ_i G$ *of maps* $G : T_1, \ldots, T_n \to T_G$ *and* $H : S_1, \ldots, S_k \to T_i$ *is defined as the pushout of* $H \leftarrow G_\partial(T_i) \to G$ *in the category* $\mathbf{G}(\mathbf{P})$*. This calculates the substitution of* H *for the subgraph* $G_\partial(T_i)$ *in* G*. The result is of type:*
  $H \circ_i G : T_1, \ldots, T_{i-1}, S_1, \ldots, S_k, T_{i+1}, \ldots, T_n \to T_G$.

- *The identity map $T_G \to T_G$ is the boundary graph $G_\partial(T_G)$.*

Proof. Recall that we are assuming $\mathbf{G}(\mathbf{P})$ to be adhesive and that pushout complements exist. Since pushouts are associative in adhesive categories [4], composition in this multicategory is associative, too. Given an object T in the multicategory, the pushout of a cospan $G \leftarrow G_\partial(T) \to G_\partial(T)$, where the right leg is the identity morphism on $G_\partial(T)$, is just the object G (correspondingly for the left leg.) Thus, the identity morphisms is the unit of composition in the multicategory. □

Figure 7a illustrates graph composition by pushout schematically and Figure 7b shows a concrete example of the operation where both graphs G and H have one input variable.



(a) Composition $H \circ G$.

(b) Composition of $G : R \to T$ and $H : S \to R$ along G_∂(R).

Figure 7: Operad composition corresponds to substitution of graphs by pushout.

Remark 4. We note the analogy of the composition operation with pushouts of partitioning spans in surface-embedded graphs in Section 2.3. One side of the span specifies a subgraph (here, $G_\partial \to H$) and the other one marks the hole in a context graph ($G_\partial \to G$). In the case of surface-embedded graphs, the multicategory definition exposes another property: the inputs to an morphism define the types of the dual boundary vertices in a graph (standing for holes) and the output has the type of its boundary vertex.

4 Graph Patterns

The multicategory picture of graphs implements the fact that a graph can be described as an arrangement of smaller graphs. A map in the multicategory take the types of the smaller graphs as inputs and specifies how they are connected with each other to build a larger graph. We now look at the opposite picture and instantiate a framework for splitting an overall structure into smaller pieces for graphs. A morphism in a *co-multicategory* takes one input argument and produces multiple outputs. We argue that the opposite structure to the multicategory of graphs defines a co-multicategory of graph *patterns*.

4.1 The Co-Multicategory of Patterns and Substitution

Proposition 3. *A category of graphs* $\mathbf{G}(\mathbf{P})$ *forms a co-multicategory as follows:*

- *The objects are pairs of objects of* $\mathbf{P}$*, i.e. pairs* $(n,m) \in \mathbb{N} \times \mathbb{N}$*. We think of objects as representing* pattern variables.
- *An n-ary map* $T_P \to T_1,\dots,T_n$ *is a graph P in the category* $\mathbf{G}(\mathbf{P})$ *with boundary* $\partial(P) = T_P$ *and containing subgraphs of the output types, i.e. there are* $\mathbf{G}(\mathbf{P})$*-morphisms* $G_\partial(T_1) \to P, \dots, G_\partial T_n \to P$*. We call such an n-ary map a* graph pattern.
- *Composition* $Q \circ_i P$ *of two patterns P and Q corresponds to the substitution of P for the pattern variable* T_i *in Q, calculated by the pushout* $P \leftarrow G_\partial(T_i) \to Q$ *in* $\mathbf{G}(\mathbf{P})$*. We think of this composition operation as pattern* refinement.
- *The identity pattern* $T_P \to T_P$ *is the corresponding boundary graph* $G_\partial(T_P)$*.*

Proof. For the same reasons as in Proposition 2, this defines a co-multicategory. □

Patterns contain the same information as graphs, but the *flow* of this information is different. Graphs combine multiple elements of types T_i into one larger structure of type T_G, whereas patterns take an overall structure of type T_P and return multiple substructures of types T_i. Thus, patterns can extract certain subregions from a larger graph. Composition in the co-multicategory describes the refinement of patterns: one of the pattern variables is instantiated with another pattern, thus overall the pattern becomes more specific. Patterns themselves do not reduce, we can only refine them by composition. To reduce a pattern, we need to apply it to a graph. As both graphs and patterns are defined as morphisms in $\mathbf{G}(\mathbf{P})$, we can study their interaction in the underlying category.

