

An Introduction to String Diagrams for Computer Scientists

Robin Piedeleu and Fabio Zanasi

May 16, 2023

Abstract

This document is an elementary introduction to string diagrams. It takes a computer science perspective: rather than using category theory as a starting point, we build on intuitions from formal language theory, treating string diagrams as a syntax with its semantics. After the basic theory, pointers are provided to contemporary applications of string diagrams in various fields of science.

Contents

1	The Case for String Diagrams	2
2	String Diagrams as Graphical Syntax	5
2.1	Adding Equations	13
2.2	Common Equational Theories	14
3	String Diagrams as Graphs	20
4	Categories of String Diagrams	25
4.1	Fewer Structural Laws	25
4.1.1	Monoidal Categories	25
4.1.2	Braided Monoidal Categories	26
4.2	More Structural Laws	27
4.2.1	Traced Monoidal Categories	27
4.2.2	Compact Closed Categories	29
4.2.3	Self-dual Compact Closed Categories	30
4.2.4	Copy-Delete Monoidal Categories	31
4.2.5	Cartesian Categories	32
4.2.6	CoCartesian Categories	33
4.2.7	Biproduct Categories	34
4.2.8	Hypergraph Categories	34
4.2.9	Mix and Match	36
5	Semantics	37
5.1	From Syntax to Semantics, Functorially	37
5.2	Soundness and Completeness	52

6	Other Trends in String Diagram Theory	56
6.1	Rewriting	56
6.2	Higher-Dimensional Diagrams	58
6.3	Inequalities	60
6.4	Relationship with Proof Nets	61
6.5	Software	62
7	String Diagrams in Science and Engineering: Some Applications	63

1 The Case for String Diagrams

The algebraic structure of programs When learning a programming language, one of the most basic tasks is understanding how to correctly write programs in the language *syntax*. This syntax is often specified as a context-free grammar. For instance, the grammar defining the syntax of a very elementary imperative programming language, where variables x, y, \dots and natural numbers $n \in \mathbb{N}$ may occur, is the following:

$$\begin{array}{ll}
 b & ::= \text{True} \mid x = y \mid x = n \mid \neg b \mid b \wedge b \mid b \vee b \\
 p & ::= \text{skip} \mid x := y \mid x := n \mid \text{while } b \text{ to } p \mid p ; p
 \end{array} \tag{1}$$

With this grammar, we can write arbitrary programs featuring assignment of value to a variable, while loops, and program concatenation. In particular, while loops will depend upon a boolean expression, whose construction is dictated by the first row of the grammar. For practitioners, this information is essential to correctly write code in the given language: an interpreter will only execute programs that are written conformably to the grammar. For computer scientists, interested in formal analysis of programs, this information has deeper consequences: it gives us a powerful tool to prove *mathematical properties* of the language. This happens in the same way as we reason, for instance, about natural numbers. The set \mathbb{N} of natural numbers can also be specified via a grammar:

$$n ::= 0 \mid n + 1 \tag{2}$$

Using this specification, we can define the usual arithmetic operations over the naturals and prove their properties, *by induction*. With this method, the base case and the inductive step correspond precisely to the clauses of grammar (2). For instance, we can prove that, for each $n \in \mathbb{N}$, $n - n = 0$, by saying that this is true for 0, i.e. $0 - 0 = 0$, and if we assume $m - m = 0$ for a given $m \in \mathbb{N}$ then it is true for $m + 1$, because $(m + 1) - (m + 1) = (m - m) + (1 - 1) = 0 + 0 = 0$.

In the same way, we can reason by induction on programs, precisely in virtue of the specification of their syntax via a grammar. For example, we can prove that a property P holds for the programs of the language specified by the grammar (1) by induction as follows: first, we show that P holds for *skip*, $x := y$, and $x := a$. Then, assuming P holds for p , we show that it holds for *while* b to p . Finally, assuming P holds for programs p and p' , we show that it holds for $p ; p'$.

This style of reasoning is extremely useful for a number of tasks. For instance, we may prove by induction properties demonstrating the *correctness*, *safety*, or *liveness* of our program. We may also define its *semantics* by induction, i.e., assign programs their behaviour in a way that respects their structure. In programming language theory, there are usually two different ways of defining the semantics of a language: operational and denotational. The former specifies directly *how* to execute every expression, while the latter specifies *what* an expression means by assigning it a mathematical objects that abstracts its intended behaviour. An inductively defined semantics is

particularly important because it enables *compositional* (also called *modular*) reasoning: the meaning of a complex program may be entirely understood in terms of the semantics of its more elementary expressions. For instance, if our semantics associates a function $[p]$ to each program p , and associates to $p ; p'$ the composite function $[p'] \circ [p]$, that means that the semantics of the expression $p ; p'$ *exclusively* depends on the semantics of simpler expressions p and p' .

Moreover, the description of a language as a syntax equipped with a compositional semantics informs us about the *algebraic* structures underpinning program design. For instance, in any sensible semantics, the program constructs $;$ and *skip* of the grammar (1) acquire a *monoid* structure, with the binary operation $;$ as its multiplication and the constant *skip* as its identity element. Indeed, the laws of monoids, namely that $[(p;q);r] = [p;(q;r)]$ (associativity) and $[p;skip] = [p] = [skip;p]$ (unitality), will usually hold for the semantics of these operations.

Graphical Models of Computation As we have seen, defining a formal language via an inductively defined syntax brings clear benefits. However, not all computational phenomena may be adequately captured via this kind of formalisms. Think for instance about signal flowing through a digital controller. In this model, information propagates through components in complex ways, requiring constraints on how resources are processed—for example, a gate may only receive a certain quantity of signal at a time, or a deadlock will occur. More sophisticated forms of interaction, such as entanglement in quantum processes, or conditional (in)dependence between random variables in a probabilistic systems, also require a language capable of capturing resource-exchange between components in a clear and expressive manner.

Historically, scientists have adopted *graphical* formalisms to properly visualise and reason about these phenomena. Graphs provide a simple pictorial representation of how information flows through a component-based system, which is difficult to encode into a conventional textual notation. Notable examples of these formalisms include electrical and digital circuits, quantum circuits, signal flow graphs (used in control theory), Petri nets (used in concurrency theory), probabilistic graphical models like Bayesian networks and factor graphs, and neural networks.

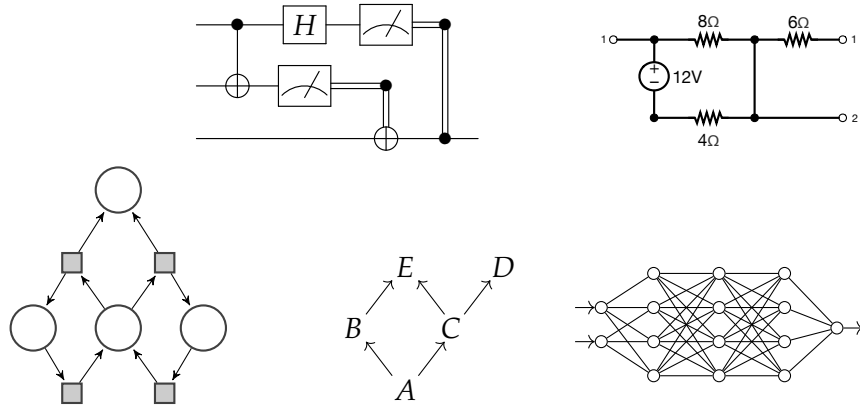


Figure 1: *Some examples of graphical formalisms: a quantum circuit, an electric circuit, a Petri net, a Bayesian network, and a neural network.*

On the other hand, graphical models have clear drawbacks compared to syntactically defined formal languages. Our ability to reason mathematically about combinatorial structures like graphs is limited: notably, we cannot use structural induction on the model components, as we would

with a standard program syntax. This is because a syntax gives *operations* for combining simpler models into complex ones, while graphical models are often treated as a monolithic entity rather than modularly, as made-up of sub-components. We typically miss formal means both to *decompose* these models and *compose* them together. Crucial features of program analysis, such as the definition of a compositional semantics and the investigation of their algebraic structures, face significant obstacles when performed on graphical formalisms. Luckily, as we will see, there is nothing essential about the dichotomy between syntax and graphical representations.

String Diagrams: The Best of Both Worlds String diagrams originate in the abstract mathematical framework of *category theory*, as a pictorial notation to describe the morphisms in a monoidal category. However, over the past three decades their use has expanded significantly in computer science and related fields, extending way beyond their initial purpose.

What makes string diagrams so appealing is their dual nature. Just like graphical models, they are a *pictorial* formalism: we can specify and reason about a string diagram as if it was a graph, with nodes and edges. However, just like programming languages, string diagrams may be specified and analysed as a formal *syntax*: we can think of the graphs they allow as made of elementary components (akin to the gates of a circuits, but a lot more general than that), composed via syntactically defined operations.

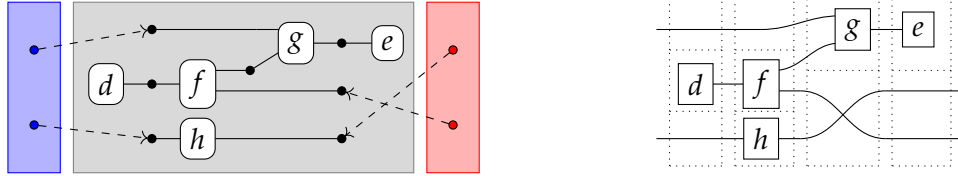


Figure 2: An example of a string diagram regarded as a (hyper)graph (left), with blue and red boxes signalling the interfaces for composing with other digrams, and the same string diagram regarded as a piece of syntax (right), with dotted lines placed to emphasise where elementary components compose, vertically and horizontally. Contrast these with the symbolic notation for the same syntactic entity, in the language of symmetric monoidal categories: $(id \otimes d \otimes id); (id \otimes f \otimes h); (g \otimes \sigma); (e \otimes id \otimes id)$.

Remarkably, understanding string diagrams as syntactically defined objects does not require switching to a different (textual) formalism: the graphical representation itself *is* made of syntax. The theory of monoidal categories provides a rigorous formalisation of how to switch between the combinatorial and the syntactic perspective on string diagrams, as well as a rich framework to investigate their semantics and algebraic properties. Indeed, like programming languages, or the syntax of an algebraic theory, we can assign a semantics to diagrams in a compositional way. This gives a modular way to specify and reason about the behaviour of the models that they represent.

String Diagrams in Contemporary Research Building on this duality of representation, string diagrams have been mainly used in two opposite manners: as a way to reason about graphical models syntactically, and as a way to reason about (textual) formal languages in a more visual, resource-sensitive manner.



Figure 3: A main reason for their appeal in certain areas of computer science, such as quantum theory, is that string diagrams are resource-sensitive by design: the graphical formalism uncovers any implicit assumption on how resources are handled during the computation. For instance, the term $f(g(x), g(x), y)$ may be represented with two distinct string diagrams, depending on whether we want g to ‘consume’ one (right) or two (left) copies of the resource x . In principle, these string diagrams are not equivalent, but we can force them to be: in other words, we may regard traditional, ‘resource-agnostic’ syntax as a special case of diagrammatic syntax.

Within the first trend, string diagrammatic syntax has allowed for the adoption of compositional semantics for graphical formalisms that previously lacked this feature, and were usually treated in an exclusively monolithic way. This has led to the discovery of complete axiomatisations for a variety of theories, including Petri nets [16], linear dynamical systems [20, 53, 5], quantum circuits [102], and finite-state automata [95]. String diagrammatic approaches not only provide a uniform perspective on these models, they have demonstrated their ability to produce tangible outcomes. For instance, the ZX-calculus—a diagrammatic language that generalises quantum circuits—has served as the basis for the development of a quantum circuit optimisation algorithm that outperform existing methods [46].

As examples of the second trend, string diagrams have been instrumental in the development of compilers [90] for higher-order functional languages, and in a provably sound algorithm for reverse-mode automatic differentiation [4]. In both these examples, string diagrams serve as an intermediate formalism that sits between high-level programming languages and lower-level implementations. The explicit visualisation of information propagation and other structural properties of the system that they allow, while remaining syntactic objects, makes string diagrams a useful tool for compositional reasoning.

Outline This paper is an introduction to string diagrams, their syntax and their semantics. Section 2 introduces the formal syntax (for the most common variant) of string diagrams, the rules to manipulate them, and equational theories. Section 3 shows how string diagrams may be also thought of as certain (hyper)graphs, thus providing an equivalent combinatorial perspective on these objects. In Section 4, we consider other flavours of string diagrams, which correspond to a different syntax and can be manipulated in more permissive or restrictive ways. Section 5 explains how to assign semantics to string diagrams; we cover common examples of semantics and their equational properties. Section 6 contains pointers to different trends that we do not cover in details in this introduction. Finally, Section 7 is a non-exhaustive lists of applications of string diagrams, both in and outside of computer science.

2 String Diagrams as Graphical Syntax

We have seen a couple of examples, (1) and (2), of how to specify expressions of a formal language via a grammar. In order to generalise this technique to diagrammatic expressions, it is best to understand it through the lenses of abstract algebra. From an algebraic viewpoint, a grammar is a means of presenting the *signature* of our language: the list of *operations* which we may use as the

building blocks to construct more complex expressions. For instance, we may regard $;$ (program composition) as a *binary* operation, which takes two programs p and p' as arguments and returns a program $p; p'$ as value. The type of this operation is thus $program \times program \rightarrow program$. Analogously, $skip$, $x := y$ and $x := n$ may be seen as constants (operations with no arguments) of type $program$, and the while loop yields an operation of type $boolean \times program \rightarrow program$: give a boolean expression b and a program p , we obtain a program $while\ b\ do\ p$.

This example suggests that, more generally, a signature Σ should consist of two pieces of information: a set Σ_1 of operations, and a set Σ_0 of objects (e.g. *program*, *boolean*), which may be used to indicate the type of operations. Once we fix Σ , we may construct the expressions over Σ the same way we would build the valid programs out of the grammar (1). In algebra, such expressions are usually called Σ -terms.

This process works in a fairly similar way in the context of string diagrams, with some key differences. String diagrams will be built from signatures, expect that now operations may have multiple outputs as well as multiple inputs, as displayed pictorially in (3) below. We will also see that variables are not a native concept, but rather something that may encoded in the diagrammatic representation. More on this point in Example 2.16 and Remarks 2.17 and 5.19 below.

Signatures. A string diagrammatic syntax may be specified starting from a *signature* $\Sigma = (\Sigma_0, \Sigma_1)$, with a set Σ_0 of *generating objects* and a set Σ_1 of *operations*. Each operation d has a type $v \rightarrow w$, where $v \in \Sigma_0^*$ (a word on alphabet Σ_0) is called the *arity*, and $w \in \Sigma_0^*$ the *coarity* of d . Pictorially, an operation d with arity $v = a_1 \dots a_m$ and coarity $w = b_1 \dots b_n$ will be represented as

$$v \boxed{d} w = \begin{array}{c} a_1 \\ \vdots \\ a_m \end{array} \boxed{d} \begin{array}{c} b_1 \\ \vdots \\ b_n \end{array} \quad (3)$$

or simply \boxed{d} when we do not need to name the list of generating objects in the arity and coarity.

Example 2.1. $\Sigma_0 = \{x, y\}$ and $\Sigma_1 = \left\{ y \boxed{c_1} x, x \boxed{c_2} x, x \boxed{d} y \right\}$ form a signature.

Terms. Terms are generated by combining the operations of the signature in a certain way. Once again, let us look first at how terms would be specified in traditional algebra. One would start with a set Var of variables and a signature Σ of operations, and define terms inductively as:

- For each $x \in Var$, x is a term.
- For each $f \in \Sigma$, say of arity n , if t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.

For string diagrammatic syntax, terms are generated in a similar fashion, with two important differences: (i) it is a *variable-free* approach, and (ii) the way operations in Σ are combined in the inductive step depends on the richness of the graphical structure we want to express.

A standard choice for string diagrams is to rely on *symmetric monoidal* structure. This means that the operations in Σ_1 will be augmented with some ‘built-in’ operations (‘identity’, ‘symmetry’, and ‘null’), and combined via two forms of *composition* (‘sequential’ and ‘parallel’). Fixing a signature $\Sigma = (\Sigma_0, \Sigma_1)$, the Σ -terms are generated by the following derivation rules:

1. First, we have that every operation in Σ_1 yields a term:

$$\frac{}{v \boxed{d} w} \quad d \in \Sigma_1$$

2. Next, each ‘built-in’ operation (from left to right below: identity, symmetry, and null) also yields a term.

$$\frac{}{x \boxed{} x} \quad x \in \Sigma_0 \quad \frac{}{y \boxed{} x} \quad x, y \in \Sigma_0 \quad \frac{}{\boxed{}}$$

3. As for the inductive step, a new term may be built by combining two terms, either sequentially (left) or in parallel (right).

$$\frac{u \boxed{c} v \quad v \boxed{d} w}{u \boxed{c} v \boxed{d} w} \quad \frac{v_1 \boxed{d_1} w_1 \quad v_2 \boxed{d_2} w_2}{v_1 \boxed{d_1} w_1 \quad v_2 \boxed{d_2} w_2}$$

Using the composition rules, we can define identities and symmetries by induction, for arbitrary words v, w over Σ_0 :

$$vx \boxed{} vx := \frac{v \boxed{} v}{x \boxed{} x} \quad vx \boxed{} wx := \frac{v \boxed{} w}{x \boxed{} vx} \quad vx \boxed{} vx := \frac{v \boxed{} w}{wx \boxed{} v}$$

Varying the set of ‘built-in’ terms (second clause) and the ways of combining terms (last two clauses) will capture structures different from symmetric monoidal, as illustrated in Section 4 below.

Another important point: notice that *null*, the identity over the empty word is not depicted, (or depicted as the *empty diagram*), which is shown above as an empty dotted box. Furthermore, since the type of terms is a pair of words over some generating alphabet, they can have the empty word ϵ as arity or coarity. A term of type $d : \epsilon \rightarrow w$, sometimes called a *state*, has an empty left boundary

$$\boxed{d} \text{---} w$$

while a term of type $d : v \rightarrow \epsilon$, sometimes called an *effect*¹, has an empty right boundary

$$v \text{---} \boxed{d}$$

Consequently, a term of type $d : \epsilon \rightarrow \epsilon$, which is sometimes called a *scalar*, or a *closed* term, by analogy with the corresponding algebraic notion of terms containing no free variables, has no boundary at all; it is thus depicted as just a box, with no wires:

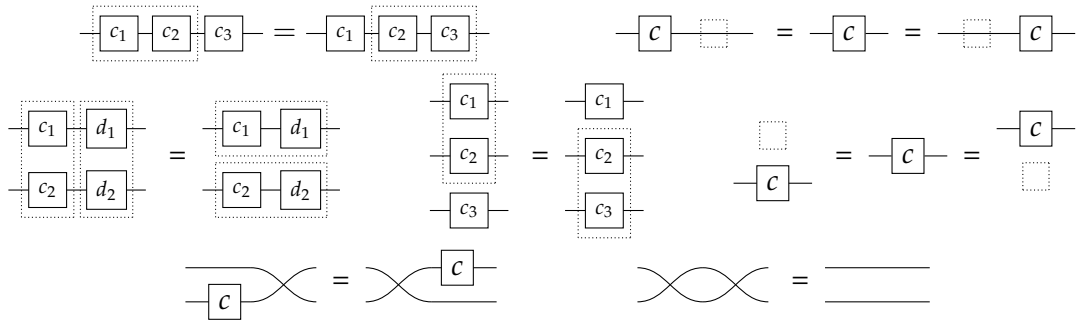
$$\boxed{d}$$

¹The names ‘state’ and ‘effect’ originated from the role played by string diagrams of this type in quantum theory [34].

String Diagrams. Terms are not quite the same as string diagrams. Indeed, as soon as we consider more elaborate terms, we realise that the above definition require us to decorate pictures with extra notation, in order to keep track of the order in which we have applied the different forms of composition. For instance, we may construct the following term from the signature in Example 2.1:


(4)

We obtain string diagrams by quotienting terms so that there is no need for dotted boxes. More formally, a string diagram on Σ is defined as an equivalence class of Σ -terms, where the quotient is taken with respect to the reflexive, symmetric and transitive closure of the following equations (where object labels are omitted for readability, and c, c_i, d_i range over Σ -terms of the appropriate arity/coarity):


(5)

If we think of the dotted frames as two-dimensional brackets, these laws tell that the specific bracketing of a term does not matter. This is similar to how, in algebra, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for an associative operation, so that we might as well write the same expression simply as $a \cdot b \cdot c$. In fact, there's an even better notation: when dealing with a single associative binary operation, we can simply forget it and write any product as a concatenation abc ! This is a simple instance of the same key insight that allows us to draw string diagrams. It is helpful to think of these diagrammatic rules as a higher-dimensional version of associativity². Particularly important is the left-most axiom on the second line: it concerns the interplay between the two forms of composition, and tells us that whichever way we choose to construct this term—by taking the parallel composition of two sequentially composed terms, or vice-versa—is irrelevant. The two equations containing wire crossings \times tell us that boxes can be pulled across wires and that wires can be entangled or disentangled as long as we do not modify how the boxes are connected.

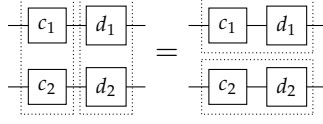
Now, we can safely remove the brackets from the term in (4) to obtain the corresponding string diagram:


(6)

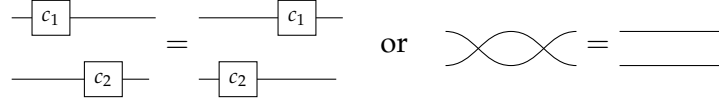
There is an important subtlety: if, formally, a string diagram is an equivalence class of terms quotiented by the laws of (5), there is not a unique way to depict a string diagram. In other words, the graphical representation (without dotted frames) sits in between terms and diagrams, as it

²In fact there is a sense in which this is precisely true, cf. Section 6.2 for a short discussion of this point.