Remark 5. Note that the underlying category $\mathbf{G}(\mathbf{P})$ is used as language for both graphs and patterns. This is in contrast to most term languages, which distinguish between a pattern and an expression language. This is a conscious choice as graphical calculi like quantum circuits encode control flow as part of their data. For example, in the ZX-calculus [10], conditional application is encoded by control operations. Another example comes from the context of reversible programming: in the programming language Theseus [22], matching on a function application (fg) on the left hand side of a definition corresponds to applying the inverse of the function f^{-1} to the right hand side.

5 Graph Pattern Matching

We now describe the operation of pattern matching for graphs in two steps: First, we specify what we mean by a graph matching a pattern. Afterwards we show how to calculate the pattern match operation, resulting in subgraphs instantiating the pattern's variables. We use the fact that graphs and patterns both live in the category $\mathbf{G}(\mathbf{P})$and define all relevant structures as morphisms in this category. We start by specifying what we mean by a *match* of a graph against a pattern.

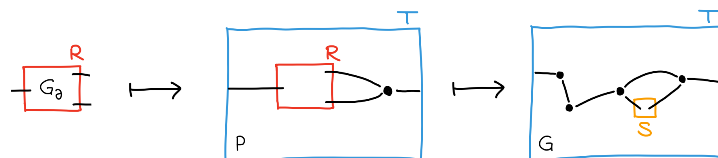
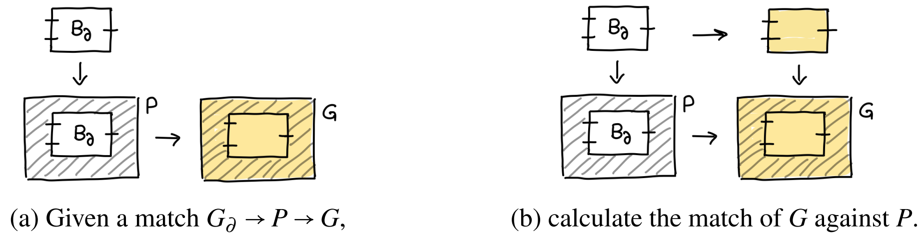


Figure 8: Example of a match of a graph $G : S \to T$ against a pattern $P : T \to R$.

Definition 7. Given a graph $G : S_1,\ldots,S_k \to T$ and a pattern $P : T \to R_1,\ldots,R_n$, a *match* is given by a composite of $\mathbf{G}(\mathbf{P})$-morphisms, $m : G_\partial(R_1) \otimes \ldots \otimes G_\partial(R_n) \to P \to G$. A match fails, if no such morphism m exists.

Figure 9a depicts the schema of a match and Figure 8 shows a concrete example.

Mapping a pattern onto a graph identifies the elements which both structures share. All vertices and edges in the pattern have to be mapped to the same elements in the graph, as all morphisms in $\mathbf{G}(\mathbf{P})$ are injections. In addition to these *shared* parts between pattern and graph, a graph may contain more elements. These are the components which are going to be exposed by a pattern matching operation.



(a) Given a match $G_\partial \to P \to G$, (b) calculate the match of G against P.

Figure 9: Schema of calculating a pattern match by taking the pushout complement of a match.

A match has exactly the structure of one side of a pushout square in the category $\mathbf{G}(\mathbf{P})$. To expose the subgraphs encoded by variables in the pattern and thus calculate the pattern match, we calculate the pushout complement of the match.

Definition 8. Given a graph G, a pattern P, and a match $m : P \to G$. The pushout complement of the composite $G_\partial \to P \to G$ is performing the *pattern match* of G against P which we write as $G \bowtie P$.

The pattern match returns the subgraphs of G that instantiate the pattern's variables, i.e. replacing the pattern's holes vertices in the match m.

Lemma 6. *Given a match $m : P \to G$, the calculation of the result of the pattern matching operation always exists and is unique.*

Proof. As we assume the existence of pushout complements for the category $\mathbf{G}(\mathbf{P})$, the uniqueness comes from the adhesive property of the category. □

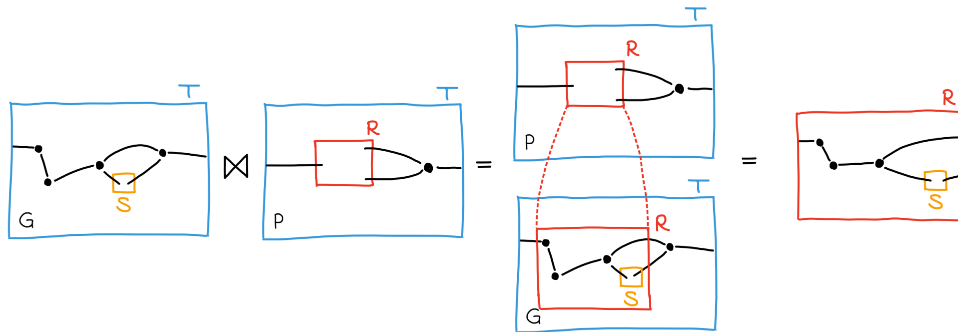


Figure 10: A concrete example of a pattern matching operation: Given the match from Figure 8, the result of the pattern match is a graph $G \bowtie P : S \to R$.

Example 3. The identity morphism $G_\partial : T_G \to T_G$ represents the wildcard pattern, matched by any graph.

Surface-Embedded Graphs As surface-embedded graphs are only adhesive in certain cases, some of the constructions for patterns and pattern matching require some care. For example, in the definition of a match, we require that the composite $G_\partial \to P \to G$ has the structure of a *boundary embedding* which is the corresponding structure to a partitioning span for calculating pushout complements. This is due to the fact that pushout complements are only defined for this kind of composite.

Definition 9. A *boundary embedding* is a pair of maps $G_\partial \xrightarrow{l} L \xrightarrow{m} G$ in **G**, where G_∂ is a boundary graph, and where: (i) $l_V(\partial)$ is defined on the boundary vertex but $l_V(\bar{\partial})$ is undefined on the dual boundary vertex; and (ii) $(m_V \circ l_V)(\partial)$ is undefined on the boundary vertex. Further, L has to be a connected graph, and m an embedding.

A boundary embedding forms one corner of a pushout square. The other corner of such a diagram is specified as an *opposite boundary embedding*: it is a composite $G_\partial \to C \to G$ in which the morphisms have exchanged properties to a boundary embedding. As interface edges have to be ordered for surface-embedded graphs, we can prove that there is at most one match of a graph against a pattern.

Lemma 7. *In* **Rot***, if a match $m : P \to G$ exists, it is unique for a pattern P and a graph G.*

Proof. As the boundary graph in the definition of a match (Definition 7) is a tensor product, we assume that P has one output variable B without loss of generality. We assume a match $m : P \to G$. The pattern $P : \partial(G) \to \partial(B)$ is a plane graph containing vertices ∂(G) as its boundary and $\bar{\partial}(B)$. Similarly, the graph $G : \partial(A) \to \partial(G)$ is a plane graph which contains vertices $\bar{\partial}(A)$ and $\partial(G)$. First, we observe that because m is part of an opposite boundary embedding, it is defined on all vertices $v \in V(P)$ except the dual boundary vertex $\bar{\partial}(B)$. Second, by the definition of morphism in **Rot**, m has to preserve the rotations of all the vertices it is defined on. Thus, a match only exists, if all rotations in P and G coincide. □

6 Applications

As an application of our pattern matching framework, we consider work on an equational theory for controlled quantum circuits [21]. This theory consists of a number of circuit equations for which we imagine a rewrite function replacing the left hand side of an equation with the right hand side inside a bigger diagram. These rules are polymorphic, as we can call them on any circuit matching their structure. Executing a rewrite rule means matching a concrete circuit against the pattern provided in the rule, thus instantiating the pattern variables with subcircuits. Figure 11 shows an example rewrite rule.