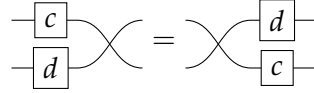
does not distinguish certain equivalent terms, *e.g.*,



Nevertheless, *the way we draw* string diagrams distinguishes some that are equivalent. The following are examples of different drawings of string diagrams that are equal under the laws of (5):



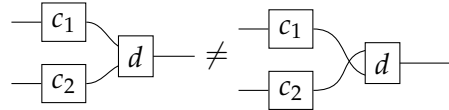
This last point is particularly salient with diagrams containing wire crossings, where the laws of (5) guarantee that any two diagrams made entirely of wire crossings over the same number of wires are equal when they define the same *permutation* of the wires. If the other rules are two-dimensional versions of associativity, the wire-crossing axioms are two-dimensional versions of commutativity. In ordinary algebra, when we have a commutative and associative binary operation, we can write products using any ordering of its elements: $abc = bac = acb$. For string diagrams, the vertical juxtaposition of boxes is not strictly commutative; nevertheless, we are allowed to move across wires, which is the next best thing:



(We invite the reader to show this, as their first exercise in diagrammatic reasoning). Furthermore, we need to keep track of how things are wired, *but* only the specific permutation of the wires matters, not how we have constructed it.

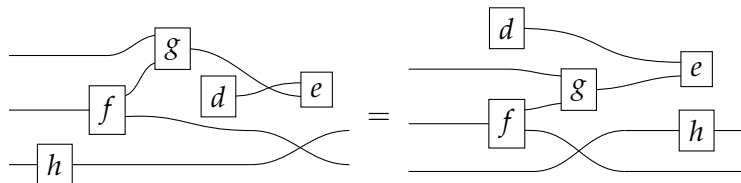
While this situation may appear slightly confusing at first, these example show that in practice the distinction between diagrams and how we depict diagrams is harmless. The topological moves that are captured by the equations of (5) are designed to be intuitive. They are often summarised by the following slogan: *only the connectivity matters*. The rule of thumb is that any deformation that preserves the connectivity between the boxes and does not require us to bend the wires backwards will give two equivalent diagrams.

Finally, keep in mind that the connection point from which we attach wires to boxes are ordered, so that the following two diagrams are *not* equivalent:



Definition 2.2 (String diagrams over Σ). *String diagrams over Σ are Σ -terms quotiented by the equations in (5).*

Example 2.3. Following the discussion above, the reader should convince themselves that the two (unframed) terms below depict the same diagram.



Syntax and Free SMC. There is a mathematically precise way to specify a syntax by defining *free* structures of a certain kind over some set of generating operations. The diagrammatic language of string diagrams comes with an associated notion of free structure: the *free symmetric monoidal category* (SMC) over a given signature.

At this point, the more mathematically-inclined reader might also object that we still have not defined rigorously what a SMC is. Somewhat circularly, we could say that a SMC is a structure in which we can interpret string diagrams! Less tautologically, it is a category with an additional operation—the monoidal product—on objects and morphisms that satisfies the laws of Fig. 5. To state them without diagrams, we need to introduce explicit notation for composition and the monoidal product. We do so in the following definition.

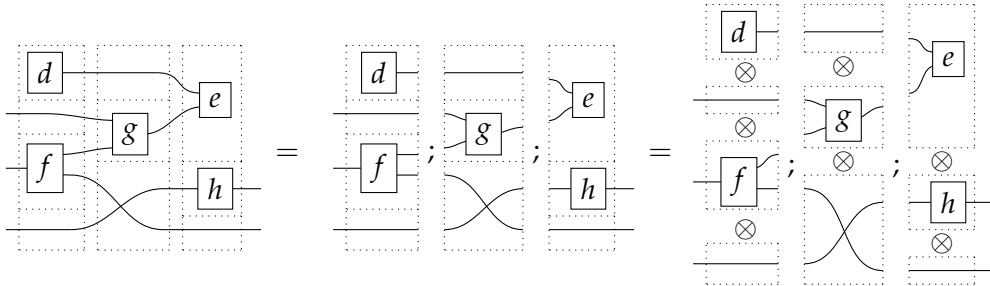
Definition 2.4 (Symmetric monoidal category). *A (strict) symmetric monoidal category (SMC) (C, \otimes, I, σ) is a category C equipped with a distinguished object I , a binary operation \otimes on objects, an operation of type $C(X_1, Y_1) \times C(X_2, Y_2) \rightarrow C(X_1 \otimes X_2, Y_1 \otimes Y_2)$ on morphisms which we also write as \otimes , and a family of morphisms σ_X^Y for any two objects X, Y , such that $id_{X \otimes Y} = id_X \otimes id_Y$ and*

$$(c_1 \otimes c_2); (d_1 \otimes d_2) = (c_1; d_1) \otimes (c_2; d_2) \quad c_1 \otimes (c_2 \otimes c_3) = (c_1 \otimes c_2) \otimes c_3 \quad id_I \otimes c = c = c \otimes id_I$$

$$(id_X \otimes c); \sigma_X^Z = \sigma_X^Y; (c \otimes id_X) \quad (\text{for any } c : Y \rightarrow Z) \quad \sigma_X^Y; \sigma_Y^X = id_X \otimes id_Y$$

Notice that the last two lines above are exactly the laws of Fig. 5 in symbolic form: we can simply replace ‘ \otimes ’ by vertical composition and ‘ $;$ ’ by horizontal composition (of compatible diagrams).

It is possible to translate any diagrams into a symbolic form. For instance, the diagram of Example 2.3 can be written as $(d \otimes id \otimes f \otimes id); (id \otimes g \otimes \sigma); (e \otimes h \otimes id)$. This expression can be obtained by successively decomposing the diagrams into horizontal and vertical layers as follows:



Like for diagrams, there are multiple ways to write a given morphism symbolic notation. In fact, because diagrams absorb into the notation some of the SMCs laws, there are usually more ways of writing a given morphism in symbolic notation than there are diagrammatic representations for it.

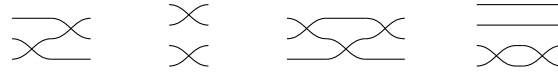
Remark 2.5 (On strictness). *The last definition is not the one that the reader is likely to encounter in the literature when looking up the terms “symmetric monoidal category”. Rather, it defines what is called a strict monoidal category; the usual notion is more general and allows for the equalities to be replaced by isomorphisms. We will not give a rigorous definition of this more general notion in this introduction and refer the reader to any standard textbook on category theory for a general introduction to SMCs [84, Chapter XI]. Our approach is nevertheless theoretically motivated by the following fundamental result: every SMC is equivalent (in a precise sense that we will not cover here in detail) to a strict SMC. This fact—known as the coherence theorem for SMCs—is what allows us to draw string diagrams. Put differently, the coherence theorem allows us to forget explicit symbols for ‘ \otimes ’ and ‘ $;$ ’, replacing them by vertical juxtaposition and*

horizontal composition without any brackets to denote the order of application³. Once again, the reader is invited to think about this as a two-dimensional generalisation of well-known facts about monoids: just like we can simply concatenate elements of a monoid and omit the symbol for the multiplication and the parentheses to bracket its application. It is then natural that more composition operations require more dimensions to represent. In fact, some of the earliest appearances of string diagrams⁴ occurred to construct free SMCs with additional structure and prove a coherence theorem for them [75, 76].

Definition 2.6 (Free SMC on a signature Σ). *The symmetric monoidal category $\text{Free}_{\text{SMC}}(\Sigma)$ is formed by letting objects be elements of Σ_0^* and morphisms be string diagrams over Σ (so Σ -terms quotiented by (5)). The monoidal product is defined as word concatenation on objects. Composition and product of string diagrams are defined respectively by sequential and parallel composition of (some arbitrary representative of each equivalence class of) Σ -terms.*

Remark 2.7. *It might appear strange that we call ‘syntax’ a free structure quotiented by some equations. This is a valid concern. The sceptical reader may be reassured to know that (1) the equivalence relation on diagrams is decidable⁵ and that (2) there is a way to see terms of this syntax as certain graphs, such that two terms are structurally equivalent precisely when they are represented by the same graph. This perspective will be developed in Section 3, but first, we turn to how we can impose more equations between diagrams.*

Example 2.8 (Free SMC over a single object). The free SMC over the signature $\Sigma = (\{\bullet\}, \emptyset)$ is easy to describe explicitly. Its diagrams are generated from horizontal and vertical compositions of \times (where we omit labels for the single generating object \bullet), modulo the laws of SMCs. Here are a few examples:



In other words, they are permutations of the wires! If we write \bullet^n for the concatenation of n bullets, a diagram $\bullet^n \rightarrow \bullet^n$ is a permutation of n elements, and there are no diagrams $\bullet^n \rightarrow \bullet^m$ for $n \neq m$.

Free SMCs on a single generating objects (and arbitrary generating morphisms) are usually called PROPs (**P**roduct and **P**ermutation categories) [83]. The PROP of permutations, which we just described, is the ‘simplest’ possible PROP — more formally, it is the initial object in the category of PROPs. Sometimes, the notion of a ‘coloured’ PROP is encountered: this is nothing but an SMC whose set of objects is freely generated (from any set of generating objects, instead of just a single generating object as in the case of ‘regular’ PROPs).

Symmetric Monoidal Functors. Whenever we define a new mathematical structure, it is a good practice to introduce a corresponding notion of mapping between them. For SMCs, this is the notion of a *symmetric monoidal functor*. We will need it when giving string diagrams an interpretation in Section 5.

Definition 2.9. *Let (C, \otimes) and (D, \boxtimes) be two SMCs with I and J their respective units, σ and θ their respective symmetries. A (strict) symmetric monoidal functor $F : (C, \otimes) \rightarrow (D, \boxtimes)$ is a mapping from objects of C to those of D that satisfies*

$$F(X_1 \otimes X_2) = F(X_1) \boxtimes F(X_2) \quad \text{and} \quad F(I) = J$$

³The coherence theorem is due to Mac Lane [84]. A recent exposition based on string diagrams can be found in [105].

⁴Though the difficulty of typesetting them at the time often meant that they did not appear as diagrams in print!

⁵This is apparent when observing that string diagrams are the same as certain hypergraphs, see Section 3 below.

and a mapping from morphisms of C to those of D that satisfies

$$F(c; d) = F(c).F(d) \quad F(id_X) = id_{F(Y)} \quad F(c_1 \otimes c_2) = F(c_1) \boxtimes F(c_2) \quad F(\sigma_X^Y) = \theta_{F(X)}^{F(Y)}$$

where we write $';$ and $'.'$ for the compositions of (C, \otimes) and (C, \boxtimes) , respectively.

In this introduction, for pedagogical reasons, we will mostly use *strict* monoidal functors, that is, functors that preserve the monoidal structure on the nose. The reader should know that it is possible, and sometimes necessary, to relax this requirement, replacing the equalities $F(X_1 \otimes X_2) = F(X_1) \boxtimes F(X_2)$ and $F(I) = J$ by *isomorphisms* (which then have to satisfy certain compatibility conditions). See [84] for a standard treatment and [98] for the connections with string diagrams.

If we have two such functors $F : (C, \otimes) \rightarrow (D, \boxtimes)$ and $G : (D, \boxtimes) \rightarrow (C, \otimes)$ that are inverses to each other— FG and GF are identity functors—we say that the two SMCs are *isomorphic* or, if the functors are not strict, *equivalent*.

Example 2.10. In Example 2.8, we saw that the morphisms/string diagrams of the free SMC over a single object looked a lot like permutations. There is a way of making this precise, by establishing an isomorphism between this SMC and another whose morphisms are permutations of finite sets. Let Bij be the category whose objects are natural numbers, and morphisms $n \rightarrow n$ are permutations of $\{1, \dots, n\}$. We can equip it with a monoidal product, given on object by addition, and on morphisms $\theta_1 : n_1 \rightarrow n_1$ and $\theta_2 : n_2 \rightarrow n_2$ by $\theta_1 \otimes \theta_2(i) = \theta_1(i)$ if $i \leq n_1$ and $\theta_1 \otimes \theta_2(i) = \theta_2(i)$ otherwise. The unit of the monoidal structure is the number 0 and the symmetry is the permutation over two elements, which we write as $\sigma : 2 \rightarrow 2$, given by $\sigma(1) = 2$ and $\sigma(2) = 1$. The isomorphism is straightforward. In one direction, let $F : \text{Free}_{\text{SMC}}(\{\bullet\}, \emptyset) \rightarrow \text{Bij}$ be given by $F(\bullet^n) = n$ on objects and $F(\times) = \sigma$. This is enough to describe F fully because all diagrams of $\text{Free}_{\text{SMC}}(\{\bullet\}, \emptyset)$ are vertical or horizontal composites of \times and F has to preserve these two forms of composition, by Definition 2.9. Furthermore, the required properties are immediately satisfied. To build its inverse, we need to know that we can factor any permutation into a composition of *adjacent transpositions* (this fact is fairly intuitive and usually covered in introductory algebra courses, so we will not prove it here). Then, notice that the transposition $(i \ i+1)$ over n elements should clearly be mapped to the diagram that is the identity everywhere and \times at the i -th and $i+1$ -th wires. Call this diagram at_i . Then, let $G : \text{Bij} \rightarrow \text{Free}_{\text{SMC}}(\{\bullet\}, \emptyset)$ be given by $G(n) = \bullet^n$ on objects and on morphisms by $G(\theta) = at_{i_1} at_{i_2} \dots at_{i_k}$ where $(i_1 \ i_1+1)(i_2 \ i_2+1) \dots (i_k \ i_k+1) = \theta$ is a decomposition of θ into adjacent transpositions. One can check that this is well-defined, and satisfies all the equations of Definition 2.9. Moreover the two are inverses of each other. For example, to see that $GF(c) = c$ it is sufficient to check that equality for \times . It holds clearly as $GF(\times) = G(12) = \times$. The other direction is a bit more lengthy, but without any major difficulties.

Thus $(\text{Bij}, +)$ gives a semantic account of the free SMC over a single object, or conversely, the latter can be seen as as diagrammatic syntax for the former.

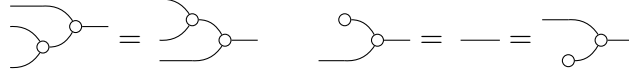
In fact, this SMC is also equivalent (in a looser sense) to the non-strict SMC of finite sets and bijections between them, with the disjoint sum as monoidal product. The equivalence is also straightforward to establish, but requires us to fix a total ordering on every finite set. Since, we stay (largely) away from non-strict SMCs in this introduction, we will not see this in more details.

This example is just a taster of an idea that we will develop further in Section 5, which is dedicated to the *semantics*—that is, the interpretation—of string diagrams.

2.1 Adding Equations

On top of the structural equivalence induced by the equations (5), we can work with additional equational theories. A *symmetric monoidal theory*—or simply *theory* when no ambiguity can arise—is given by a set of pairs of string diagrams over some signature Σ , where each component has the same arity and co-arity. The pairs it contains are interpreted as equalities—and we will usually write them as such—that we call *axioms*. A theory E is sometimes also identified with the smallest congruence relation (w.r.t. sequential and parallel composition) containing E , which we write as $\underline{\underline{E}}$.

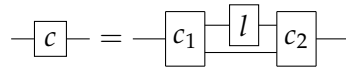
Example 2.11. Given the signature $(\{\bullet\}, \{\square, \circ\})$ (note that we can just omit wire labels when there is a single generating object), we can consider the equational theory given by the following three axioms:



The reader might have recognised this as the theory of *monoids*: the first is the diagrammatic counterpart of associativity, while the next two represent unitality. More on this theory in Example 2.15 below.

Remark 2.12. The reader familiar with abstract algebra will find exactly the same idea here. Algebraic structure, like monoids, groups, or rings can be presented axiomatically by a certain number of operations and equations. For example, one can define a monoid as a set equipped with a binary operation m and one nullary (constant) operation e satisfying associativity ($m(x, m(y, z))$) and unitality ($m(x, e) = x = m(e, x)$) axioms. In ordinary algebra, there is only one way to combine operation, namely by composing them with each other; in diagrammatic algebra, there are two ways, by composing them sequentially (taking the place of the composition of ordinary algebra) and in parallel. The latter can be compared to tuples of operations in algebra, but with significant differences (we will compare the two more closely in Remark 5.19).

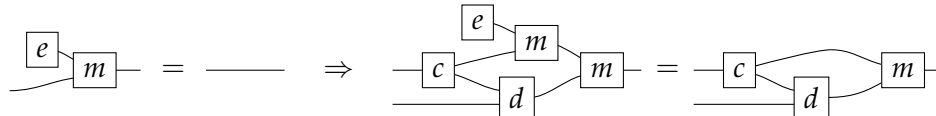
Remark 2.13 (Equations and diagrammatic rewriting.). It might be helpful to see the equations as two-ways rewriting rules that can be applied in an arbitrary context. More precisely, assume that we have some equation of the form $l = r$, where l, r have the same domain and codomain; to apply it in context, we need to identify l in a larger diagram c , i.e., find c_1 and c_2 such that



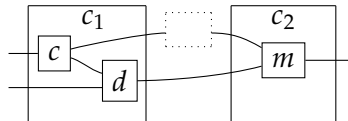
and simply replace l by r , forming the new diagram $\text{---} \boxed{c_1} \boxed{r} \boxed{c_2} \text{---}$. This is just the diagrammatic version of standard algebraic reasoning. We can summarise this process as follows:

$$\forall c_1, c_2 \quad \text{---} \boxed{l} \text{---} = \text{---} \boxed{r} \text{---} \Rightarrow \text{---} \boxed{c_1} \boxed{l} \boxed{c_2} \text{---} = \text{---} \boxed{c_1} \boxed{r} \boxed{c_2} \text{---}$$

For example,



where the context is



We will briefly come back to this point, in the context of graph-rewriting, in Section 6.1.

Free SMC over an equational theory. In the same way that string diagrams corresponded to a free structure—the free symmetric monoidal category (SMC) over Σ —quotienting by further equations also presents a free structure: given a signature Σ and a theory E we can form the free SMC $\text{Free}_{\text{SMC}}(\Sigma, E)$ obtained by quotienting the free SMC $\text{Free}_{\text{SMC}}(\Sigma)$ by the equivalence relation over diagrams given by $\stackrel{E}{=}$.

Definition 2.14 (Free SMC over a signature Σ and an equational theory E). *The symmetric monoidal category $\text{Free}_{\text{SMC}}(\Sigma, E)$ is formed by letting objects be elements of Σ_0^* and morphisms be equivalence classes of string diagrams over Σ quotiented by $\stackrel{E}{=}$. The monoidal product is defined as word concatenation on objects; composition and product of morphisms are defined respectively by sequential and parallel composition of arbitrary representatives of each equivalence class.*

2.2 Common Equational Theories

Some theories occur frequently in the literature. Many authors assume familiarity with the theories hiding behind the words “monoids”, “comonoids”, “bimonoids/bialgebras” or “Frobenius monoid/algebra”, and how all of these relate to one another. For this reason it is valuable to know them well, especially when trying to distinguish routine moves from key steps in diagrammatic proofs. This section describes a few of the most commonly found theories.

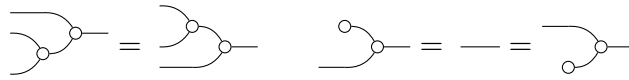
Example 2.15 (Monoids). Let us begin with the deceptively easy example of monoids. Many readers will undoubtedly be familiar with the algebraic theory of monoids, which can be presented by two generating operations, say $m(-, -)$ of arity 2 and u of arity 0 (in other words, a constant) satisfying the following three axioms:

$$m(m(x, y), z) = m(x, m(y, z)) \quad \text{and} \quad m(u, x) = x = m(x, u)$$

Analogously, the symmetric monoidal theory of monoids can be presented, as we already have in Example 2.11 above, by two generators that we write as simple white dots:



and three axioms for associativity and (two-sided) unitality:



Notice that we have just replaced variables with wires and algebraic operations with diagrammatic generators. As in ordinary algebra, two terms/diagrams are equal if they one can be obtained from the other by applying some sequence of these three equations (recall Remark 2.13).

However, this analogy hides an important subtlety. There is, in fact, a difference between the symmetric monoidal theory of monoids and the corresponding algebraic theory: if in the former the wires play the role of variables, they have to be used precisely once; unlike variables in ordinary algebra, we cannot use wires more than once or omit to use them at all! This would prevent us, for example, to specify the theory of *idempotent* monoids (monoids satisfying the additional axiom $m(x, x) = x$). This restriction—often termed *resource-sensitivity*—is an important feature of diagrammatic syntax (though there are ways to recover the resource-insensitivity of ordinary algebraic syntax, as we will see in Section 4.2.5). Of course, properties that do not involve multiple uses of variables can be specified completely analogously. For example, the theory of

commutative monoids—those monoids that satisfy $m(x, y) = m(y, x)$ —simply adds the following diagrammatic equation (note the use of the symmetry \times):

$$\text{X} = \text{X}$$

When dealing with monoids, there are several straightforward syntactic simplifications that the reader is likely to encounter in the literature. First, a simple observation: in standard algebraic syntax, the associativity axiom $m(m(a, b), c) = m(a, m(b, c))$ implies that any two ways of applying monoid multiplication to the same list of elements are all equal. Therefore it is unambiguous to introduce a generalised monoid operation for any finite arity, e.g. $m(a, b, c)$, to denote all possible ways of applying m to these three elements, and avoid a flurry of parentheses. (Note that, with this syntactic sugar, the unit e denotes the application of m to zero elements.) The same trick works for an associative $m : 2 \rightarrow 1$ diagram: one will often find generalised n -ary operations as a dot with n -many wires

$$n \left\{ \begin{array}{c} \text{---} \\ \vdots \\ \text{---} \end{array} \right\} \text{---}$$

as syntactic sugar to denote multiple applications of the binary operation of a monoid. For this reason, the reader might also encounter diagrammatic proofs that identify different ways of applying a monoid operation to the same list of elements, much like a practitioner well-versed in ordinary algebra will usually omit parentheses where they can do so unambiguously.