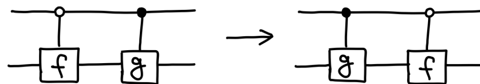


Figure 11: One of the equations on control in quantum circuits, presented as a rewrite rule.

The diagram on the left hand side can be represented as a pattern $(2,2) \to (1,1) \otimes (1,1)$ with pattern variables f and g. When the rewrite function is called on a concrete circuit, f and g are instantiated with subcircuits. The result of the rewrite is calculated by the substitution of f g with the concrete subcircuits in the right-hand side of the rule.

Another example application for our language are global rewrite rules, such as phase teleportation in quantum circuits using the ZX-calculus [24]. This work introduces global rewrite rules which reduce the number of T-gates in a circuit. While the T-gate locations are specified globally, the rule remains polymorphic with respect to the subcircuits positioned in between them. We can describe the structure of this rewrite rule as a pattern, with pattern variables standing for subcircuits.

References

[1] Malin Altenmüller (2025): *Combinatorial Presentations of String Diagrams for Non-Symmetric Monoidal Categories*. Ph.D. thesis, University of Strathclyde. Available at `https://maltenmuller.github.io/thesis/thesis.pdf`.

[2] Malin Altenmüller & Ross Duncan (2022): *A Category of Surface-Embedded Graphs*. In Jade Master & Martha Lewis, editors: *Proceedings Fifth International Conference on Applied Category Theory, ACT 2022, Glasgow, United Kingdom, 18-22 July 2022*, 380, pp. 41–62, doi:10.4204/EPTCS.380.3. Available at `https://doi.org/10.4204/EPTCS.380.3`.

[3] Krzysztof Bar, Aleks Kissinger & Jamie Vicary (2018): *Globular: an online proof assistant for higher-dimensional rewriting*. *Log. Methods Comput. Sci.* 14, doi:10.23638/LMCS-14(1:8)2018. Available at `https://doi.org/10.23638/LMCS-14(1:8)2018`.

[4] Nicolas Behr & Paweł Sobociński (2020): *Rule Algebras for Adhesive Categories*. *Logical Methods in Computer Science* 16, doi:10.23638/LMCS-16(3:2)2020.

[5] J. M. Boardman & R. M. Vogt (1973): *Homotopy invariant algebraic structures on topological spaces*. *Lecture Notes in Mathematics* 347, Springer-Verlag, Berlin, New York, doi:10.1007/BFb0068547.

[6] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2016): *Rewriting modulo symmetric monoidal structure*. In Martin Grohe, Eric Koskinen & Natarajan Shankar, editors: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, ACM, pp. 710–719, doi:10.1145/2933575.2935316. Available at `https://doi.org/10.1145/2933575.2935316`.

[7] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski & Fabio Zanasi (2017): *Confluence of Graph Rewriting with Interfaces*. In Hongseok Yang, editor: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, 10201, Springer, pp. 141–169, doi:10.1007/978-3-662-54434-1_6. Available at `https://doi.org/10.1007/978-3-662-54434-1_6`.

[8] Filippo Bonchi, Joshua Holland, Robin Piedeleu, Pawel Sobocinski & Fabio Zanasi (2019): *Diagrammatic algebra: from linear to concurrent systems*. *Proc. ACM Program. Lang.* 3, p. 25:1–25:28, doi:10.1145/3290338. Available at `https://doi.org/10.1145/3290338`.

[9] Rod M Burstall (1969): *Proving properties of programs by structural induction*. *The Computer Journal* 12, pp. 41–48, doi:10.1093/comjnl/12.1.41.

[10] Bob Coecke & Ross Duncan (2011): *Interacting quantum observables: categorical algebra and diagrammatics*. *New Journal of Physics* 13, p. 43016.

[11] Bob Coecke, Ross Duncan, Aleks Kissinger & Quanlong Wang (2015): *Generalised compositional theories and diagrammatic reasoning*, pp. 309–366. Springer.

[12] Bob Coecke & Aleks Kissinger (2018): *Picturing quantum processes: A first course on quantum theory and diagrammatic reasoning*. In: *International conference on theory and application of diagrams*, pp. 28–31.

[13] Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru & Jamie Vicary (2024): *homotopy.io: A Proof Assistant for Finitely-Presented Globular n-Categories*. In Jakob Rehof, editor: *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, Tallinn, Estonia, July 10-13, 2024*, 299, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, p. 30:1–30:26, doi:10.4230/LIPICS.FSCD.2024.30. Available at `https://doi.org/10.4230/LIPIcs.FSCD.2024.30`.

[14] Jack R Edmonds (1960): *A combinatorial representation for oriented polyhedral surfaces*. *Notices of the American Mathematical Society* 7, p. 646.

[15] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Springer, doi:10.1007/3-540-31188-2. Available at `https://doi.org/10.1007/3-540-31188-2`.

[16] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski & Francesco Parisi-Presicce (1991): *Parallelism and concurrency in high-level replacement systems*. *Mathematical Structures in Computer Science* 1, pp. 361–404, doi:10.1017/S0960129500001353.

[17] Marcelo Fiore & Matias De la Cuadra Campos (2013): *The Algebra of Directed Acyclic Graphs*. In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '13)*, IEEE Computer Society, pp. 323–332, doi:10.1109/LICS.2013.38.

[18] Michael J Gordon, Arthur J Milner & Christopher P. Wadsworth (1979): *Edinburgh LCF: A Mechanised Logic of Computation*. 78, Springer-Verlag, doi:10.1007/3-540-09724-4.

[19] Jules Hedges, Evguenia Shprits, Viktor Winschel & Philipp Zahn (2016): *Compositionality and String Diagrams for Game Theory*. *CoRR* abs/1604.06061. Available at `http://arxiv.org/abs/1604.06061`.

[20] Lothar Heffter (1891): *Über das Problem der Nachbargebiete*. *Mathematische Annalen* 38, pp. 477–508.

[21] Chris Heunen, Robin Kaarsgaard & Louis Lemonnier (2025): *One rig to control them all*. Available at `https://arxiv.org/abs/2510.05032`.

[22] Roshan P James & Amr Sabry (2014): *Theseus: A High Level Language for Reversible Computing*. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, ACM, pp. 95–107, doi:10.1145/2535838.2535845.

[23] Aleks Kissinger & John van de Wetering (2020): *PyZX: Large Scale Automated Diagrammatic Reasoning*. In: *Proceedings 16th International Conference on Quantum Physics and Logic (QPL 2019)*, 318, pp. 229–241, doi:10.4204/EPTCS.318.14.

[24] Aleks Kissinger & John van de Wetering (2020): *Reducing the number of non-Clifford gates in quantum circuits*. *Physical Review A* 102, p. 22406, doi:10.1103/PhysRevA.102.022406.

[25] Aleks Kissinger & Vladimir Zamdzhiev (2015): *Quantomatic: A Proof Assistant for Diagrammatic Reasoning*. In Amy P Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, 9195, Springer, pp. 326–336, doi:10.1007/978-3-319-21401-6_22. Available at `https://doi.org/10.1007/978-3-319-21401-6_22`.

[26] Alex Kissinger (2023): *Chyp: An interactive theorem prover for string diagrams*.

[27] Stephen Lack (2004): *Composing PROPs*. *Theory and Applications of Categories* 13, pp. 147–163. Available at `http://www.tac.mta.ca/tac/volumes/13/9/13-09abs.html`.

[28] Stephen Lack & Pawel Sobocinski (2005): *Adhesive and quasiadhesive categories*. *RAIRO Theor. Informatics Appl.* 39, pp. 511–545, doi:10.1051/ITA:2005028. Available at `https://doi.org/10.1051/ita:2005028`.

[29] Stephen Lack & Paweł Sobociński (2004): *Adhesive categories*. In: *International Conference on Foundations of Software Science and Computation Structures*, pp. 273–288.

[30] Saunders Mac Lane (1965): *Categorical Algebra*. *Bulletin of the American Mathematical Society* 71, pp. 40–106, doi:10.1090/S0002-9904-1965-11234-4.