Example 2.16 (Comonoids). Unlike algebraic syntax, string diagrams allow for operations with co-arity different from 1, manifested by multiple (or no) right boundary ports. It is therefore possible to flip the generators and axioms of the theory of monoids, to obtain the symmetric monoidal theory of *comonoids*! Unsurprisingly, it is presented by a signature with a single object (which we therefore omit in diagrams), two generators, called comultiplication and counit,

$$\text{---} \bullet \text{---} \quad \text{---} \bullet$$

and the following three axioms:

$$\text{---} \bullet \text{---} \bullet \text{---} = \text{---} \bullet \text{---} \bullet \text{---} \quad \text{---} \bullet \text{---} = \text{---} = \text{---} \bullet \text{---}$$

called coassociativity and counitality. As one can see immediately, diagrams for comonoids are just the mirrored version of those for monoids. Therefore, any diagrammatic statement involving only comonoids can be proved by simply flipping the corresponding proof about monoids along the vertical axis. For example, as we did for monoids, it is possible to reason silently modulo coassociativity and introduce a generalised comultiplication node with co-arity n for any natural:

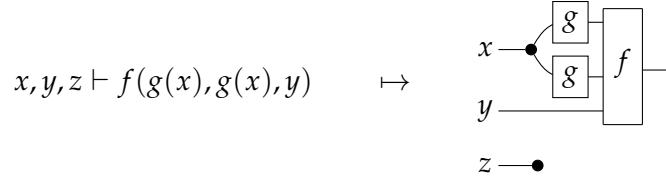
$$\text{---} \bullet \left\{ \begin{array}{c} \text{---} \\ \vdots \\ \text{---} \end{array} \right\} n$$

A comonoid is furthermore *cocommutative* if

$$\text{---} \bullet \text{---} = \text{---} \bullet \text{---}$$

As we will see, distinguished cocommutative comonoid structures play a special role in many theories: for example, they can be used to represent a form of copying and discarding, which allows us to interpret the wires of our diagrams as variables in standard algebraic syntax. The

comultiplication allows us to reference a variable multiple times and the counit gives us the right to omit some variable in a diagram. Following this intuition, we may for instance depict the term $f(g(x), g(x), y)$ in the context given by variables x, y, z , as follows:



For this reason, from the diagrammatic perspective, algebraic theories (or Lawvere theories, their categorical cousins, see [70]) always carry a chosen *commutative* comonoid structure [23], even though this structure does not appear in the usual symbolic notation for variables (which relies instead on an infinite supply of unique names to serve as identifiers for variables). We will come back to this point in Remark 5.19.

Remark 2.17 (Symmetric monoidal theories and linearity). *Much like monoids in ordinary algebra, monoids or comonoids in symmetric monoidal theories can have additional properties. We have already encountered commutative monoids and cocommutative comonoids. Note however that the resource-sensitivity of diagrams means that there are certain equations that one cannot impose on a (co)monoids in the monoidal/diagrammatic context: we have already seen that it makes no sense to refer to the diagrammatic theory of idempotent monoids. Indeed, to state the idempotency axiom (which in ordinary algebra is simply $m(x, x) = x$ with the monoid multiplication written as above) one requires the ability to duplicate variables, which in the diagrammatic syntax amounts to duplicating wires. In general, we do not have this ability, and only axioms that involve each variable precisely once on each side of the equality sign—so called linear axioms—can be imposed. As we will see, idempotency can be expressed, but in the monoidal context, is a property of a more complex algebraic structure than simply a monoid; it can be stated as a property of a bimonoid, which is our next example.*

There are several ways of mixing (the theories of) monoids and comonoids together, as the next two examples illustrate.

Example 2.18 (Co/commutative bimonoids). The theory of monoids and comonoids can interact in different ways. By ‘interact’ in this context, we mean that there are different equations that one can impose when considering a signature that contains both the generators of monoids and those of comonoids with their respective theories. One possible theory axiomatises a structure called a *bimonoid*. We will consider this structure is presented by the generators of monoids and comonoids



together with their respective axioms and the following additional four equations:

$$\begin{array}{c} \text{multiplication} \end{array} = \begin{array}{c} \text{multiplication} \end{array} \quad \begin{array}{c} \text{comultiplication} \end{array} = \begin{array}{c} \text{comultiplication} \end{array} \quad \begin{array}{c} \text{unit} \end{array} = \begin{array}{c} \text{unit} \end{array} \quad \begin{array}{c} \text{counit} \end{array} = \begin{array}{c} \text{counit} \end{array} \quad (7)$$

Intuitively, these axioms can be seen as particular cases of the same general principle: whenever one of the monoid generators meets (*i.e.*, is composed with) one of the comonoid generators, they pass through one another, producing multiple copies of each other. This is a two-dimensional form of *distributivity*. For example, when the unit meets the comultiplication, the latter duplicates the former; when the multiplication meets the comultiplication, they duplicate each other (notice how this requires the symmetry, the ability to cross wires). Using the generalised monoid

and comonoid operations introduced in the previous examples, we can formulate a generalised bimonoid axiom *scheme* that captures all four axioms (and more):

(8)

Then, the four defining axioms can be recovered for the particular cases where the number of wires on each side is zero or two.

As we have already mentioned in Example 2.16, comonoids can mimic the multiple use of variables in ordinary algebra. Thus, in the context of bimonoids, we can state ordinary equations that involve more or less than one occurrence of the same variable (see also Remark 2.17 on this point). For example, a bimonoid is idempotent when it satisfies the following additional equality, which clearly translates the usual $m(x, x) = x$ into a diagrammatic axiom:

Example 2.19 (Frobenius structures). Bimonoids are not the only way that monoids and comonoids can interact—there is another structure that frequently appear in the literature, under the name of *Frobenius structure*, *Frobenius algebra*, or *Frobenius monoid*⁶. This structure is presented by the generators of monoids and comonoids—we will write them as nodes of the same colour this time, as this is how Frobenius structures tend to appear in the literature, and will allow us to distinguish them from bimonoids in the rest of the paper:

together with their respective axioms and the following two additional axioms, both called Frobenius' law:

(9)

These axioms provide an alternative way for the multiplication and comultiplication of the monoid and comonoid structures to interact: unlike the case of bimonoids, this time they do not duplicate each other, but simply slide past one another, on either side. This is a fundamental difference which, in fact, turns out to be incompatible with the bimonoid axioms in an interesting way that we will examine later in this paper (see Section 4.2.9).

This seemingly simple law has important consequences. At first, the string diagram novice may find it difficult to internalise all the laws that make up a given equational theory; when proving some equality, it is not always clear which law to apply at which point to reach the desired goal and it is easy to get overwhelmed by all the choices available. However, in some nice cases, as we saw for (co)monoids, there are high level principles that allow us to simplify reasoning and see more clearly the key steps ahead. For example, reasoning up to associativity becomes second nature after enough practice and one no longer sees two different composites of (co)multiplication as different objects.

In the same way, the Frobenius law can be thought of as a form of two-dimensional associativity; it simplifies reasoning about complex composites of monoid and comonoid operations even further and allows us to identify easily when any two of these are equal. To see this, it is helpful

⁶The term ‘algebra’ usually refers to a monoid which is also a vector space (and whose multiplication is a linear map), as this is the context in which Frobenius algebras were first studied.

to think of diagrams as (undirected) graphs, whose vertices are any of the boxes (m , u , c or v in the nomenclature above) and edges are wires. We say that a diagram is *connected* if there is a path between any two vertices in its corresponding graph. It turns out that, for Frobenius structures, any connected diagram composed out of (finitely many) m , u , c , and v (without wire crossings), using vertical or horizontal composition is equal to one of the following form:

$$n \left\{ \begin{array}{c} \text{diagram with } k \text{ loops} \\ \vdots \\ \text{diagram with } k \text{ loops} \end{array} \right\} m$$

where we use ellipsis to represent an arbitrarily large composite following the same pattern. In other words, the only relevant structure for a (connected) diagram in the theory of Frobenius structures is the number of left and right wires it has, and how many paths there are from any left leg to any right leg (how many loops it has in the normal form depicted above). This theorem is sometimes called the *spider theorem* and justifies introducing generalised vertices or spider nodes as syntactic sugar:

$$\begin{array}{c} k \\ \text{---} \bullet \text{---} \\ n \quad m \end{array}$$

where the natural number k represents the number of inner loops in the normal form above. All the laws of Frobenius structures can now be summarised into a single convenient axiom scheme:

$$\begin{array}{c} n_2 - k \\ \left\{ \begin{array}{c} \text{diagram with } k_2 \text{ loops} \\ \vdots \\ \text{diagram with } k_2 \text{ loops} \end{array} \right\} m_2 \\ n_1 \left\{ \begin{array}{c} \text{diagram with } k_1 \text{ loops} \\ \vdots \\ \text{diagram with } k_1 \text{ loops} \end{array} \right\} m_1 - k \end{array} = \begin{array}{c} k + k_1 + k_2 \\ \left\{ \begin{array}{c} \text{diagram with } k + k_1 + k_2 \text{ loops} \\ \vdots \\ \text{diagram with } k + k_1 + k_2 \text{ loops} \end{array} \right\} m_1 + m_2 - k \end{array}$$

where m is the number of middle wires that connect the two spiders on the left hand side of the equality⁷. Only having to keep track of the number of open wires and loops in a diagrams greatly reduces the mental load on the mathematician that reasons about this equational theory.

The reader will encounter other special cases of Frobenius structures in the literature. When a Frobenius structure satisfies the following idempotency law they are called *special* (or sometimes *separable*):

$$\begin{array}{c} \text{---} \bullet \text{---} \\ \text{---} \bullet \text{---} \end{array} = \text{---}$$

The normal form above then simplifies even further, since we can now forget about the inner loops:

$$\begin{array}{c} \text{---} \bullet \text{---} \\ \text{---} \bullet \text{---} \end{array} m := n \left\{ \begin{array}{c} \text{diagram with } k \text{ loops} \\ \vdots \\ \text{diagram with } k \text{ loops} \end{array} \right\} m$$

In this case, the only relevant structure of any connected diagram in the theory of special Frobenius structures is the number of its left and right wires. We can thus introduce the same syntactic sugar, omitting the number of loops above the spider, and the spider fusion scheme also simplifies further, as we no longer need to keep track of the number of legs that connect the two fusing spiders.

⁷This rule can be formalised in a number of different ways but we intentionally present it informally here to give the intuition behind how one might reason more easily about Frobenius structures.

Example 2.20 (Special and commutative Frobenius structures). The commutative and special Frobenius structures are very common in the literature, as they are an algebraic structure one finds naturally in most languages of relations (as we will see when we study semantics of string diagrams in Section 5). We summarise below the full equational theory for future reference:

In what follows, we will refer to this theory as *scFrob*.

In the commutative case, the spider theorem also holds and includes diagrams composed out of (finitely many) of $\text{---}\bullet\text{---}$, $\text{---}\bullet\bullet\text{---}$, $\text{---}\bullet\bullet\bullet\text{---}$ and *wire crossings*, using vertical or horizontal composition. Then any diagram in the free SMC over the theory of commutative and special Frobenius structures is a fully determined by a list of spiders s_1, \dots, s_k and where each of their respective legs are connected. In other words, diagrams with n left wires and m right wires are in one-to-one correspondence with maps $n + m \rightarrow k$ for some k . We will come back to this in Example 5.11.

A special Frobenius structure that moreover satisfies the following axiom is sometimes called *extra-special*

$$\bullet\text{---}\bullet = \square$$

This means that we can forget about network of black nodes without any dangling wires—they can always be eliminated. In this case, diagrams with n left wires and m right wires in the (free SMC over the) theory of a commutative extra-special Frobenius structure are in one-to-one correspondence with partitions of $\{1, \dots, n + m\}$. We will come back to this in Example 5.12.

...and beyond. The reader will frequently encounter these theories—monoids, comonoids, bimonoids, Frobenius structures, potentially with additional properties—as the basic building blocks of a number of more complex diagrammatic calculi that axiomatise different algebraic structures. For example, the ZX-calculus, a presentation of a diagrammatic calculus that generalises quantum circuits (see Section 7), contains not one, but two special commutative Frobenius structures (often denoted by a red and green dot respectively) that interact together to form two bimonoids: the green monoid with the red comonoid forms a bimonoid and so do the red monoid with the green comonoid. Phew! At first, this seems like a lot of structure to absorb, but quickly, one learns to use the spider theorem to think of monochromatic diagrams so that most of the complexity comes from the interaction of the two colours. In fact, modern presentations of the ZX calculus prefer to give the equational theory using spiders of arbitrary arity and co-arity as generators and the spider fusion rules as axioms. Strikingly, very similar equational theories can be found ubiquitously in a number of different applications, across different fields of science: it appears there is something fundamental to the interaction of monoid-comonoid pairs in the way we model computational phenomena. We will expand on this point in Section 7.

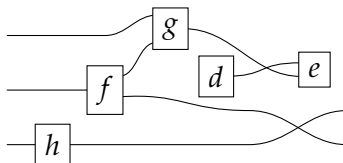
Remark 2.21 (Distributive Laws). *Interestingly, both the equations of bimonoids (Example 2.18) and of Frobenius monoids (Example 2.19) describe the interaction of a monoid and a comonoid. As mentioned, such interaction may be described as a factorisation: the bimonoid laws allow us to factorise any string diagram*

as one where all the comonoid generators precede the monoid generators, as in (8); dually, the Frobenius laws yield a monoid-followed-by-comonoid factorisation. More abstractly, both equational theories can be described as the specification of a distributive law involving the monoid and the comonoid. Distributive laws are a familiar concept in textbook algebra: the chief example is the one of a ring, whose equations describe the distributivity of a monoid over an abelian group. In the context of monoidal theories, distributive laws are even more powerful, as they can be used to study the interaction of theories with generators with arbitrary coarity, such as comonoids, Frobenius monoids, etc. The systematic study of distributive laws of monoidal theories has been initiated by Lack [80], and expanded in more recent works [109, 22, 23]. Understanding an equational theory as a distributive law allows us to obtain a factorisation result for its string diagrams, such as (8). Moreover, it provides insights on a more concrete representation (a semantics) for syntactically specified theories of string diagrams - a theme which we will explore in Section 5. For example, the phase-free fragment of the aforementioned ZX-calculus can be understood in terms of a distributive law between two bimonoids. This observation is instrumental in showing that the free model of the phase-free ZX-calculus is a category of linear subspaces [19]. We refer to [109] for a more systematic introduction on distributive laws of monoidal theories, as well as on other ways of combining together equational theories of string diagrams.

3 String Diagrams as Graphs

The previous section introduced string diagrams as a *syntax*. However, a strength of the formalism is that diagram may be also treated as some kind of *graphs*, with nodes and edges. This perspective is often convenient to investigate properties of string diagrams having to do with their combinatorial rather than syntactic structure. Another important reason to explore a combinatorial perspective to string diagrams is that their graph representation ‘absorbs’ the structural laws of symmetric monoidal categories, and it is thus more adapted than the term representation for certain computation tasks, such as rewriting (see Section 6.1 below). Therefore, the goal of this section is to illustrate graph interpretations of string diagrams.

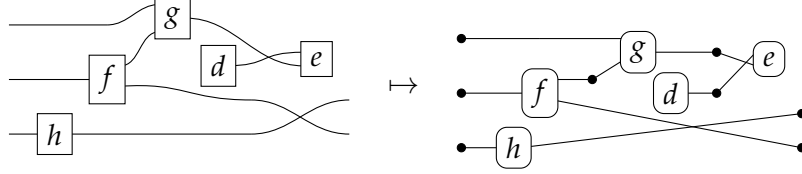
As a starting point, let us take for example a diagram we have previously considered:



If we forget about the term structure that underpins this representation, and try to understand it as a graph-like structure, the seemingly most natural approach is to think of boxes as *nodes* and the wires as *edges* of a graph. In fact, this is usually the intended interpretation adopted in the early days of usage of string diagrams as mathematical notation, see e.g. [73]. An immediate challenge for such approach is that ‘vanilla’ graphs do not suffice: string diagrams present loose, open-ended edges, which only connect to a node on one side — or even on no side, as for instance the graph representation of the ‘identity’ wires: $\frac{x}{\text{---}}$. Historically, a solution to this problem has been to consider as interpretation a more sophisticated notion of graph, endowed with a topology allowing to define when edges are ‘loose’, ‘half-loose’, ‘pinned’... see [73]. Another, more recent approach has been to understand string diagrams as graphs with two sorts of nodes, where the second sort just plays the bureaucratic role of closing edges that are drawn with a loose end [44].

The approach we present, which follows [13], does not regard boxes as nodes, but rather as *hyperedges*: edges that connect *lists* of nodes, instead of individual nodes. This perspective allows

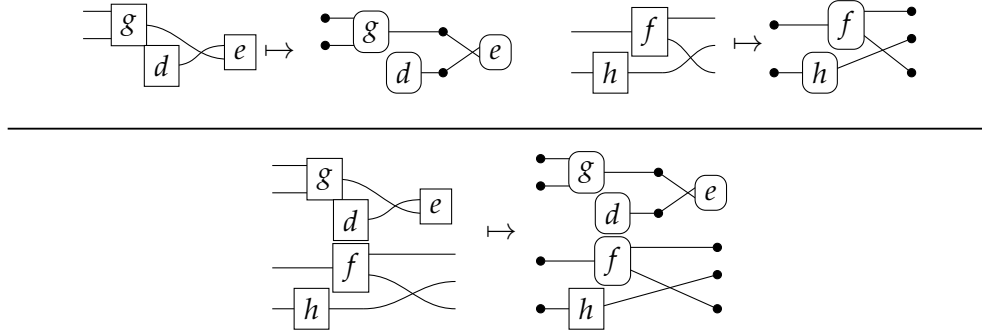
us to work with a well-known data structure (simpler than the ones above) called a *hypergraph*: the only entities appearing in a hypergraph are hyperedges—which interpret the boxes of the string diagram—and nodes—which now interpret exclusively the loose ends of wires in the string diagram. And the wires themselves? They are simply a depiction of how hyperedges connect with the associated nodes. Such an interpretation is best displayed in the previous example:



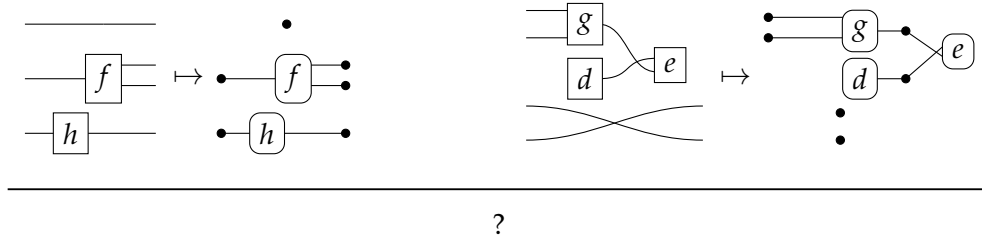
Note that, even though they are seemingly very close in shape, the two entities displayed above are of a very different nature. The one on the left is a syntactic object: the string diagram representing some term modulo the laws of SMCs. The one on the right is a combinatorial object: a hypergraph, with nodes indicated as dots and hyperedges indicated as boxes with round corners, labeled with Σ -operations.

Note that all of this section generalises to more than one generating object, by labelling vertices by the appropriate object and disallowing merging of vertices with different labels. We have preferred to keep things simpler here by treating only the case of a single generating object, as the general case poses no significant conceptual difficulty.

In order to turn this mapping into a formal interpretation, we need an understanding of how to handle composition of string diagrams. Intuitively, parallel composition is simple: if we stack one hypergraph over the other, we still obtain a valid hypergraph. For instance:

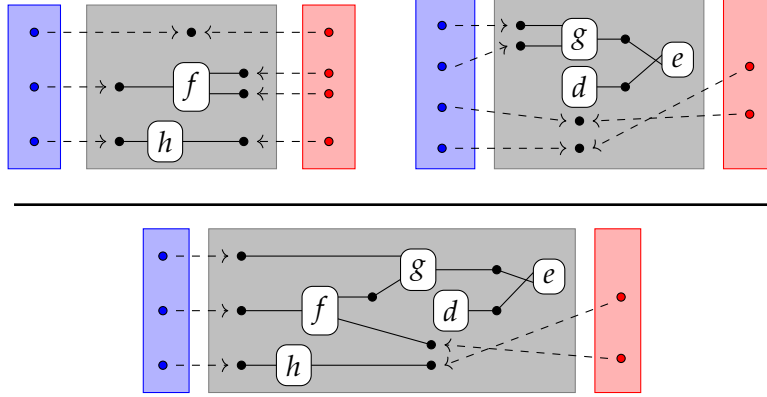


Sequential composition is subtler, as we need to formally specify how loose wires of one diagram are ‘plugged in’ loose wires of another diagram.