[31] Tom Leinster (2004): *Higher operads, higher categories*. Cambridge University Press.

[32] J Peter May (1972): *The Geometry of Iterated Loop Spaces*. 271, Springer-Verlag, doi:10.1007/BFb0067491.

[33] Frederik V McBride, Donald J T Morrison & Raymond M Pengelly (1970): *A symbol manipulation system*, pp. 337–347. Edinburgh University Press.

[34] Paul-André Melliès (2014): *Local States in String Diagrams*. In Gilles Dowek, editor: *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014, Proceedings*, 8560, Springer, pp. 334–348, doi:10.1007/978-3-319-08918-8_23. Available at `https://doi.org/10.1007/978-3-319-08918-8_23`.

[35] Maciej Piróg & Nicolas Wu (2016): *String diagrams for free monads (functional pearl)*. *ACM SIGPLAN Notices* 51, doi:10.1145/2951913.2951947.

[36] Peter Selinger (2011): *A Survey of Graphical Languages for Monoidal Categories*, doi:10.1007/978-3-642-12821-9_4.

[37] Pawel Sobocinski (2013): *Nets, Relations and Linking Diagrams*. In Reiko Heckel & Stefan Milius, editors: *Algebra and Coalgebra in Computer Science - 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings*, 8089, Springer, pp. 282–298, doi:10.1007/978-3-642-40206-7_21. Available at `https://doi.org/10.1007/978-3-642-40206-7_21`.

[38] David I Spivak (2013): *The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits*. Available at `https://arxiv.org/abs/1305.0297`.

A Supplementary Material for Section 1

Patterns provide a polymorphic way of specifying multiple terms of algebraic types that share a similar structure. They consist of metavariables (called *pattern variables*) and constructors of the relevant type. Pattern variables represent any term, but the positioning of a constructor within the pattern a is fixed information. A pattern may consist of a single constructor or a sequence of multiple constructor applications, e.g. two applications of the `cons` constructor for lists describing any list with at least two elements.

A.1 Haskell Example

Figure 12 shows the implementation of the data type of lists in Haskell, together with two functions on lists. `isEmpty` checks whether a list contains any elements and `sum` adds all elements of a list. Both functions are defined by list patterns on their left hand side with each containing a base case using the empty list pattern `[]`. In addition, the `sum` function provides a case for non-empty lists, defined by a pattern containing the pattern variables `x` and `xs` and one call to the cons constructor `(:)`.

```
data List a = [] | a : List a

isEmpty : [a] -> Bool
isEmpty [] = True
isEmpty _ = False
```

```
sum :: Num a => List a -> a
sum [] = 0
sum (x : xs) = x + sum xs

s = sum (26 : 18 : 7 : 12 : [])
```

Figure 12: The data type of lists and example operations on lists, in Haskell.

The last line in In Figure 12 shows an example of a function call to the `sum` function on the concrete list `26:18:7:12:[]`. The input list contains the `(:)` constructor, therefore it matches the pattern `x:xs` in the second case. Subsequently, the pattern matching operation maps the structure `_:_` specified in the pattern onto the concrete term and instantiates the pattern variables `x` and `xs` with concrete values: `x` with the numeral value `26` from the head of the list and `xs` with the tail list `18:7:12:[]`. On the right hand side of the function, these two values can now be *substituted* for the pattern variables in the function implementation: `26` is used as the summand and the tail list as argument of the recursive call to `sum`.

B Supplementary Material for Section 3

Composition of multicategories is typically defined for all inputs $a_1, \ldots, a_n$ at once, but we can express composition at one input by using the identity map on all other inputs:

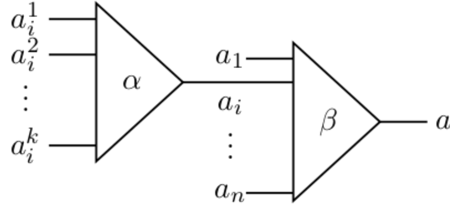


Figure 13: Schema of indexed multicategory composition $\mathbf{C}(a_1, \ldots, a_n; a) \circ_i \mathbf{C}(a_i^1, \ldots, a_i^k; a_i)$.