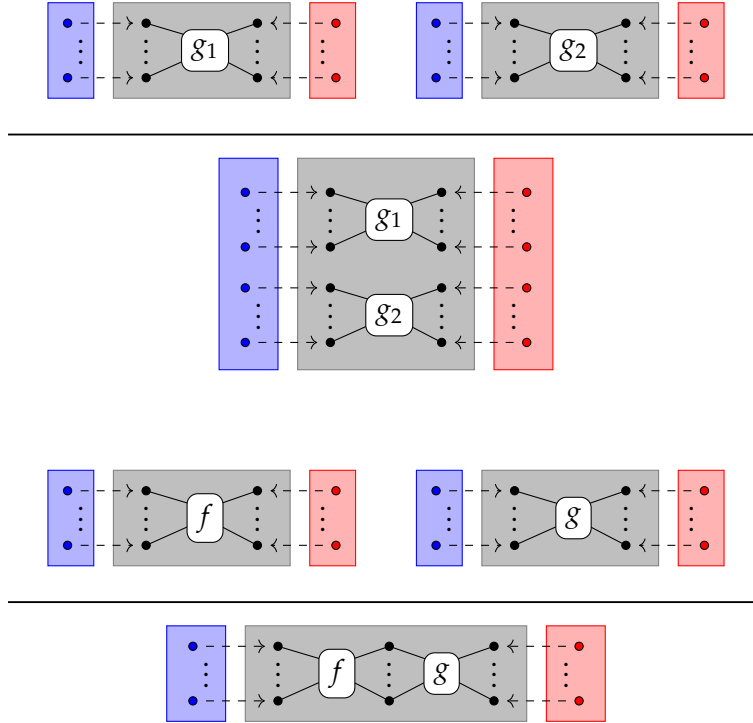
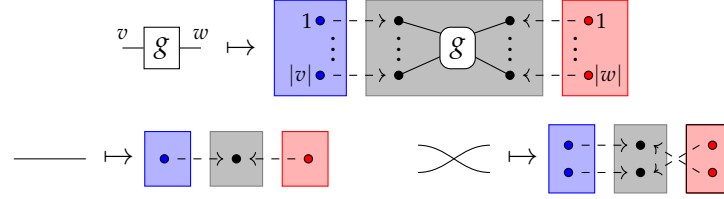


A proper definition of this composition operation is what leads to the notion of *open* hypergraph: a hypergraph with a special indication of what nodes form its *left* interface (the blue nodes below) and what nodes form its *right* interface (the red nodes below). Note that one node can be in both, as

illustrated below. Thanks to this additional information, open hypergraph come endowed with a built-in notion of sequential composition, mimicking the analogous operation on string diagrams:



Equipped with this notion, we may define the interpretation of string diagram as open hypergraphs *inductively* on Σ -terms.

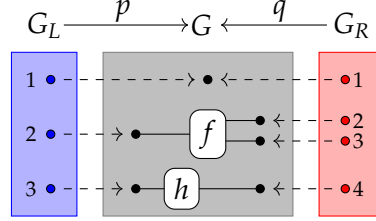


(11)

In words, the vertical composition takes the disjoint sum of each of the interfaces and hypergraphs, while the horizontal composition identifies the middle interface labels and includes them

as nodes into the composite hypergraph. Note this definition extends to an interpretation of string diagrams by verifying that it respects equality modulo the laws of SMCs—that is, if two Σ -terms are represented by the same string diagram, then they are mapped to the same open hypergraph.

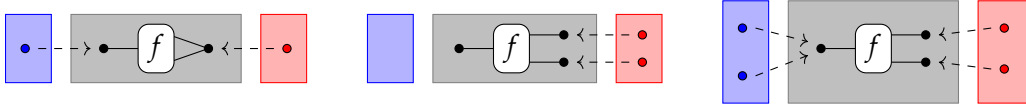
As a side note, it is significant that moving from hypergraphs to open hypergraphs does not force us to overcomplicate the notion of graph at hand, for instance by adding a different sort of nodes. From a mathematical viewpoint, a open hypergraph G may be simply expressed as a structure consisting of two hypergraph homomorphisms $\langle p: G_L \rightarrow G, q: G_R \rightarrow G \rangle$, where G_L and G_R are *discrete*⁸ hypergraphs, and the image $p[G_L]$ (respectively, $q[G_R]$) identify the nodes in the left (right) interface of G . For instance:



The structure $\langle p: G_L \rightarrow G, q: G_R \rightarrow G \rangle$ is often called a *cospan* of hypergraphs (see Example 5.11 on the simpler cospans of sets), with carrier G . When referring to G as an open hypergraph, we always implicitly refer to G together to one such cospan structure. Reasoning with cospans is mostly convenient as they come with a built-in notion of composition (by ‘pushout’ in the category of hypergraphs) which is exactly how the informal composition of open hypergraphs given above is formally defined. We will come back to this point in Section 6.1, as it plays a role in how we *rewrite* with string diagrams.

The interpretation of string diagrams as open hypergraphs given in (11) above defines a monoidal functor $\llbracket - \rrbracket$ from the free SMC over signature Σ to the SMC of cospans of hypergraphs.

An important question stemming from the interpretation (11) is to what extent the syntactic and the combinatorial perspective on string diagrams are interchangeable. First, one may show that $\llbracket - \rrbracket$ is an *injective* mapping: string diagrams that are distinct (modulo the laws of SMCs) are mapped to distinct open hypergraphs. However, it is clearly not surjective. Here are some examples of open hypergraphs over a signature Σ which are not the interpretation of any Σ -string diagram.



These examples have something in common: nodes are allowed to behave more freely than in the image of interpretation (11). For instance, in the first hypergraph there is an ‘internal’ node (not on the interface) that has multiple outgoing links to hyperedges. In the second hypergraph, there is an internal node that has no incoming links. Finally, the third hypergraph features a node that can be plugged in twice on the left interface—meaning that, when composing with another hypergraph on the left, it will have two incoming links.

We can prove that such features are forbidden in the image of $\llbracket - \rrbracket$. The property that disallows them is called *monogamy* [13].

Definition 3.1 (Degree of a node). *The in-degree of a node v in a hypergraph G is the number of pairs (h, i) where h is a hyperedge with v as its i -th target. Similarly, the out-degree of v in G is the number of pairs (h, i) where h is a hyperedge with v as its i -th source.*

⁸Recall that a graph is discrete when it has just nodes and no (hyper)edge.

Definition 3.2 (Monogamy). An open hypergraph $m \xrightarrow{f} G \xleftarrow{g} n$ is monogamous if f and g are injective and for all nodes v of G

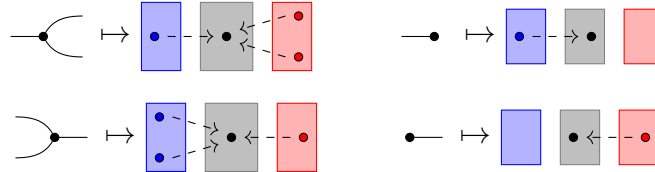
- the in-degree of v is 0 if v is in the image of f and 1 otherwise;
- the out-degree of v is 0 if v is in the image of g and 1 otherwise.

Theorem 3.3. An open hypergraph is in the image of $\llbracket - \rrbracket$ if and only if it is monogamous.

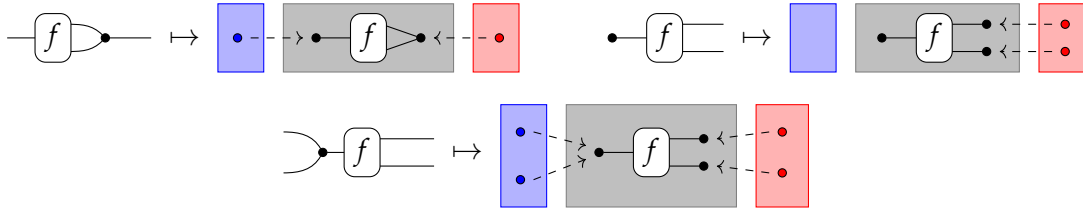
Corollary 3.4. Σ -string diagrams are in 1-1 correspondence with Σ -labeled monogamous open hypergraphs.

Theorem 3.3 settles the question of what kind of open hypergraphs correspond to ‘syntactically generated’ string diagrams. We may also ask the converse question: what do we need to add to the algebraic specification of string diagrams in order to capture all the open hypergraphs?

Remarkably, the commutative and special *Frobenius structure* from Example 2.20 is tailored for the role. Indeed we can give a special interpretation to the generators of Example 2.20, as discrete open hypergraphs:



Intuitively, the Frobenius generators are modelling the possibility that a nodes has multiple or no ingoing/outgoing links, just as in the above examples. If we now consider Σ -string diagrams augmented with the Frobenius generators, we can infer the string diagrams for the above open hypergraphs:



These observations generalise to the following result, thus completing the picture of the correspondence between string diagrams and open hypergraphs.

We write $\Sigma + \text{scFrob}$ for the signature given by the disjoint union of the generators of Σ and of scFrob , the theory given in (10).

Theorem 3.5. String diagrams on the signature $\Sigma + \text{scFrob}$ modulo the axioms of special commutative Frobenius structures given in (10) are in 1-1 correspondence with Σ -labeled open hypergraphs.

Note that it is not just the signature: also the Frobenius equations play a role in the result, as they model precisely equivalence of open hypergraphs.

Remark 3.6. Given a signature $\Sigma = (\Sigma_0, \Sigma_1)$, Open hypergraph with Σ_0 -labeled nodes and Σ_1 -labeled hyperedges form a symmetric monoidal category Hyp_Σ , whose morphisms are hypergraph homomorphisms respecting the labels. The monogamous open hypergraphs form a subcategory MHyp_Σ of Hyp_Σ . One may phrase Theorem 3.3 and Theorem 3.5 in terms of these categories, by saying that there is an isomorphism between $\text{Free}_{\text{SMC}}(\Sigma)$ and MHyp_Σ , and an isomorphism between $\text{Free}_{\text{SMC}}(\Sigma + \text{scFrob})$ and Hyp_Σ .

4 Categories of String Diagrams

Manipulating string diagrams can be confusing to the newcomer because there are many flavours, each of which authorises or forbids different topological deformations and manipulations. To make matters worse, many papers will assume that the reader is comfortable with the rules of the game for the authors' specific flavour, and gloss over the basic transformations. This is not necessarily a bad thing, as the point of string diagrams is to serve as a useful computational tool, a syntax that empowers its users by absorbing irrelevant details into the topology of the diagrams themselves. This section is here to convey the basic rules for the most common forms one is likely to encounter in the literature. We will give some intuition about the manipulations that are authorised and those that are forbidden in each context, illustrating them through several examples.

In previous sections, we made the conscious choice of starting with string diagrams for *symmetric monoidal categories*. Let us recall the rules of the game briefly: we were allowed to compose boxes horizontally, as long as the types of the wires matched, and vertically, without restriction. In addition, we were allowed to cross wires however we wanted, and the only relevant structure of an arbitrary vertical or horizontal composite of multiple wire-crossings is the resulting permutation of the wires that it defines.

We will now see that there are various ways of strengthening or weakening these rules and the kinds of diagrams we have at our disposal. These examples, and more, are all covered in detail in Selinger's extensive survey of diagrammatic languages for monoidal categories [98].

4.1 Fewer Structural Laws

4.1.1 Monoidal Categories What if we take away the ability to cross wires? This means in particular that we no longer have the diagram depicted as a crossing of two wires at our disposal.

Terms of the free monoidal category over a chosen signature $\Sigma = (\Sigma_0, \Sigma_1)$ are generated by the following derivation rules:

$$\begin{array}{c}
 \frac{}{x} \quad x \in \Sigma_0 \qquad \frac{}{v \boxed{d} w} \quad d \in \Sigma_1 \\
 \\
 \frac{u \boxed{c} v \quad v \boxed{d} w}{u \boxed{c} \boxed{d} w} \qquad \frac{v_1 \boxed{d_1} w_1 \quad v_2 \boxed{d_2} w_2}{v_1 \boxed{d_1} w_1 \quad v_2 \boxed{d_2} w_2}
 \end{array}$$

We consider two terms structurally equivalent when they can be obtained from one another using the axioms of monoidal category, *i.e.* the strict subset of those of symmetric monoidal categories that does not involve the symmetry/wire-crossing, as found in (5). For example, we still have the interchange law

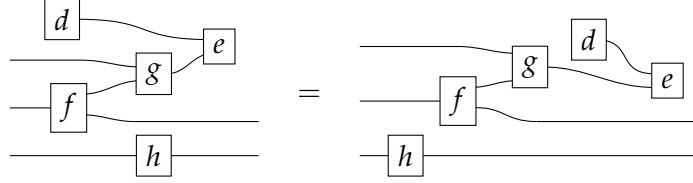
$$\begin{array}{c}
 \boxed{c_1} \boxed{d_1} \\
 \boxed{c_2} \boxed{d_2}
 \end{array}
 =
 \begin{array}{c}
 \boxed{c_1} \boxed{d_1} \\
 \boxed{c_2} \boxed{d_2}
 \end{array}$$

but we do not have $\bowtie_{y'}^x \equiv \text{---}_y^x$, as \bowtie is not even a term of our syntax anymore.

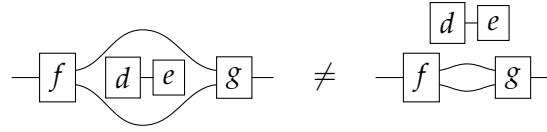
Intuitively, they are the plane cousins of their symmetric counterpart, *i.e.* the subset of string diagrams we can draw in the plane in a symmetric monoidal category without crossing any wires. In a way, the rules are much simpler: we can only compose diagrams horizontally (with the usual

caveat that the right ports of the first have to match the left ports of the second) and vertically. That's it.

Two such diagrams are equivalent if one can be deformed into the other *without any intermediary steps that involve crossing wires*. For example,



The last caveat is important, as two monoidal diagrams could be equivalent if we interpreted them (via the obvious embedding) as symmetric monoidal diagrams, but not equivalent as monoidal diagrams. This is the case for the two below:



This example is typical of the problem. In the monoidal case, certain diagrams can be trapped between some wires, without any way to move them on either side (where, in the symmetric monoidal case, we could have just pulled the middle diagram out, past the surrounding wires).

4.1.2 Braided Monoidal Categories These [72] are one step up from monoidal categories, but are not symmetric. They allow a form of wire crossing that keeps track of which wire goes over which. To this effect, we introduce a *braiding* for each of the two possibilities depicted suggestively as follows:



Term formation rules for the free braided monoidal category over a given signature Σ are those of monoidal categories plus the following two:

$$\frac{}{x \text{ } y} \quad x, y \in \text{Obj}(\Sigma) \qquad \frac{}{y \text{ } x} \quad x, y \in \text{Obj}(\Sigma)$$

The notion of structural equivalence is up to the axioms of braided monoidal categories, which we now give:

$$\begin{array}{c} \frac{x \text{ } y}{y \text{ } x} \text{ Br} \equiv \frac{x}{y} \\ \frac{x \text{ } y}{z \text{ } x} \text{ YB} \equiv \frac{x \text{ } z}{y \text{ } x} \end{array}$$

As the drawings suggest, the intuition for the braiding is that diagrams inhabit a three-dimensional space in which we are free to cross wires by moving them over or under each other. The first laws states that the two braidings are inverses and the second—called the *Yang-Baxter equation* [29]—is an instance of the naturality of the braiding. Notice that the braiding satisfies structural laws that are similar to the symmetry in a symmetric monoidal category except that it is not self-inverse: $\text{X}^x_y \equiv \text{X}^x_y$ does not hold if we take X^y_x instead (we can intuitively see why: we obtain a twist of the wires).

As a result, we can draw any diagram we could draw in a symmetric monoidal category, but we have to pick which wire goes under and which goes over for each crossing.

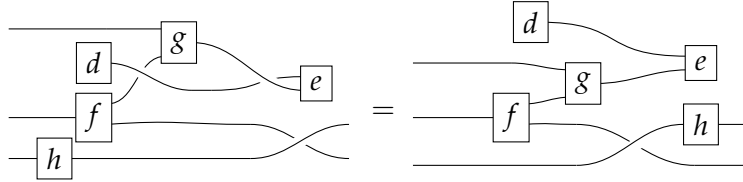
Two diagrams are equivalent if they can be deformed into each other without ever moving two wires through one another to magically disentangle them. Once more, this gives an equivalence that is finer⁹ than that of symmetric monoidal categories. A simple illustrative example of this phenomenon is the following twisting:



If we replaced the two braidings by the symmetry to obtain a term of a symmetric monoidal category, this diagram would simply be the identity ---^x_y . This serves as a reminder that the braiding is not self-inverse. One can find much more interesting examples—in fact, we can draw arbitrary braids:



or diagrams containing other generators with arbitrary braidings between them, which can be transformed like those of symmetric monoidal categories, as long as we do not move any of the wires through one another:



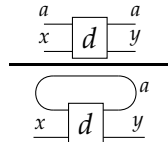
Remark 4.1. *Contrary to the case of symmetric monoidal categories (see Section 3), there is no known representation of string diagrams for braided monoidal categories as graphs.*

4.2 More Structural Laws

Just like we can weaken the structure of symmetric monoidal categories and draw more restricted diagrams, we can also extend our diagrammatic powers. The following is a non-exhaustive list of the most common variations one might find in the literature.

4.2.1 Traced Monoidal Categories Diagrams in a (symmetric) monoidal category keep to a strict discipline of acyclicity: we can only connect the right and left ports of two boxes. One could imagine relaxing this requirement, while keeping a clear correspondence between left ports as inputs and right ports as outputs.

Terms formation rules for traced monoidal categories are those of symmetric monoidal categories with the addition of the following operation, called the *partial trace*¹⁰:



⁹Of course this sentence does not make sense formally, because the terms of free braided and symmetric monoidal categories are different, but we can map those of the former to the latter by sending the braid to the symmetry—we then mean that this mapping is not injective, *i.e.*, that two different braided monoidal diagrams may be mapped to the same symmetric monoidal diagram.

¹⁰The name comes from the usual linear algebraic notion of trace. We will examine this concrete case in Example 5.13.

The corresponding notion of structural equivalence is given by the following axioms (with object labels removed for clarity).

- Vanishing:

$$\begin{array}{c} \text{---} \boxed{d} \text{---} \end{array} \overset{\epsilon}{=} \text{---} \boxed{d} \text{---} \quad \quad \quad \begin{array}{c} \text{---} \boxed{d} \text{---} \end{array} = \begin{array}{c} \text{---} \boxed{d} \text{---} \end{array}$$

- Superposing:

$$\begin{array}{c} \text{---} \boxed{d} \text{---} \end{array} = \begin{array}{c} \text{---} \boxed{d} \text{---} \end{array}$$

- Yanking:

$$\begin{array}{c} \text{---} \end{array} = \text{---}$$

(12)

- Tightening:

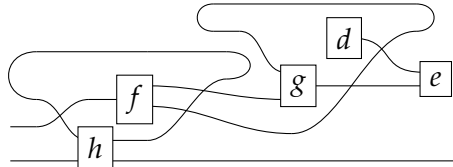
$$\begin{array}{c} \text{---} \boxed{c} \boxed{d} \text{---} \end{array} = \begin{array}{c} \text{---} \boxed{c} \boxed{d} \text{---} \end{array} \quad \quad \quad \begin{array}{c} \text{---} \boxed{d} \boxed{c} \text{---} \end{array} = \begin{array}{c} \text{---} \boxed{d} \boxed{c} \text{---} \end{array}$$

- Sliding:

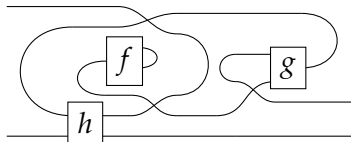
$$\begin{array}{c} \text{---} \boxed{d} \boxed{c} \text{---} \end{array} = \begin{array}{c} \text{---} \boxed{c} \boxed{d} \text{---} \end{array}$$

Here, we have had to briefly go back to using dotted frames, because the axioms of traced monoidal categories are almost diagrammatic tautologies (and this is the point of adopting a diagrammatic notation for these).

In short, diagrams for traced monoidal categories include those of symmetric monoidal ones, but add the possibility of *connecting any right port of any diagram to any left port of any other*, as in the following example:



In essence, we have the ability to draw loops, breaking free from the acyclicity requirement of plain symmetric monoidal diagrams. However, we cannot bend wires arbitrarily, or connect right ports to left ports. For example, the following is *not* allowed:



(Not to worry, we will soon introduce a syntax for which this kind of diagrams is allowed.)

The reader should convince themselves that the diagram above is equivalent to the one on the right below:

$$\begin{array}{c} \text{---} \boxed{f} \boxed{g} \boxed{d} \text{---} \end{array} \quad \quad \quad \begin{array}{c} \text{---} \boxed{f} \boxed{g} \text{---} \end{array} \quad \quad \quad \begin{array}{c} \text{---} \boxed{d} \text{---} \end{array}$$

The trick—as for symmetric monoidal diagrams—to check this equality is in verifying that the connectivity of the different boxes is preserved.

Remark 4.2. *It is also natural to consider trace-like operations that do not satisfy the yanking axiom (12). This makes sense when the trace-like operation is intended to represent a form of feedback which introduces a temporal delay. Examples abound in the theory of automata. Note that the sliding rule might also fail in this case. The associated graphical language generalises that of traced categories, and the associated structure is sometimes called a delayed or guarded traced category, or a category with feedback [74, 43, 42].*

4.2.2 Compact Closed Categories

Compact closed categories are special cases of traced monoidal categories where, rather than adding a global trace operation, we add ways of moving ports from left to right and vice-versa, using extra generators that represent wire bendings directly, as operations of the signature.

The term formation rules are the same as those of symmetric monoidal categories, with the following additions:

$$\begin{array}{cccc} \xrightarrow{x} & x \in \Sigma_0 & \xleftarrow{x} & x \in \Sigma_0 \\ \text{---} & & \text{---} & \\ \text{---} & & \text{---} & \end{array} \quad \begin{array}{cc} \xrightarrow{x} & x \in \Sigma_0 \\ \text{---} & \\ \text{---} & \end{array} \quad \begin{array}{cc} \xrightarrow{x} & x \in \Sigma_0 \\ \text{---} & \\ \text{---} & \end{array} \quad \begin{array}{cc} \xrightarrow{x} & x \in \Sigma_0 \\ \text{---} & \\ \text{---} & \end{array}$$

In words, we introduce an object x^* (called the dual of x) for every object x and write \xrightarrow{x} for the identity on x and \xleftarrow{x} for the identity on x^* . The objects x and x^* are related by two wire-bending diagrams \xrightarrow{x} and \xleftarrow{x} , called cup and cap respectively.

The corresponding notion of structural equivalence is defined by the axioms of symmetric monoidal category and the following two axioms, which capture the duality between inputs and outputs:

$$\begin{array}{c} x \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad \begin{array}{c} x \\ \text{---} \\ \text{---} \\ \text{---} \end{array} = \begin{array}{c} x \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad \begin{array}{c} x \\ \text{---} \\ \text{---} \\ \text{---} \end{array} = \begin{array}{c} x \\ \text{---} \\ \text{---} \\ \text{---} \end{array}$$

Using the symmetry, we can define two other cups and caps, bending wires in the other direction, which we write as:

$$\text{---} \quad \text{---} \quad \text{---} \quad \text{---} \quad \text{---} \quad \text{---}$$

In this way, we also obtain a partial trace operation given simply by

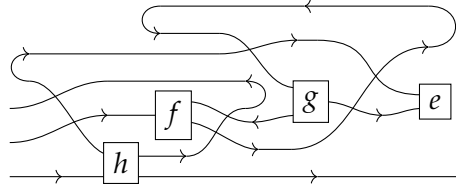
$$\begin{array}{c} a \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad \begin{array}{c} a \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad \begin{array}{c} a \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad \begin{array}{c} a \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \quad \begin{array}{c} a \\ \text{---} \\ \text{---} \\ \text{---} \end{array}$$

This operation satisfies all the required axioms of traced monoidal categories (it is a nice exercise to prove them). Importantly, what was a global operation before is now decomposed into smaller components that use the added generators.

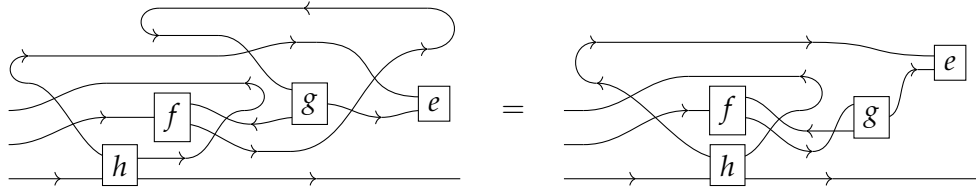
For traced monoidal categories, we could draw loops directly, to connect any left port to any right port of a diagram. We could always read information in a given diagram as flowing from left to right, except in a looping wire, where it flowed backwards at the top of the loop, until it reaches its destination. Now that we can move left ports to the right of a diagram and right ports to the left, we have to be a bit more careful. This is why we have to annotate each wire with a direction.

We can understand this as layering a notion of input and output on top of those of left and right ports. We can call inputs those wires that flow into a diagram and outputs those that flow out of a diagram, whether they are on the left or right boundary. Then, in a compact closed category,

we are allowed to *connect any input to any output*, i.e., we can assign a consistent direction to any wiring:



Furthermore, as is now a leitmotiv, only the connectivity matters: we are allowed to straighten or bend wires at will, as long as we preserve the connections between the different sub-diagrams. For example, the following two diagrams are equivalent:



4.2.3 Self-dual Compact Closed Categories Often, one encounters compact closed categories where the distinction between inputs and outputs disappears completely.

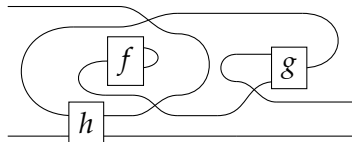
The term formation rules are the same as those symmetric monoidal categories, with the following additions:

$$\frac{}{\text{cup}_x^x} \quad x \in \Sigma_0 \qquad \frac{}{\text{cap}_x^x} \quad x \in \Sigma_0$$

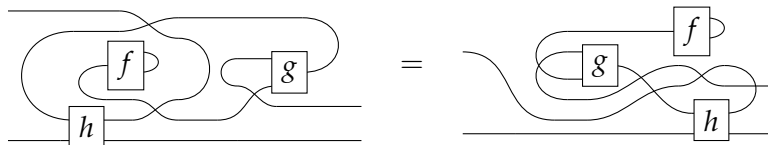
They are also very close to those of compact closed categories, but we identify x and its dual. As a result, there is no need to introduce a direction on wires. The added constants cup_x^x and cap_x^x satisfy the same equations as their directed cousins:

$$\text{cup}_x^x \stackrel{z}{=} \text{cap}_x^x \stackrel{s}{=} \text{cup}_x^x$$

Without distinct duals there is no need to keep track of the directionality of wires, and the resulting diagrammatic calculus is even more permissive—there are no inputs or outputs and we are now allowed to connect *any two ports together*:



As before the structural equivalence on diagrams allows us to identify any two diagrams where the same ports are connected:



4.2.5 Cartesian Categories Often, one would like to go further than having an operation that allows us to split and end wires— certain monoidal categories extend the capability of these operations to *copy and delete boxes* too. This is the ability that Cartesian categories¹¹ give us.

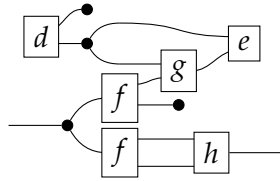
The term formation rules for Cartesian categories are the same as those of CD categories. The corresponding notion of structural equivalence further quotients that of CD categories with the following axioms, which capture the ability to copy and delete diagrams: for any $d : v \rightarrow w \in \Sigma_1$, we have

$$\begin{array}{c} v \text{---} [d] \text{---} \bullet \begin{array}{l} \text{---} w \\ \text{---} w \end{array} \stackrel{\text{dup}}{=} v \text{---} \bullet \begin{array}{l} [d] \text{---} w \\ [d] \text{---} w \end{array} \end{array} \quad \begin{array}{c} v \text{---} [d] \text{---} \bullet \stackrel{\text{del}}{=} v \text{---} \bullet \end{array} \quad (14)$$

Note that the axiom scheme above applies to generating operations with potentially multiple wires. To instantiate it, recall what the definition of $\text{---}\bullet$ and $\bullet\text{---}$ for multiple wires given in (13). Then, if we apply these to $g : x_1x_2 \rightarrow y_1y_2$, we get

$$\begin{array}{c} \text{---} [g] \text{---} \bullet \begin{array}{l} \text{---} \\ \text{---} \end{array} \end{array} = \begin{array}{c} \bullet \begin{array}{l} \text{---} [g] \text{---} \\ \text{---} [g] \text{---} \end{array} \end{array} \quad \begin{array}{c} \text{---} [g] \text{---} \bullet \end{array} = \begin{array}{c} \text{---} \bullet \end{array}$$

where we omit object labels for clarity. As for CD categories, we can now connect a given right port of a diagram to a (possibly empty) *set* of left ports of another:



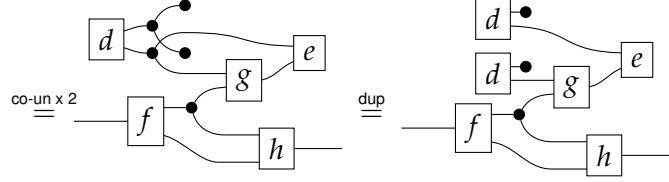
But the structural notion of equivalence for Cartesian categories is much coarser. For the first time in this paper, we encounter a diagrammatic language where the structural equivalence is not topological, and where equivalence cannot simply be checked by examining the connectivity of the different subdiagrams. In practice, this can make it more difficult to identify when two diagrams are equivalent. As well as those that have the connectivity between the different components, we can identify diagrams where one contains several copies of the same sub-diagram, connected by the same $\text{---}\bullet$, or where one contains a sub-diagram connected to a $\text{---}\bullet$ and the other does not, e.g.,

$$\begin{array}{c} \text{---} [d] \text{---} \bullet \begin{array}{l} \text{---} \\ \text{---} \end{array} \end{array} \begin{array}{c} \bullet \begin{array}{l} \text{---} [f] \text{---} \\ \text{---} [f] \text{---} \end{array} \end{array} \begin{array}{c} \text{---} [g] \text{---} \\ \text{---} [g] \text{---} \end{array} \begin{array}{c} \text{---} [e] \\ \text{---} [h] \end{array} = \begin{array}{c} \text{---} [d] \text{---} \bullet \begin{array}{l} \text{---} [d] \text{---} \\ \text{---} [d] \text{---} \end{array} \end{array} \begin{array}{c} \bullet \begin{array}{l} \text{---} [f] \text{---} \\ \text{---} [f] \text{---} \end{array} \end{array} \begin{array}{c} \text{---} [g] \text{---} \\ \text{---} [g] \text{---} \end{array} \begin{array}{c} \text{---} [e] \\ \text{---} [h] \end{array}$$

Above, from left to right, we have merged the two occurrences of f and copied d ; ($\text{---}\bullet \otimes \text{id}$). It is helpful to break down the required equational steps:

$$\begin{array}{c} \text{---} [d] \text{---} \bullet \begin{array}{l} \text{---} \\ \text{---} \end{array} \end{array} \begin{array}{c} \bullet \begin{array}{l} \text{---} [f] \text{---} \\ \text{---} [f] \text{---} \end{array} \end{array} \begin{array}{c} \text{---} [g] \text{---} \\ \text{---} [g] \text{---} \end{array} \begin{array}{c} \text{---} [e] \\ \text{---} [h] \end{array} \stackrel{\text{dup}}{=} \begin{array}{c} \text{---} [d] \text{---} \bullet \begin{array}{l} \text{---} [d] \text{---} \\ \text{---} [d] \text{---} \end{array} \end{array} \begin{array}{c} \bullet \begin{array}{l} \text{---} [f] \text{---} \\ \text{---} [f] \text{---} \end{array} \end{array} \begin{array}{c} \text{---} [g] \text{---} \\ \text{---} [g] \text{---} \end{array} \begin{array}{c} \text{---} [e] \\ \text{---} [h] \end{array} \stackrel{\text{co-un}}{=} \begin{array}{c} \text{---} [d] \text{---} \bullet \begin{array}{l} \text{---} [d] \text{---} \\ \text{---} [d] \text{---} \end{array} \end{array} \begin{array}{c} \bullet \begin{array}{l} \text{---} [f] \text{---} \\ \text{---} [f] \text{---} \end{array} \end{array} \begin{array}{c} \text{---} [g] \text{---} \\ \text{---} [g] \text{---} \end{array} \begin{array}{c} \text{---} [e] \\ \text{---} [h] \end{array}$$

¹¹The reader familiar with category will know that a Cartesian category is a category with finite products. Having products is a *property* of a category. A monoidal product, on the other hand, is a *structure* over a category. While categorical products do define a monoidal product, a given category may be equipped with a monoidal product that is not its categorical product.



In plain English, we first merge the two occurrences of f using the (dup) equation (from right to left); apply the counitality axiom of the comonoid structure to get rid of the extra counit and leave a plain wire; apply the counitality axiom of the comonoid twice (from right to left this time) to produce a diagram from which the (dup) axiom applies to d , which is the last equality.

Remark 4.3. The copying and deleting axioms imply that all diagram $d : v \rightarrow w$, with $w = x_1 \dots x_n$, can be decomposed uniquely into n diagrams $d_i : v \rightarrow x_i$, $1 \leq i \leq n$. Let us see concretely what this decomposition means and how to obtain the components for the case $w = x_1 x_2$:

$$[d_1] := [d] \text{ (left wire)} \quad [d_2] := [d] \text{ (right wire)}$$

We can check that

$$[d_2] \text{ (left)} := [d] \text{ (left)} \stackrel{\text{cpy}}{=} [d] \text{ (left)} \stackrel{\text{u}}{=} [d]$$

The general case is completely analogous.

In fact, the syntax of Cartesian categories is intimately connected to the standard symbolic syntax of algebraic theories. The connection is easier to glimpse in the single-sorted case, i.e. when the set of objects contains a single generator. Then, the decomposition property of diagrams implies that every diagram can be seen as a collection of operations with the same arity. We will come back to this point in Remark 5.19.

Remark 4.4. Starting from the structure of CD categories, it is possible to enforce different requirements. Cartesian categories impose the copy and delete axioms, but we could have dropped either of the two requirements to obtain a sensible concept. If we only impose the deleting axiom, we obtain a Markov category [55], a structure that has emerged in the literature as a synthetic setting for probability theory (see Section 7 for some pointers to recent work).

4.2.6 CoCartesian Categories If we flip all diagram of the previous section along the vertical axis, we obtain the dual of a Cartesian categories, namely *coCartesian categories*. They extend the language of symmetric monoidal categories, not with a commutative comonoid, but with a commutative monoid (cf. Example 2.15) instead:

$$\frac{}{x \circ x} \quad x \in \Sigma_0 \quad \frac{}{\circ x} \quad x \in \Sigma_0$$

Furthermore, we want these to *merge* (or co-copy) and *spawn* (or co-delete) any diagram as follows:

$$x \circ [d] \stackrel{\text{codup}}{=} [d] \circ x \quad \circ [d] \stackrel{\text{codel}}{=} \circ x \quad (15)$$

4.2.7 Biproduct Categories Categories that are both Cartesian and coCartesian combine both comonoid and monoids, where all diagrams satisfy (dup)-(del) and (codup)-(codel) respectively. Note one important consequence: if we apply (dup) to $d = \frown$, (dup) to $d = \circ$, (codup) to $d = \bullet$, and (del) to $d = \bullet$, we get the following:

These are the defining axioms of bimonoids! Cf. Example 2.18.

4.2.8 Hypergraph Categories We can extend further the capabilities of CD categories to allow connections between arbitrary sets of nodes. Like for biproduct categories, they include both a monoid and a comonoid but, as we will see, these interact differently.

The term formation rules are the same as for biproduct categories, *i.e.*, those symmetric monoidal categories, with the following additions:

The first two are similar to the extra generators of CD/Cartesian categories; the last two are their mirror image (we write them in black instead of the white generators of coCartesian and biproduct categories since they will play a different role, as we will now see).

The corresponding notion of structural equivalence is given by the laws of symmetric monoidal categories with the addition of the axioms of commutative, special Frobenius structures (Example 2.20) summarised in (10).

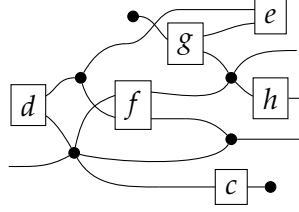
Notice however that we do not impose that every diagram can be (co)copied or (co)deleted as we did for (co)Cartesian categories. This is a key difference—in fact, as we will see later, these two requirements turn out to be incompatible in a rather fundamental way.

The diagrammatic language of hypergraph categories is the most permissive: it allows any set of ports (left or right) of the same sort to be connected together via $\frac{x}{x} \bullet \frac{x}{x}$, $\frac{x}{\bullet}$, $\frac{x}{x} \bullet \frac{x}{x}$, and $\bullet \frac{x}{x}$. In fact, as we have seen in Example 2.19, any two connected¹² diagrams made exclusively of these black generators, are equal if and only if they have the same number of left and right ports, a fact known as the spider theorem. For example, we can use the defining axioms of Frobenius algebra to show that the following two diagrams are equivalent:

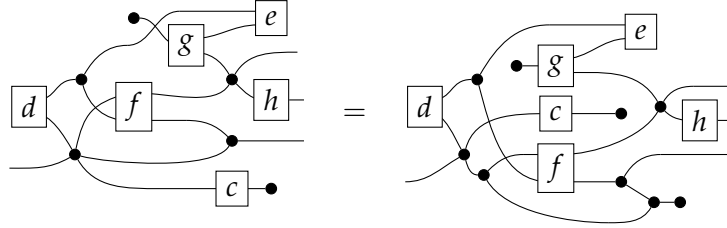
This means that the only relevant structure of a given connected diagram made entirely of $\frac{x}{x} \bullet \frac{x}{x}$, $\frac{x}{\bullet}$, $\frac{x}{x} \bullet \frac{x}{x}$, and $\bullet \frac{x}{x}$ is the number of ports on the left and on the right. As a result, as we have explained in Example 2.19, we can introduce the following hypernodes as syntactic sugar for any such diagram with n dangling wires on the left and m on the right:

¹²In the sense that there is a path from any two vertices.

Using this convenient notation, any diagram in a hypergraph category will look like a hypergraph-like structures, where any subset of ports can be connected wired together via hypernodes:



Once more, two diagrams are equivalent if they connect the same ports via hypernodes. For example, the following two are equivalent:



We could justify this equality through a sequence of axioms for Frobenius algebras and symmetric monoidal categories, but it would be very time-consuming! It suffices to check that the connectivity of the different labelled boxes and black vertices remains the same. This is why hypergraph categories are very appealing.

Remark 4.5. *It is the opportune time to clarify the relationship between hypergraph categories and the open hypergraph interpretation for string diagrams from Section 3. A motivation for the terminology—dating back at least to [85]—is that the category Hyp_Σ of Σ -labelled open hypergraphs form a hypergraph category. However, what yields the hypergraph category structure in Hyp_Σ is not the fact that objects are hypergraphs. For instance, the subcategory MHyp_Σ of monogamous hypergraphs is generally not a hypergraph category! The reason is rather to be found in the isomorphism between $\text{Free}_{\text{SMC}}(\Sigma + \text{scFrob})$ and Hyp_Σ —cf. Remark 3.6. More specifically, in the fact that the subcategory $\text{Free}_{\text{SMC}}(\text{scFrob})$ of $\text{Free}_{\text{SMC}}(\Sigma + \text{scFrob})$ and thus of Hyp_Σ is the free hypergraph category.*

Also note that hypergraph categories are the same as the well-supported compact closed categories studied in [27]. In fact, this work first observed that such structure always gives rise to a compact closed category in a canonical way.

Note that, in particular, hypergraph categories are self-dual compact closed. With the previous intuition, this is not too surprising—if we are able to connect any set of ports, we can connect any two pairs of ports. More formally, we can define cups and caps as $\bigcap_x^x := \bullet \bullet \curvearrowright$ and $\bigcup_x^x := \curvearrowleft \bullet \bullet$, for any x in the signature. That they satisfy the axioms of compact closed categories is a consequence of the Frobenius algebra axioms (or, more generally, of the Spider theorem). We give the diagrammatic proof explicitly here, as it is instructive:

$$\bigcap_x^x := \bullet \bullet \curvearrowright \stackrel{\text{Fr}}{=} \bullet \bullet \curvearrowright \bullet \bullet \stackrel{\text{coU}}{=} \bullet \bullet \curvearrowright \stackrel{\text{U}}{=} \text{---}$$

The other equation can be proved in the same way. That the resulting compact structure is also self-dual is immediate, since the cups and caps we have defined relate any given object to itself.

Remark 4.6 (A matter of perspective.). *The reader may have noticed that the additional structure of CD categories, self-dual compact closed, and hypergraph categories can also be seen as the free SMC over a signature that includes some additional generators and an equational theory (cf. Section 2.1). For example, the free CD category over the signature Σ is just the free SMC over $\Sigma' = (\Sigma_0 + \{-\bullet\curvearrowright, -\bullet\})$ quotiented by the axioms that make $-\bullet\curvearrowright, -\bullet$ a commutative comonoid. Whether we see the equations of that theory as structural features or an additional equational theory layered on top of a free SMC is a matter of perspective.*

4.2.9 Mix and Match Some of the above can be combined, but beware! Certain combinations are degenerate. The classic example is that of the incompatibility (in a sense that we will make precise) between Cartesian and compact closed categories. Indeed we claim the following: a category that is both Cartesian and compact closed is necessarily a poset. A poset is also a category, just one in which each homset contains at most one morphism. So if we can show from the axioms of Cartesian and compact closed categories that all morphisms between any two objects are equal, we are done. To prove this, we show that we can disconnect any wire, in two steps: first we show that the cup splits as follows,

$$\begin{array}{c} x \\ \text{cup} \\ x \end{array} \stackrel{\text{coU}}{=} \begin{array}{c} x \\ \bullet \\ \text{cup} \\ x \end{array} \stackrel{\text{coU}}{=} \begin{array}{c} x \\ \bullet \\ \bullet \\ \bullet \\ \text{cup} \\ x \end{array} \stackrel{\text{dup}}{=} \begin{array}{c} x \\ \bullet \\ \bullet \\ \text{cup} \\ x \end{array}$$

(the reader might recognise this as an instance of Remark 4.3). Then we show that the identity can be disconnected

$$x \stackrel{z}{=} \begin{array}{c} x \\ \text{cup} \\ x \end{array} = \begin{array}{c} x \\ \bullet \\ \bullet \\ \text{cup} \\ x \end{array} \stackrel{z}{=} \begin{array}{c} x \\ \bullet \\ \bullet \\ \text{cup} \\ x \end{array} =: \bullet \bullet$$

Finally, we can show what we wanted: for any $f : v \rightarrow w$,

$$v \boxed{f} w = v \boxed{f} \bullet \bullet w \stackrel{\text{del}}{=} v \bullet \bullet w$$

Not all combinations of diagrammatic languages are as badly behaved, however. If we weaken the Cartesian compact closed structure to that of a Cartesian traced monoidal category, the resulting combination is closely related to the notion of (parameterised) fixed-point [67]. A parameterised fixed-point operator in a Cartesian SMC $(\mathcal{C}, \times, 1)$ takes a morphism $f : X \times A \rightarrow X$ and produces $f^\dagger : A \rightarrow X$. The operator $(-)^+$ is then required to satisfy a certain number of intuitive axioms. For instance, f^\dagger should indeed be a fixed-point of f , i.e.,

$$\boxed{f^\dagger} = \bullet \begin{array}{c} \boxed{f^\dagger} \\ \bullet \\ \boxed{f} \end{array} \quad (16)$$

It is easy to see how to define such an operation in a traced category: let

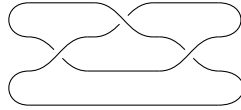
$$\boxed{f^\dagger} := \text{trace of } \boxed{f}$$

The axioms the fixed-point operator is required to satisfy are then consequences of the axioms of traced monoidal categories with those of Cartesian categories. For instance, it does satisfy (16):

$$\bullet \boxed{f^\dagger} \boxed{f} := \text{trace of } \boxed{f} \boxed{f} = \text{trace of } \boxed{f} \stackrel{\text{dup}}{=} \bullet \boxed{f} \quad (17)$$

where the second equality holds by the yanking and sliding axioms of the trace. Conversely, from a given fixed-point operator, we can define a trace. In fact, the two notions—parameterised fixed-points and Cartesian traces—are equivalent [67, Theorem 3.1].

Another classic example of a useful interaction between different structures: diagrams for braided and self-dual compact closed categories, allow us to draw arbitrary *knots*:



The central result for these categories is that two different, closed (*i.e.* with empty left and right boundary) diagrams are equal if and only if the corresponding knots can be topologically deformed into one another—thus, the topological notion of knot has been fully captured by a few algebraic axioms. We see here the advantage of working with monoidal categories: they give an algebraic home to topological concepts that are difficult to express in standard algebraic syntax.

5 Semantics

In this section we explain how to assign meaning to diagrams. So far, we have explored an arsenal of diagrammatic syntax, each corresponding to a specific flavour of monoidal category. However, Their semantics—what these different diagrams with their corresponding form of composition *mean*—was left open.

This is similar to how we can assign a denotational semantics to programming languages, in order to define what each program is supposed to mean. Computers do not just compute, but we program them to compute something; semantics specifies what that ‘something’ is intended to be. The semantics allows to define the behaviour of programs in a given language rigorously and prove properties that they satisfy more easily. The same language can even have different interpretations. A well chosen semantics may allow us to circumscribe more precisely the expressiveness of a language or to rule out entire classes of behaviour.

The same idea also appears in algebra where we might define axiomatically some algebraic structure—groups, rings, modules, vector spaces etc.—via an algebraic theory (given by some signature and equations, like we have done for the different forms of diagrammatic languages in the previous section) and study concrete *models* of this theory. Thus, the reader more familiar with abstract algebra might prefer to hear the word “model” every time they come across the word “semantics”. Models, in turn, can teach us valuable things about the theory. Like for programming languages, working with a model may be easier than working with the syntax directly; some computations are often easier to carry out in a well chosen model.

5.1 From Syntax to Semantics, Functorially

Before introducing semantics, it will be beneficial to recall what we mean by syntax. We have seen that the mathematical way to specify a syntax is by defining *free* structures of a certain kind. Following the same patters, all of the diagrammatic languages of the previous sections were a recipe to construct a free monoidal category of some kind over a signature, *i.e.*, a set of generators (and some equations). As we saw in the previous section, the axioms vary according to the kind of diagrams that we want to draw.

The first diagrammatic language we encountered in Section 2 was that of *symmetric monoidal categories* (SMCs). We gave the definition of the free SMC over a chosen signature in Definition 2.6.

Similarly, all the diagrammatic languages covered above (in Section 4) come with an associated notion of free structure: they define the free monoidal category, the free braided category, the free traced monoidal category, the free compact closed category etc. over a given signature. In each case, the free X -category $\text{Free}_X(\Sigma)$ (with $X = \text{monoidal, braided monoidal, symmetric monoidal, etc.}$) over a given signature $\Sigma = (\Sigma_0, \Sigma_1)$ is the category that has objects all lists of elements of Σ_0 and as morphisms all corresponding string diagrams built from Σ using the derivation rules of X , modulo the structural equations of X . Moreover, given a signature *and a theory* we can form the SMC $\text{Free}_X(\Sigma, E)$ obtained by quotienting $\text{Free}_X(\Sigma)$ by the equivalence relation over diagrams given by $\stackrel{E}{\sim}$.

Then, to give semantics to diagrams amounts to assigning a mathematical definition to each basic building block, making sure that they satisfy the relevant axioms. Giving such a mapping involves specifying:

- an object $\llbracket x \rrbracket$ for each generating object $x \in \Sigma_0$;
- a morphism $\llbracket c \rrbracket$ for each generating operation $c \in \Sigma_1$;
- where to map the elements that make up the additional structure X , such that the relevant axioms are satisfied.

For this, we need to choose some X -category (Sem, \otimes, I) and two mapping:

$$\llbracket - \rrbracket_0 : \Sigma_0 \rightarrow \text{Objects of } (\text{Sem}, \otimes, I) \quad \llbracket - \rrbracket_1 : \Sigma_1 \rightarrow \text{Morphisms of } (\text{Sem}, \otimes, I)$$

such that the defining elements of the X -structure are mapped to those of C .

For the last point, each particular case has different requirements. We have already seen what a symmetric monoidal functor (Definition 2.9) should be: $\llbracket - \rrbracket$ maps the symmetry \bowtie_x^y in the syntax to some morphism $\llbracket x \rrbracket \otimes \llbracket y \rrbracket \rightarrow \llbracket y \rrbracket \otimes \llbracket x \rrbracket$ which satisfies all the relevant axioms (cf. (5)) in the semantics. Similarly, if we are working with hypergraph categories, we may prefer to consider functors that map the designated Frobenius structure generators $\overset{x}{\bullet} \overset{x}{\circlearrowleft}, \overset{x}{\bullet} \overset{x}{\circlearrowright}, \overset{x}{\circlearrowleft} \bullet, \overset{x}{\circlearrowright} \bullet$ to designated morphisms that satisfy the axioms of Frobenius structures (Example 2.19), for each generating object x in the syntax.

Because $\text{Free}_X(\Sigma)$ is free, the two mappings above define a X -monoidal functor into Sem : a mapping

$$\llbracket - \rrbracket : \text{Free}_X(\Sigma) \rightarrow (\text{Sem}, \otimes, I)$$

which satisfies $\llbracket w_1 w_2 \rrbracket = \llbracket w_1 \rrbracket \otimes \llbracket w_2 \rrbracket$, $\llbracket \epsilon \rrbracket = I$, and

$$\llbracket \begin{array}{c} u \\ \boxed{c} \\ v \end{array} \begin{array}{c} \boxed{d} \\ w \end{array} \rrbracket = \llbracket \begin{array}{c} u \\ \boxed{c} \\ v \end{array} \rrbracket ; \llbracket \begin{array}{c} v \\ \boxed{d} \\ w \end{array} \rrbracket \quad \llbracket \begin{array}{c} v_1 \\ \boxed{d_1} \\ v_2 \end{array} \begin{array}{c} w_1 \\ \boxed{d_2} \\ w_2 \end{array} \rrbracket = \llbracket \begin{array}{c} v_1 \\ \boxed{d_1} \\ v_2 \end{array} \rrbracket \otimes \llbracket \begin{array}{c} v_2 \\ \boxed{d_2} \\ w_2 \end{array} \rrbracket \quad (18)$$

Let us emphasise this last point: since the syntax is free, it is enough to define $\llbracket - \rrbracket$ only on the generating objects and morphisms of the signature. The semantics of an arbitrary (composite) diagram can then be computed using the composition and monoidal product in the semantics, as long as the latter has the appropriate structure. This is what we mean by *compositionality* in this context.

Remark 5.1. Note that the requirement that the image of the added X -structure satisfy the right properties (e.g., that $\llbracket \bowtie \rrbracket$ behave as a genuine symmetry in the semantics, or similarly for $\llbracket \bullet \circlearrowleft \rrbracket, \llbracket \bullet \circlearrowright \rrbracket$ etc.) is often verified by default as a consequence of the two equations in (18) (preservation of composition and monoidal

product). Take the example of the symmetry: we need $\llbracket \bowtie_x^y \rrbracket ; \llbracket \times_y^x \rrbracket = id_{\llbracket x \otimes y \rrbracket}$. But $\llbracket \bowtie_x^y \rrbracket ; \llbracket \times_y^x \rrbracket = \llbracket \times_y^x \rrbracket = \llbracket \text{---}_y^x \rrbracket = id_{\llbracket x \otimes y \rrbracket}$ because $\bowtie_y^x = \text{---}_y^x$ in the syntax.

We now give several useful examples of semantics. Each time, we use the same notation $\llbracket - \rrbracket$ for the semantic functor.

Example 5.2 (Functions, \times). The category **Set** of sets and functions with function composition can be equipped with a monoidal structure in different ways. The Cartesian product of sets is an example of a monoidal product: on objects it is simply the set of pairs, given by $X_1 \times X_2 = \{(x_1, x_2) \mid x_1 \in X_1 \wedge x_2 \in X_2\}$ and on morphisms it is given by $(f_1 \times f_2)(x_1, x_2) = (f_1(x_1), f_2(x_2))$. The unit for the product is the singleton set $1 = \{\bullet\}$ (any singleton set will do). It is straightforward to check that these satisfy the axioms of monoidal categories in (5). Let us prove the interchange law carefully to see how this works:

$$\begin{aligned} ((f_1 \times f_2) ; (g_1 \times g_2))(x_1, x_2) &= (g_1 \times g_2)(f_1(x_1), f_2(x_2)) \\ &= (g_1(f_1(x_1)), g_2(f_2(x_2))) \\ &= ((f_1 ; g_1)(x_1), (f_2 ; g_2)(x_2)) = ((f_1 ; g_1) \times (f_2 ; g_2))(x_1, x_2) \end{aligned}$$

(Side note: strictly speaking, using pairs for ' \times ' does not define an associative monoidal product, because $(X_1 \times X_2) \times X_3$ is not *equal* to $X_1 \times (X_2 \times X_3)$, but merely isomorphic to it. See Remark 2.5 below.)

All of this means that, given the free monoidal category $\text{Free}_{MC}(\Sigma)$ over some signature Σ , specifying a monoidal functor $\llbracket - \rrbracket : \text{Free}_{MC}(\Sigma) \rightarrow \text{Set}$ involves assigning a set to each element of Σ_0 and a function $\llbracket c \rrbracket : \llbracket v \rrbracket \rightarrow \llbracket w \rrbracket$ for each $c : v \rightarrow w$ in Σ_1 , as explained above. Then the semantics of an arbitrary diagram can be obtained by using the definition of composition (notice the inversion of the usual notation for composition) and the Cartesian product:

$$\llbracket \text{---}_c^u \text{---}_d^v \text{---}_w \rrbracket = \llbracket \text{---}_d^v \text{---}_w \rrbracket \circ \llbracket \text{---}_c^u \text{---}_v \rrbracket \quad \llbracket \begin{array}{c} \text{---}_{d_1}^{v_1} \\ \text{---}_{d_2}^{v_2} \end{array} \text{---}_{w_2}^{w_1} \rrbracket = \llbracket \text{---}_{d_1}^{v_1} \text{---}_{w_1} \rrbracket \times \llbracket \text{---}_{d_2}^{v_2} \text{---}_{w_2} \rrbracket$$

Furthermore, **Set**, \times is also a *symmetric* monoidal category, with symmetry given by the function $\sigma_X^Y : X \times Y \rightarrow Y \times X$ defined by $\sigma_X^Y(x, y) = (y, x)$. If we want $\llbracket - \rrbracket$ to also be a symmetric monoidal functor, then we can just set $\llbracket \times_y^x \rrbracket = \sigma_Y^X$, for $\llbracket x \rrbracket = X$, $\llbracket y \rrbracket = Y$.

This symmetric monoidal category is moreover a *Cartesian* category (Section 4.2.5). The comonoid structure is given by the following copy $\Delta : X \rightarrow X \times X$ and discarding $! : X \rightarrow 1$ maps:

$$\llbracket \begin{array}{c} X \\ \bullet \\ X \end{array} \rrbracket (x) = \Delta(x) = (x, x) \quad \llbracket \begin{array}{c} X \\ \bullet \end{array} \rrbracket (x) = !(x) = \bullet$$

Note that there is only one diagram $X \rightarrow 1$ for any set X , namely the discarding map $!_X$ (there is only one way to discard some element in this category). One can readily check that these satisfy the axioms of commutative comonoids (2.16) and that any function satisfies the equations (dup) and (del) from (14). To build a bit more intuition, let us verify (dup) for example. For any $x \in X$, we have

$$\begin{aligned} \llbracket \begin{array}{c} X \\ \text{---} f \text{---} \bullet_Y^Y \end{array} \rrbracket (x) &= (\llbracket \text{---} \bullet \text{---} \rrbracket \circ \llbracket f \rrbracket)(x) = \llbracket \text{---} \bullet \text{---} \rrbracket (\llbracket f \rrbracket(x)) \\ &= (\llbracket f \rrbracket(x), \llbracket f \rrbracket(x)) = (\llbracket f \rrbracket \times \llbracket f \rrbracket)(x) = \llbracket \begin{array}{c} X \\ \bullet \\ \begin{array}{c} \text{---} f \text{---}^Y \\ \text{---} f \text{---}_Y \end{array} \end{array} \rrbracket (x) \end{aligned}$$

Example 5.3 (Relations, \times). The category Rel has as objects, sets, and as morphisms $R : X \rightarrow Y$, binary relations, *i.e.* subsets $R \subseteq X \times Y$. The composition of two relations $R : X \rightarrow Y$ and $S : Y \rightarrow Z$ is defined by $R ; S = \{(x, z) \mid \exists y (x, y) \in R \wedge (y, z) \in S\}$. The Cartesian product $X \times Y$ further defines a monoidal product on Rel , with unit the singleton set $1 = \{\bullet\}$. Furthermore, Rel is *symmetric* monoidal, with the symmetry $X \times Y \rightarrow Y \times X$ given by the relation $\{((x, y), (y, x)) \mid x \in X, y \in Y\}$. It is easy to see that this relation is self-inverse and satisfies the other axioms required of the symmetry. Even though this monoidal product is the same as in Set on objects, the properties of the two SMCs are very different.

Given the free monoidal category $\text{Free}_{\text{MC}}(\Sigma)$ over some signature Σ , specifying a monoidal functor $\llbracket - \rrbracket : \text{Free}_{\text{MC}}(\Sigma) \rightarrow \text{Rel}$ means assigning a set to each element of Σ_0 and a relation $\llbracket c \rrbracket \subseteq \llbracket v \rrbracket \times \llbracket w \rrbracket$ for each $c : v \rightarrow w$ in Σ_1 . Then monoidal functoriality—aka compositionality—means that:

$$\begin{aligned} \llbracket \begin{array}{c} \text{---}^u \text{---}^v \text{---}^w \\ \boxed{c} \text{---}^w \end{array} \rrbracket &= \{(x, z) \mid \exists y (x, y) \in \llbracket \text{---}^u \text{---}^v \rrbracket \wedge (y, z) \in \llbracket \text{---}^v \text{---}^w \rrbracket\} \\ \llbracket \begin{array}{c} \text{---}^{v_1} \text{---}^{w_1} \\ \boxed{d_1} \text{---}^{w_1} \\ \text{---}^{v_2} \text{---}^{w_2} \\ \boxed{d_2} \text{---}^{w_2} \end{array} \rrbracket &= \{((x_1, x_2), (y_1, y_2)) \mid (x_1, x_2) \in \llbracket \text{---}^{v_1} \text{---}^{w_1} \rrbracket \wedge (x_2, y_2) \in \llbracket \text{---}^{v_2} \text{---}^{w_2} \rrbracket\} \end{aligned}$$

One can moreover make Rel into a *self-dual compact closed* category (Section 4.2.3) by choosing cups and caps, coherently for every set X . One common possibility is to define the cup as $\{(\bullet, (x, x)) \mid x \in X\}$ and the cap as $\{((x, x), \bullet) \mid x \in X\}$. It is clear that these two relations satisfying the defining axioms of compact closed categories.

In fact, we can go even further: Rel can be made into a *hypergraph* category (Section 4.2.8). For this we need to choose a special, commutative Frobenius structure on every object X . One possibility is the one whose comultiplication is the diagonal relation and counit is the projection, with the multiplication and unit given by the converse relations:

$$\begin{aligned} \llbracket \begin{array}{c} \text{---}^X \text{---}^X \\ \bullet \text{---}^X \end{array} \rrbracket &= \{(x, (x, x)) \mid x \in X\} & \llbracket \begin{array}{c} \text{---}^X \\ \bullet \end{array} \rrbracket &= \{(x, \bullet) \mid x \in X\} \\ \llbracket \begin{array}{c} \text{---}^X \\ \bullet \text{---}^X \end{array} \rrbracket &= \{((x, x), x) \mid x \in X\} & \llbracket \begin{array}{c} \bullet \text{---}^X \end{array} \rrbracket &= \{(\bullet, x) \mid x \in X\} \end{aligned} \tag{19}$$

Note that the cups and caps from the (self-dual) compact closed structure of Rel comes from this Frobenius structure, as explained in Section 4.2.8. Let us check (one side of) the Frobenius law, to see how this works in more detail:

$$\begin{aligned} ((x_1, x_2), (x'_1, x'_2)) &\in \llbracket \begin{array}{c} \text{---} \text{---} \text{---} \\ \bullet \text{---} \text{---} \bullet \end{array} \rrbracket \\ \Leftrightarrow ((x_1, x_2), (x'_1, x'_2)) &\in \left(\llbracket \text{---} \rrbracket \times \llbracket \begin{array}{c} \text{---} \bullet \text{---} \end{array} \rrbracket \right) ; \left(\llbracket \begin{array}{c} \text{---} \bullet \text{---} \end{array} \rrbracket \times \llbracket \text{---} \rrbracket \right) \\ \Leftrightarrow \exists x''_1, x''_2, x''_3 ((x_1, x_2), (x'_1, x'_2, x''_3)) &\in \left(\llbracket \text{---} \rrbracket \times \llbracket \begin{array}{c} \text{---} \bullet \text{---} \end{array} \rrbracket \right) \wedge ((x'_1, x'_2, x''_3), (x'_1, x'_2)) \in \left(\llbracket \begin{array}{c} \text{---} \bullet \text{---} \end{array} \rrbracket \times \llbracket \text{---} \rrbracket \right) \\ \Leftrightarrow \exists x''_1 \exists x''_2 \exists x''_3 [(x_1 = x''_1) \wedge (x_2 = x''_2 = x''_3)] &\wedge ((x'_1 = x''_1 = x'_1) \wedge (x'_2 = x''_2)) \\ \Leftrightarrow x_1 = x_2 = x'_1 = x'_2 & \\ \Leftrightarrow \exists x [(x_1 = x_2 = x) \wedge (x = x'_1 = x'_2)] & \\ \Leftrightarrow ((x_1, x_2), (x'_1, x'_2)) &\in \llbracket \begin{array}{c} \text{---} \bullet \text{---} \bullet \end{array} \rrbracket \end{aligned}$$

In practice, one rarely reasons this way about string diagrams for relations. There is a much more intuitive way: if we think of each wire of the diagram as being labelled by a variable, then black

nodes force all variables labelling its left and right legs to be equal. With this in mind, it is plain to see that any connected network of black nodes forces all of the corresponding variables to be the same. This can be seen as a semantic instance of the spider theorem (covered in Example 2.19)! The special case we have shown above falls out as a corollary.

Finally, functions can also be seen as relations via their graph $\text{Graph}(f) = \{(x, y) \mid y = f(x)\}$. Moreover, the composite (as relations) of two functional relations is the graph of the composite of the two corresponding functions: $\text{Graph}(g \circ f) = \text{Graph}(f) ; \text{Graph}(g)$. In other words, Graph defines a functor from $\text{Set} \rightarrow \text{Rel}$. In Rel , it is possible to identify functional relations—relations that are the graph of some function—purely by how they interact with the hypergraph structure: they are precisely those that satisfy the (dup) and (del) axioms with respect to \multimap and \multimap , as in the example of Set above. Indeed, a relation f satisfies (dup) iff it is single-valued, and it satisfies (del) iff it is total (a nice exercise). This is a useful characterisation that often comes up in the literature.

Remark 5.4 (Not strict?). *The observant reader may have noticed an issue with the previous examples: the SMCs of functions and relations are not strict. This is because taking the Cartesian product is not strictly associative, i.e. the set $(X \times Y) \times Z$ is not equal to the set $X \times (Y \times Z)$. However, it is harmless to pretend that they are (and again, this is why we can draw string diagrams in this category). If the reader is still uncomfortable with this idea, we invite them to give an equivalent presentation of the same SMC that does not rely on taking pairs, but tuples of arbitrary length. This SMC would then be the strictification of Set or Rel and nothing of importance would be lost.*

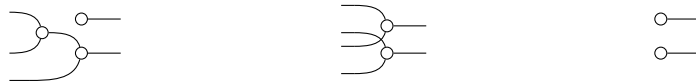
Example 5.5 (Functions, +). The Cartesian product is only one among several possible choices of monoidal structures that one SMC in at least one other interesting way. Instead of taking the monoidal product to be the Cartesian product of sets, we consider the disjoint sum: $X_1 + X_2 := ((X_1 \times \{1\}) \cup (X_2 \times \{2\}))$ on objects, and $f_1 + f_2$ on morphisms, given by $(f_1 + f_2)(x, i) = f_i(x)$. We also take the unit to be the empty set. Unsurprisingly, the category is *not* Cartesian by it is *coCartesian* (Section 4.2.6)! The canonical monoid on each object is given by the following maps:

$$\left[\begin{array}{c} X \\ \multimap \\ X \end{array} \right] (x, i) = x \text{ (for } i = 1, 2) \quad \left[\begin{array}{c} X \\ \multimap \\ \emptyset \end{array} \right] = \emptyset$$

It is straightforward to check that every function satisfies the (codup) and (codel) axioms with respect to these.

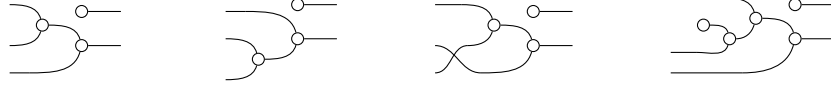
In fact, every function between finite sets can be represented using this syntax. We only need $\begin{array}{c} 1 \\ \multimap \\ 1 \end{array}$ and $\begin{array}{c} \multimap \\ 1 \end{array}$ over the set singleton set 1, which we will simply write as \multimap and \multimap . Given a function $f : X \rightarrow Y$, we first fix some ordering of X and Y so that we can identify them with finite ordinals, which encode sequences of wires in the diagrammatic setting (we will assume the same thing for many examples below). Then we can use as many \multimap as necessary to connect all elements x in the domain to the single y in the codomain to which f sends them; those elements of Y that are not in the image of f (the map might not be surjective) are simply connected to a \multimap . Here are a few examples of the translation:

$$f : \{0, 1, 2\} \rightarrow \{0, 1\} \quad g : \{0, 1, 2, 3\} \rightarrow \{0, 1\} \quad ! : \emptyset \rightarrow \{0, 1\}$$



The first is given by $f(0) = f(1) = f(2) = 1$, the second by $g(0) = g(2) = 0$ and $g(1) = g(3) = 1$, and the last is the unique map from the empty set to $\{0, 1\}$.

Notice that there are several ways of drawing the same function, depending on how we choose to arrange the different \rhd and \circ . The following diagrams all represent f , as defined above:



In a coCartesian category, all these diagrams are equal, as \rhd and \circ form a commutative monoid. This is the first instance of an monoidal theory we encounter that fully characterises the chosen semantics. In other words, *the free coCartesian category over a single object* (and no morphisms) is equivalent to the SMC of finite sets and functions, with the disjoint sum. In this case, we say that the theory of a commutative monoid is *complete* for this semantics. We will come back to completeness of theories in Section 5.2.

Example 5.6 (Bijections, $+$). If we restrict the previous example to bijections (one-to-one and onto functions) only, we obtain the simplest example of a SMC—call it *Bij*. Indeed, string diagrams for *Bij* are simply permutations of the wires! If we restrict further to finite ordinals, the resulting SMC is *the free SMC over the signature* $(\{\bullet\}, \emptyset)$ —the SMC of permutations we have already encountered in Example 2.8.

Example 5.7 (Relations, $+$). As for functions, the disjoint sum gives another interesting monoidal product on relations. On objects, it remains the same: $X_1 + X_2 := ((X_1 \times \{1\}) \cup (X_2 \times \{2\}))$. On morphisms, $R_1 + R_2$ is given by $((x, i), (y, i)) \in R_1 + R_2$ iff $(x, y) \in R_i$ for some $i \in \{1, 2\}$. Once again, the unit is the empty set. In this case, the category is *not* compact closed and *a fortiori* not a hypergraph category.

With the disjoint sum as monoidal product, $(\text{Rel}, +)$ is a coCartesian category, with monoid given by the graph of the relations that give its coCartesian structure to $(\text{Set}, +)$:

$$\left[\begin{array}{c} X \\ \rhd \\ X \end{array} \right] = \{((x, i), x) \mid x \in X, i = 1, 2\} \quad \left[\begin{array}{c} X \\ \circ \\ \emptyset \end{array} \right] = \emptyset$$

In fact, $(\text{Rel}, +)$ is also a *biproduct* category (Section 4.2.7), with copying and deleting relations given by the converse of the above relations:

$$\left[\begin{array}{c} X \\ \bullet \\ X \end{array} \right] = \{(x, (x, i)) \mid x \in X, i = 1, 2\} \quad \left[\begin{array}{c} X \\ \bullet \\ \emptyset \end{array} \right] = \emptyset$$

One can check that \bullet and \bullet form a commutative comonoid for any X , and that they can copy and delete any relation, *i.e.*, that they satisfy the (dup) and (del) axioms from (14).

Intuitively, we can think of the diagrams \rhd , \circ , \bullet , \bullet in this category as directing the flow of a single token that travels around the wires. The intuition here is that the \rhd transfers to the right wire the token that comes through any one of its left wires, \circ is able to generate a token, \bullet is a non-deterministic forl and \bullet a dead-end. As we have already said, the relations for \rhd and \circ are simply the graphs of the functions defined in the category of sets and functions. This makes sense: functions can only direct the token deterministically from left to right. Hence, they lack the nondeterministic \bullet and \bullet .

Note that the particle intuition for diagrams in *Rel* with the disjoint sum as monoidal product is quite different from the intuition for diagrams in *Rel* with the Cartesian product, where values for all wires are set to compatible values globally, all at once. For this reason, diagrams in biproduct categories are sometimes called *particle-style* while those of self-dual compact closed categories are said to be *wave-style* in some contexts [1].

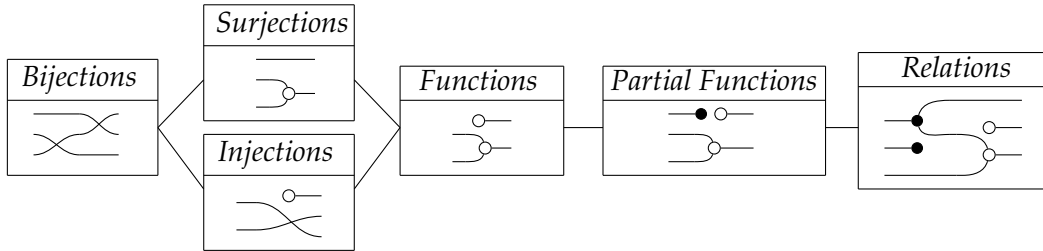
As in the previous example, we can represent *any* relation between finite sets, using these basic diagrams. For this, we only need $\overset{1}{\bullet} \overset{1}{\curvearrowright}$, $\overset{1}{\bullet} \overset{1}{\circ}$, $\overset{1}{\circ} \overset{1}{\curvearrowright}$ and $\overset{1}{\circ} \overset{1}{\circ}$ over the set singleton set 1, which we will once gain write as $\bullet \curvearrowright$, $\bullet \circ$, $\circ \curvearrowright$ and $\circ \circ$. A relation $R : X \rightarrow Y$ corresponds to a diagram d with $|X|$ wires on the left and $|Y|$ wires on the right. The j -th port on the left is connected to the i -th port on the right exactly when $(i, j) \in R$. For example, the relation $R : \{0, 1, 2\} \rightarrow \{0, 1\}$ given by $\{(0, 0), (0, 1), (2, 1)\}$, can be represented by the following diagram:



where a single wire is interpreted as the identity over some choice of singleton set. We see that this representation extends that of functions, by adding the possibility of connecting one wire on the left to several (or none) on the right. This is precisely the difference between functions and relations, reflected in the diagrams.

Note that a relation can also be seen as a matrix with Boolean coefficient. The relationship between the diagrams above and matrices (over more general semirings) will be explained more generally below in Example 5.14.

Remark 5.8. We have just seen that going straight from functions to relations (with the disjoint sum as product) amounted to adding $\bullet \curvearrowright$ and $\bullet \circ$ to $\circ \curvearrowright$ and $\circ \circ$. These two examples fit into a whole hierarchy of expressiveness:



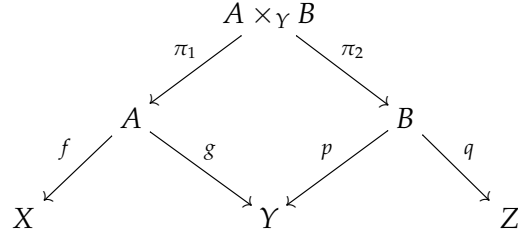
Of course, any subset of the generators $\bullet \curvearrowright$, $\bullet \circ$, $\circ \curvearrowright$, $\circ \circ$ gives a well-defined sub-SMC of $(\text{Rel}, +)$. We have not included all 2^4 of them as they do not all correspond to well-known mathematical notions.

We should also note that the way in which these theories are combined define distributive laws, a topic we have mentioned briefly in Remark 2.21.

Example 5.9 (Spans, \times). The category **Span**(Set) has sets as objects and, as morphisms $X \rightarrow Y$, pairs of maps $f : A \rightarrow X, g : A \rightarrow Y$ with the same set A as domain. We will write spans as $X \xleftarrow{f} A \xrightarrow{g} Y$. One way to think about spans is as *witnessed* or *proof relevant* relations that keep track of the way in which two elements are related: an element a of the apex A can be thought of as a witness or a proof of the fact that $(f(a), g(a))$ are related by the span. Thus, the difference with relations is that there may be several ways in which two elements from X and Y are related by the same span (A, f, g) ; if $f(a) = f(a') = x$ and $g(a) = g(a') = y$, then (x, y) are related by two different witnesses a and a' . If we forget the additional structure at the apex, we obtain a relation (in fact there is a way of constructing Rel from **Span**(Set); the interested reader will find this explained in [59, 110]).

The composition of two spans is obtained by computing what is called *the pullback* of g and p

below and composing the resulting outer two functions on each side:



where $A \times_Y B := \{(a, b) \mid (g(a) = p(b))\}$ and π_1, π_2 are the two projections onto A and B . Thus, the composition of $X \xleftarrow{f} A \xrightarrow{g} Y$ followed by $Y \xleftarrow{p} B \xrightarrow{q} Z$ is $X \xleftarrow{f \circ \pi_1} A \times_Y B \xrightarrow{q \circ \pi_2} Z$. For a set X , the identity span is $X \xleftarrow{id_X} X \xrightarrow{id_X} X$. As given, this operation is not strictly associative or unital. To make **Span**(Set) into a bona fide category, we need to identify all isomorphic spans: two spans $X \xleftarrow{f} A \xrightarrow{g} Y$ and $Y \xleftarrow{p} B \xrightarrow{q} Z$ are isomorphic when there is a bijection $h : A \rightarrow B$ such that $p \circ h = f$ and $q \circ h = g$.

Span(Set) can be made into a symmetric monoidal category with the Cartesian product of sets: on objects $X_1 \times X_2$ is the usual set of pairs of elements of X_1 and X_2 , on morphisms $(X_1 \xleftarrow{f_1} A_1 \xrightarrow{g_1} Y_1) \otimes (X_2 \xleftarrow{f_2} A_2 \xrightarrow{g_2} Y_2) = (X_1 \times X_2 \xleftarrow{f_1 \times f_2} A_1 \times A_2 \xrightarrow{g_1 \times g_2} Y_1 \times Y_2)$. With the singleton set as unit and the symmetry as $X \times Y \xleftarrow{id} X \times Y \xrightarrow{\sigma_X^Y} Y \times X$ where $\sigma_X^Y(x, y) = (y, x)$ as before, this equips **Span**(Set) with a symmetric monoidal structure.

Furthermore, like relations, spans form a hypergraph category, with the choice of Frobenius structure for each set X given by

$$\begin{aligned} \left[\begin{array}{c} X \\ \bullet \\ \frac{X}{X} \end{array} \right] &= X \xleftarrow{id_X} X \xrightarrow{\Delta_X} X \times X & \left[\begin{array}{c} X \\ \bullet \end{array} \right] &= X \xleftarrow{id_X} X \xrightarrow{!_X} 1 \\ \left[\begin{array}{c} \frac{X}{X} \\ \bullet \\ X \end{array} \right] &= X \times X \xleftarrow{\Delta} X \xrightarrow{id_X} X & \left[\begin{array}{c} X \\ \bullet \\ X \end{array} \right] &= 1 \xleftarrow{!_X} X \xrightarrow{id_X} X \end{aligned} \quad (21)$$

where Δ is the usual diagonal, defined by $\Delta_X(x) = (x, x)$, and $!_X$ the unique map $X \rightarrow 1$. The proof that these satisfy the appropriate axioms can be computer much like for relations, with the added complexity that one has to keep track of the witnesses in each apex.

Notice that if we forget the apex of each of these, and only keep track of the pairs that they relate, the resulting relations are exactly those that give Rel its hypergraph structure, cf. (19). This is a general fact about spans and relations. If spans (of sets or any category with categorical products) of type $X \rightarrow Y$ can be seen as maps $A \rightarrow X \times Y$, relations are precisely *injective* (or monomorphic, in the general categorical setting) spans. One can always obtain a relation from a span $A \rightarrow X \times Y$ by first factorising the map into a surjective map (epimorphism) followed by an injective (monomorphism) one, and keeping only the latter.

Example 5.10 (Spans,+). As for relations, spans can also be equipped with the disjoint sum as monoidal product. In fact, with this monoidal product, we can use the same diagrammatic language to represent any span of finite sets, $\dashv\!\!\!\!\!\bullet$, $\dashv\!\!\!\!\!\bullet$, $\dashv\!\!\!\!\!\bullet$, $\dashv\!\!\!\!\!\bullet$ with the following slightly modified interpretation:

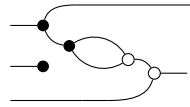
$$\left[\begin{array}{c} X \\ \bullet \\ \frac{X}{X} \end{array} \right] = X \xleftarrow{\nabla_X} X + X \xrightarrow{id_X} X + X \quad \left[\begin{array}{c} X \\ \bullet \end{array} \right] = X \xleftarrow{0} \emptyset \xrightarrow{id_X} \emptyset$$

$$\left[\begin{array}{c} X \\ \text{X} \end{array} \right] = X + X \xleftarrow{id_X} X + X \xrightarrow{\nabla_X} X \quad \left[\begin{array}{c} X \\ \text{X} \end{array} \right] = \emptyset \xleftarrow{id_X} \emptyset \xrightarrow{0} X$$

where $\nabla_X : X + X \rightarrow X$ is defined by $\nabla_X(x, i) = x$ and $0_X : \emptyset \rightarrow X$ is the only map from the empty set to any set X .

With these, $(\mathbf{Span}(\mathbf{fSet}), +)$ is also a biproduct category as defined in Section 4.2.7. In fact, it is the free biproduct category over a single object and no additional morphism [80, 5.3].

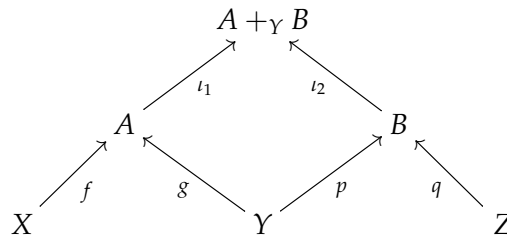
As for relations, we can use $\overset{1}{\bullet} \overset{1}{\circ} \overset{1}{\circ}$, $\overset{1}{\bullet} \overset{1}{\bullet}$, $\overset{1}{\circ} \overset{1}{\bullet}$ and $\overset{1}{\circ} \overset{1}{\circ}$ to represent any span of finite sets with $+$ as monoidal product: for the span $n \xleftarrow{f} A \xrightarrow{g} m$, there is a path from the i -th wire on the left to the j -th one on the right in the corresponding diagram for each element of $\{a \in A \mid f(a) = j, g(a) = i\}$. For example, the span $3 \xleftarrow{f} 4 \xrightarrow{g} 2$, with $f(0) = f(1) = f(2) = 0, f(3) = 2$ and $g(0) = 0, g(1) = g(2) = g(3) = 1$, can be represented by (for example) the following diagram:



Notice that this diagram denotes the same *relation* as in (20), but the two represent different spans.

Another way to understand the correspondence is to notice that spans of finite sets can be seen as matrices with coefficients in \mathbb{N} . Because we identify isomorphic spans, the specific label of each witness in the apex plays no role. In this sense a span just keeps track of *how many ways* two elements in each of its legs are related. More precisely, given the span $n \xleftarrow{f} A \xrightarrow{g} m$, we can represent it as an $m \times n$ matrix whose (i, j) -th coefficient is the cardinality of $\{a \in A \mid f(a) = j, g(a) = i\}$. The diagrammatic calculus for matrices (over any semiring) will be explained in more details in Example 5.14 below. This perspective is also developed in [24], where the authors study some of the algebraic properties of the SMC of spans and their dual, *cospans*, which we will introduce next.

Example 5.11 (Cospans). The category $\mathbf{Cospan}(\mathbf{Set})$ has sets as objects and morphisms $X \rightarrow Y$ given by pairs of maps $f : X \rightarrow A$ and $g : Y \rightarrow A$ with the same set A as codomain, which we write as $X \xrightarrow{f} A \xleftarrow{g} Y$. The composition of two cospans is obtained by computing what is called *the pushout* of g and p below and composing the resulting outer two functions on each side:



where $A +_Y B = (\{(a, 1) \mid a \in A\} \cup \{(b, 2) \mid b \in B\}) / \sim$ where \sim is the equivalence relation defined by $(a, 1) \sim (b, 2)$ iff $a = g(y)$ and $b = p(y)$ for some $y \in Y$, and i_1, i_2 are the obvious inclusion maps of A and B into $A +_Y B$. Then, the composition of $X \xrightarrow{f} A \xleftarrow{g} Y$ with $Y \xrightarrow{p} B \xleftarrow{q} Z$ is $X \xrightarrow{i_1 \circ f} A \times_Y B \xleftarrow{i_2 \circ q} Y$. For a set X , the identity cospan is $X \xrightarrow{id_X} X \xleftarrow{id_X} X$. As for spans, this operation is not strictly associative or unital. To make $\mathbf{Cospan}(\mathbf{Set})$ into a bona fide category, we need to identify all isomorphic cospans: two cospans $X \xrightarrow{f} A \xleftarrow{g} Y$ and $Y \xrightarrow{p} B \xleftarrow{q} Z$ are isomorphic when there is a bijection $h : A \rightarrow B$ that makes the two resulting triangles commute.

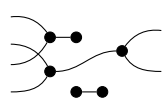
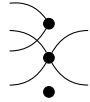
Like for spans, we can equip cospans with the structure of a SMC—this time with the disjoint sum as monoidal product. Take $X_1 + X_2$ to be the monoidal product on objects and, on morphisms, $(X_1 \xrightarrow{f_1} A_1 \xleftarrow{g_1} Y_1) \otimes X_2 \xrightarrow{f_2} A_2 \xleftarrow{g_2} Y_2 = X_1 + X_2 \xrightarrow{f_1+f_2} A_1 + A_2 \xleftarrow{g_1+g_2} Y_1 + Y_2$ where $(f_1 + f_2)(x, i) = f_i(x)$, with the empty set as unit and the symmetry given by $X + Y \xrightarrow{id} X + Y \xleftarrow{\iota_2+\iota_1} Y + X$, where $\iota_1 : X \hookrightarrow X + Y$, $\iota_2 : Y \hookrightarrow X + Y$ are the injections into the first and second component given respectively by $\iota_1(x) = (x, 1)$ and $\iota_2(y) = (y, 2)$.

Once again, this is a hypergraph category. The chosen Frobenius structure is given for each set X by

$$\begin{aligned} \left[\begin{array}{c} X \\ \bullet \\ X \end{array} \right] &= X \xrightarrow{id_X} X \xleftarrow{\nabla_X} X + X & \left[\begin{array}{c} X \\ \bullet \end{array} \right] &= X \xrightarrow{id_X} X \xleftarrow{0_X} \emptyset \\ \left[\begin{array}{c} X \\ X \end{array} \bullet \right] &= X + X \xrightarrow{\nabla_X} X \xleftarrow{id_X} X & \left[\begin{array}{c} X \\ \bullet \end{array} \right] &= \emptyset \xrightarrow{0_X} X \xleftarrow{id_X} 0 \end{aligned} \quad (22)$$

where $\nabla_X : X + X \rightarrow X$ is defined as before by $\nabla_X(x, i) = x$ and $0_X : \emptyset \rightarrow X$ is the unique map from the empty set to X . Notice the similarity (and differences!) with (21).

In fact, there is more than a coincidental relationship between cospans and hypergraph categories: **Cospan**(fSet), the category of cospans restricted to finite sets, is the *free hypergraph category on a single object* and no morphisms [80, 5.4], that is, on the signature $\Sigma = (\{\bullet\}, \emptyset)$. This means in particular that any cospan between finite sets (seen again as ordinals) can be represented as a diagram using only \curvearrowright , \bullet , \curvearrowleft , $\curvearrowright\bullet$. To build some intuition for this correspondence, take for example the pair $f : \{1, 2, 3, 4\} \rightarrow \{1, 2, 3\}$ and $g : \{1, 2\} \rightarrow \{1, 2, 3\}$, given by $f(1) = f(3) = 1$, $f(2) = f(4) = 2$ and $g(1) = g(2) = 2$; this cospan can be depicted as:


or, using spider notation


(23)

The rule of thumb is easy to formulate: each element of the apex of the cospan—here, $\{1, 2, 3\}$ —corresponds to one connected network of black generators, and a boundary point is connected to a black dot if it is mapped to the corresponding apex element by (one of the legs of) the cospan. There is only one way of forming such a network from the generators modulo the axioms of special commutative Frobenius structures, by the spider theorem, a result we saw in Example 2.19.

Moreover, researchers noticed that many hypergraph categories can be seen as categories of cospans equipped with additional structure [50]. Interestingly, not *all* hypergraph categories can be described in this way. For that, we need the notion of *corelation*, which we cover next.

Example 5.12 (Corelations). After seeing the last few examples, it is natural to wonder: spans are to relations as cospans are to *what*? The answer is *equivalence relations*, also known as *corelations* in this context [37]. It turns out that we can organise equivalence relations into a SMC—into a hypergraph category, in fact.

A corelation $C : X \rightarrow Y$ is an equivalence relation (*i.e.* a reflexive, symmetric and transitive relation) over $X + Y$. Given two corelations $C : X \rightarrow Y$ and $D : Y \rightarrow Z$, their composition $C ; D : X \rightarrow Z$ is defined by glueing together equivalence classes from C and D along shared elements. To define it formally, we temporarily rename $C.D$ the usual composition of relations (*cf.* Example 5.3) and let R^* be the transitive closure of a relation R ; then $C ; D$ is the restriction of $C \cup D \cup (C.D)^*$ to elements of $X + Z$. Intuitively, two elements a and b are in the same equivalence class of $C ; D$ if there exists some sequence of elements of $X + Y + Z$, that are equivalent either according to C or D , starting with a and ending with b . The disjoint sum of sets can be extended to

correlations to give a monoidal product. Once more, this category is not only symmetric monoidal, but it is a hypergraph category with Frobenius structure on each set X given by

$$\begin{aligned} \left[\begin{array}{c} X \\ \bullet \\ X \end{array} \right] &= \{((x, i), (x, j)) \mid i, j \in \{0, 1, 2\}\} & \left[\begin{array}{c} X \\ \bullet \end{array} \right] &= X \\ \left[\begin{array}{c} X \\ X \\ \bullet \end{array} \right] &= \{((x, i), (x, j)) \mid i, j \in \{0, 1, 2\}\} & \left[\begin{array}{c} \bullet \\ X \end{array} \right] &= X \end{aligned} \quad (24)$$

As we did for cospans, it is easy to represent any corelation between finite sets as a diagram using only \curvearrowright , \bullet , \curvearrowleft , \bullet . For example, the corelation $\{1, 2, 3\} \rightarrow \{1, 2\}$ given by the two equivalence classes $\{1, 2, 3\} \rightarrow \{1, 2\}$ and $\{1, 2, 3\} \rightarrow \{1, 2\}$ can be depicted by any of the following diagrams, using spider notation:



Notice that the first is the same diagram as in (23). The individual black dot, which represented an element of the apex of the cospan that was not in the image of any of the two leg maps, is absent from the second diagram. These two diagrams represent the same corelation, since an isolated black dot represents an empty equivalence class: $[\bullet] = [\bullet \curvearrowright \bullet] = \emptyset$. In diagrammatic terms, this means that we can always remove networks of black generators that are not connected to any boundary points using the fact that $\bullet \curvearrowright \bullet = \square$.

Much like relations can be seen as jointly injective spans, *i.e.* injective maps $R \hookrightarrow X \times Y$, corelations $C : X \rightarrow Y$ are—dually—jointly *surjective* cospans, *i.e.* surjective maps $X + Y \twoheadrightarrow S$. We have already seen that, given a span $A \rightarrow X \times Y$, one can extract a relation R by factorising it into $A \twoheadrightarrow R \hookrightarrow X \times Y$, a surjective map followed by an injective map. Similarly, one can obtain a corelation from a cospan by keeping only the surjective map in the factorisation of the corresponding map $X + Y \rightarrow S$. As we have just seen, diagrammatically, this corresponds to removing isolated black dots. In category theory, the factorisation of Set maps into an surjective map followed by an injective one can be abstracted into a notion called a *factorisation system*. It turns out that corelations can be defined for different factorisation systems than the surjective-injective one. Moreover, they can be decorated with additional structure. In fact, these two generalisations are so powerful that every hypergraph category can be constructed as a category of *decorated corelations* [51, 52].

Example 5.13 (Linear maps, \otimes). The category \mathbf{fVect} of finite-dimensional vector spaces (over some chosen field \mathbb{K}) and linear maps is also a symmetric monoidal category, in at least two different ways. This example deals with the tensor product, while the next one considers the direct product.

We will not go over the rigorous definition of the tensor product of vector spaces here; suffices to say that $X_1 \otimes X_2$ can be defined as a quotient of the free vector space over $X_1 \times X_2$ that make \otimes bilinear. On morphisms, it is uniquely specified as the linear map $(f_1 \otimes f_2)$ that satisfies $(f_1 \otimes f_2)(u_1 \otimes u_2) = f_1(u_1) \otimes f_2(u_2)$. This defines a SMC, with unit the field \mathbb{K} itself, since $X \otimes \mathbb{K} \cong X$, and symmetry the map fully characterised by $\sigma(u_1 \otimes u_2) = u_2 \otimes u_1$.

Crucially, this SMC is *not* Cartesian. The intuition here is that, when we choose a comultiplication operation \curvearrowright over X , we also choose some set of elements $v \in X$ that this operation copies, *i.e.* that verify:

$$\left[\begin{array}{c} X \\ \boxed{v} \curvearrowright X \end{array} \right] = \left[\begin{array}{c} X \\ \boxed{v} \\ \boxed{v} \end{array} \right] \quad (25)$$

But then, given two such copyable elements v_1, v_2 , consider their sum $u = v_1 + v_2$ —we should have

$$\left[\left[u \right] \begin{array}{c} \xrightarrow{X} \\ \bullet \\ \xleftarrow{X} \end{array} \right] = \left[\left[v_1 \right] \begin{array}{c} \xrightarrow{X} \\ \bullet \\ \xleftarrow{X} \end{array} \right] + \left[\left[v_2 \right] \begin{array}{c} \xrightarrow{X} \\ \bullet \\ \xleftarrow{X} \end{array} \right] = \left[\left[v_1 \right] \begin{array}{c} \xrightarrow{X} \\ \boxed{v_1} \\ \xleftarrow{X} \end{array} \right] + \left[\left[v_2 \right] \begin{array}{c} \xrightarrow{X} \\ \boxed{v_2} \\ \xleftarrow{X} \end{array} \right] = v_1 \otimes v_1 + v_2 \otimes v_2$$

On the other hand

$$\left[\left[u \right] \begin{array}{c} \xrightarrow{X} \\ \boxed{u} \\ \xleftarrow{X} \end{array} \right] = u \otimes u = (v_1 + v_2) \otimes (v_1 + v_2) = v_1 \otimes v_1 + v_1 \otimes v_2 + v_2 \otimes v_1 + v_2 \otimes v_2$$

Thus, no linear map can copy all elements of a given vector space, as is required for the comonoid structure of a Cartesian category.

In fact, $(\mathbf{fVect}, \otimes)$ is *compact closed* (recall that the requirement of Cartesian-ness and compact closed-ness are incompatible in the sense explained in Section 4). The dual of a vector space defined to be its algebraic dual in the usual sense: A^* , the space of linear maps $A \rightarrow \mathbb{K}$. Then, the cap on a vector space X is the unique linear map $X^* \otimes X \rightarrow \mathbb{K}$ that satisfies $\left[\begin{array}{c} \nearrow \\ \searrow \end{array} \right] (f \otimes v) = f(v)$ (also known as the evaluation map). The cup is its adjoint: to describe it explicitly, we need to pick a basis $\{e_i\}_i$ of X and a dual basis $\{f_i\}_i$ of X^* in the sense that $f_i(e_j) = 1$ if $i = j$ and 0 otherwise; $\left[\begin{array}{c} \nwarrow \\ \nearrow \end{array} \right]$ is then the map $\mathbb{K} \rightarrow X \times X^*$ given by extending $1 \mapsto \sum_i e_i \otimes f_i$ by linearity. In summary, using the bases $\{e_i\}_i$ and $\{f_i\}_i$ for both cups and caps, we have:

$$\left[\begin{array}{c} \nwarrow \\ \nearrow \end{array} \right] (k) = \sum_i k(e_i \otimes f_i) \quad \left[\begin{array}{c} \nearrow \\ \searrow \end{array} \right] \left(\sum_{i,j} \lambda_{i,j} (e_i \otimes f_j) \right) = \sum_i \lambda_{i,i} f_i(e_i)$$

Note however, that the resulting maps are independent of the specific choice of bases. With these expressions, we can verify the yanking equation for \nwarrow and \nearrow . Let u be some element of X such that $v = \sum_i \lambda_i e_i$; we have:

$$\begin{aligned} \left[\begin{array}{c} \nwarrow \\ \nearrow \end{array} \right] (v) &= \left(\left[\begin{array}{c} \nwarrow \\ \nearrow \end{array} \right] \otimes \mathbb{I} \right) ; \left(\mathbb{I} \otimes \left[\begin{array}{c} \nearrow \\ \searrow \end{array} \right] \right) (v) \\ &= \left(\mathbb{I} \otimes \left[\begin{array}{c} \nearrow \\ \searrow \end{array} \right] \right) \left(\left(\sum_i (e_i \otimes f_i) \right) \otimes v \right) \\ &= \sum_i f_i(v) e_i \\ &= \sum_i f_i \left(\sum_j \lambda_j e_j \right) e_i \\ &= \sum_i \sum_j \lambda_j f_j(e_i) e_i \\ &= \sum_i \lambda_i e_i = v \end{aligned}$$

Note also that, in this SMC, the elements of a vector space X are precisely the diagrams $\mathbb{K} \rightarrow X$.

As we have seen, every compact closed category is also traced. In fact, the name (partial) *trace* comes from linear algebra, where the trace $\text{Tr} f$ of a linear map $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is the sum of the diagonal coefficients of its matrix representation in any basis. Thus, we expect that,

$$\begin{array}{c} \curvearrowright \\ \boxed{f} \\ \curvearrowleft \end{array} = \text{Tr} f = \text{Tr} A = \sum_i a_{ii}$$

where $A = (a_{ij})$ is the matrix that represents the action of f on some chosen basis. It is a nice exercise to show that this is indeed the case. It is then immediate to derive certain well-known properties of the trace in linear algebra, such as $\text{Tr}(AB) = \text{Tr}(BA)$ (or, more generally, that it is invariant under circular shifts).

Another important feature (with consequences, among other areas, in quantum theory) is that commutative and special Frobenius structure in $(\mathbf{fVect}, \otimes)$ correspond to a choice of a basis for the supporting vector space [35, Section 6]. We have seen above that—even if there is no linear copying map for all the elements of a vector space—when we choose a comultiplication operation $\text{---}\bullet\text{---}$ over X we also choose some set of elements $v \in X$ that $\text{---}\bullet\text{---}$ copies. Conversely, given any basis, we can define a comonoid operation that copies its elements, *i.e.*, whose comultiplication and counit are defined respectively by extending the following maps by linearity: $e_i \mapsto e_i \otimes e_i$ and $e_1 \mapsto 1$. What about the monoid? A monoid in $(\mathbf{fVect}, \otimes)$ is more commonly known as an *algebra*. Any basis defines not only a comonoid but an algebra given by extending the comparison map $e_i \otimes e_j \mapsto \delta_{ij}^i e_i$ by linearity. Not only that, the corresponding monoid-comonoid pair satisfies the Frobenius axioms and define a special and commutative Frobenius structure. Conversely, the copyable states of any commutative and special Frobenius structure form a basis of X . The last direction is more difficult to prove and we will not do so here. Instead, we refer the interested reader to the lectures notes of Vicary and Heunen, who deal with a related case in details [69, Chapter 5] and use string diagrams throughout.

Finally, a brief historical note: one of the earliest instances of string diagrams are *Penrose graphical notation* [94] for working with tensors, which are precisely string diagrams for $(\mathbf{fVect}, \otimes)$, later systematised and generalised in [73].

Example 5.14 (Matrices, \oplus). Another possible monoidal product is given by the direct sum of vector spaces $X_1 \oplus X_2$ on objects and by $(f_1 \oplus f_2)(x_1, x_2) = (f_1(x_1), f_2(x_2))$ on morphisms. The unit of the product is the vector space $\{0\} \cong \mathbb{K}^0$ and, with the symmetry given by $\sigma_X^Y(x, y) = (y, x)$, the resulting structure is a SMC. It is well-known that isomorphic finite-dimensional vector spaces are uniquely identified by their dimension. Therefore, in the same way that we identified finite sets with finite ordinals, we will restrict our attention to the subcategory of \mathbf{fVect} whose objects are \mathbb{K}^n for some $n \in \mathbb{N}$. We can go even further: give a linear map $\mathbb{K}^m \rightarrow \mathbb{K}^n$, we can identify it with its representation in the canonical bases of \mathbb{K}^m and \mathbb{K}^n . We call $\text{Mat}_{\mathbb{K}}$ the category whose objects are natural numbers (representing the dimension of a vector space) and morphisms $m \rightarrow n$ are $n \times m$ matrices (notice the reversal). Nothing is lost, since $\text{Mat}_{\mathbb{K}}$ and \mathbf{fVect} are equivalent.

The SMC $(\text{Mat}_{\mathbb{K}}, \oplus)$ is Cartesian with the canonical comonoid structure over some vector space X is given by

$$\left[\begin{array}{c} n \\ \text{---}\bullet\text{---} \\ n \end{array} \right] (x) = (x, x) \quad \left[\begin{array}{c} n \\ \bullet \\ n \end{array} \right] (x) = \bullet$$

Since linear maps are maps with extra structure, this comonoid is inherited from Set (cf. Example 5.2) and the proof that this monoidal category is Cartesian is exactly the same. $\text{Mat}_{\mathbb{K}}$ also contains a remarkable monoid structure on every object, given by addition and zero:

$$\left[\begin{array}{c} n \\ \text{---}\circ\text{---} \\ n \end{array} \right] (x_1, x_2) = x_1 + x_2 \quad \left[\begin{array}{c} n \\ \circ \\ n \end{array} \right] = 0$$

Note that the comonoid and monoid do not interact to form a Frobenius structure but a *bimonoid* (cf. Section 2.18). In fact, we have even more structure: this category satisfies the axioms of *biproduct categories* (Section 4.2.7). This means that, maps do not only satisfy the (dup) and (del) axioms

of Cartesian categories, but the dual axioms of coCartesian categories (codup) and (codel): in semantic terms, these last two axioms are simply implied by linearity: all maps preserve addition. Note that this structure is very similar to that of relations with the disjoint sum as monoidal product (cf. Example 5.7), the chief difference being that the bimonoid is not idempotent for matrices over a field. The close similarity between the two cases comes from the fact that relations can be seen as matrices, not over field, but over the semiring of the Booleans.

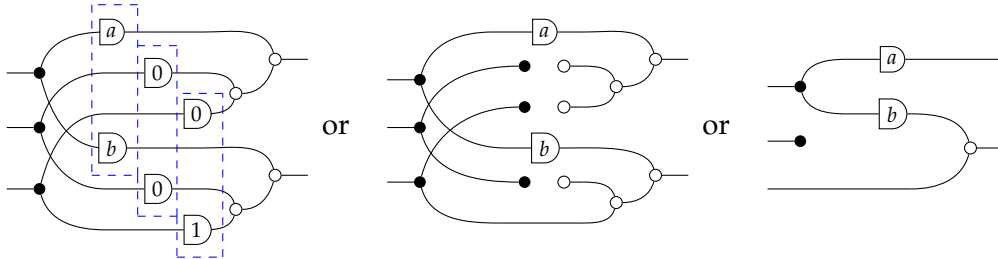
With the bimonoid above, we are very close to being able to express matrices diagrammatically.

As before, we will use $\frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$, $\frac{1}{\bullet} \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$ and $\frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$, which we will simply write as $\frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$, $\frac{1}{\bullet} \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$ and $\frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$. Contrary to the case of relations, we cannot express arbitrary matrices with just these; we need to add generating diagrams $\frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$ for scalar multiplication by some $a \in \mathbb{K}$, interpreted as $\llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket (x) = ax$. There are few special cases of interest: multiplying by 1 is the same as the identity, so $\llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket = \llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket$, and multiplying by zero always yields zero, so $\llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket = \llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket$.

Putting all these ingredients together, we are now ready to represent matrices. An $n \times m$ matrix $A = (a_{ij})$ corresponds to a diagram d with m wires on the left and n wires on the right—the left ports can be interpreted as the columns and the right ports as the rows of A . The left j -th port is connected to the i -th port on the right through an a -weighted wire whenever coefficient a_{ij} is a scalar $a \in \mathbb{K}$. When coefficient a_{ij} is 0, they are disconnected. In addition, given that $\llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket = \llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket$, we can simply draw the connection as a plain wire when $a_{ij} = 1$ and since $\llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket = \llbracket \frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet} \rrbracket$ we can also omit a connecting wire when $a_{ij} = 0$. Conversely, given a diagram, we recover the matrix by summing weighted paths from left to right ports. For example, the matrix

$$A = \begin{pmatrix} a & 0 & 0 \\ b & 0 & 1 \end{pmatrix}$$

can be represented by any of the following diagrams, which are all semantically equal (i.e., represent the same matrix):



The dotted boxes in the diagram on the left represent the columns of the corresponding matrix. As we will see, we can then quotient the diagrammatic syntax by an equational theory that makes these three equal. In fact, we can define an equational theory E such that the diagrams modulo \equiv form a SMC isomorphic to $\text{Mat}_{\mathbb{K}}$. Such a theory is called *complete*. We will come back to completeness of equational theories in Section 5.2 and give a complete theory for matrices in Example 5.18.

Finally, everything we have claimed in this example, would have worked as well with an arbitrary semiring R , instead of a field: we would just need to consider matrices with coefficients in R and have diagrams $\frac{1}{\bullet} \frac{1}{\square} \frac{1}{\bullet}$ for all $a \in R$.

Example 5.15 (Linear relations, \times). In the last two examples, we have considered linear *maps* with different monoidal products. It is possible to extend the notion of linearity to *relations*: given two vector spaces X and Y , a linear relation $X \rightarrow Y$ is a linear subspace of $X \oplus Y$, i.e. a subset of the direct sum that is closed under linear combinations. The composition (as relations) of two linear relations is still a linear relation (exercise), and the identity relation is linear. Therefore, linear

relations can be organised into a category. We call $\text{LinRel}_{\mathbb{K}}$, the category whose objects are natural numbers and morphisms $m \rightarrow n$ are linear relation $\mathbb{K}^m \rightarrow \mathbb{K}^n$. With the Cartesian product, $\text{LinRel}_{\mathbb{K}}$ become a SMC, with unit and symmetry the same as those of $\text{Mat}_{\mathbb{K}}$ (cf. previous example).

This SMC has a very rich structure. Firstly, just like any function can be seen as a relation, any linear map f can be seen as a linear relation $\text{Graph}(f)$, by taking its graph: $\text{Graph}(f) := \{(x, y) \mid y = f(x)\}$. Thus, $\text{LinRel}_{\mathbb{K}}$ also contains the diagrams that allowed us to depict linear maps/matrices diagrammatically in the previous example, namely $\text{---}\bullet\text{---}$, $\text{---}\bullet\text{---}$, $\text{---}\circ\text{---}$, $\text{---}\circ\text{---}$ for any choice of underlying vector space \mathbb{K}^n . Their interpretation as relation is given by the graph of their corresponding maps:

$$\begin{aligned} \left[\begin{array}{c} n \\ \text{---}\bullet\text{---} \\ n \end{array} \right] &= \{(x, (x, x)) \mid x \in X\} & \llbracket \text{---}\bullet \rrbracket &= \{(x, \bullet) \mid x \in X\} \\ \left[\begin{array}{c} n \\ \text{---}\circ\text{---} \\ n \end{array} \right] &= \{((x_1, x_2), x_1 + x_2) \mid x_1, x_2 \in X\} & \llbracket \text{---}\circ \rrbracket &= \{(0, \bullet)\} \end{aligned}$$

Interestingly, the converse of these relations are also linear, and we adopt the mirror image of the corresponding diagrams to depict them: $\text{---}\bullet\text{---}$, $\text{---}\bullet\text{---}$, $\text{---}\circ\text{---}$, $\text{---}\circ\text{---}$, with semantics given by

$$\begin{aligned} \left[\begin{array}{c} n \\ \text{---}\bullet\text{---} \\ n \end{array} \right] &= \{((x, x), x) \mid x \in X\} & \llbracket \bullet\text{---} \rrbracket &= \{(\bullet, x) \mid x \in X\} \\ \left[\begin{array}{c} n \\ \text{---}\circ\text{---} \\ n \end{array} \right] &= \{(x_1 + x_2, (x_1, x_2)) \mid x_1, x_2 \in X\} & \llbracket \bullet\text{---}\circ \rrbracket &= \{(\bullet, 0)\} \end{aligned}$$

Notice that these are the same relations as the first two, with the pairs flipped. We can do this for any map f : let $\text{coGraph}(f) := \{(y, x) \mid f(x) = y\}$. If f is linear, its cograph will also be a linear relation. This duality will translate into a pleasant symmetry of the equational theory, which we will cover in Example 5.20.

Example 5.16 (Monotone relations, \times). So far all the examples of compact closed categories we have covered (spans, relations, cospans, corelations) have been hypergraph categories. Of course, there are compact closed categories where the compact structure does not come from some chosen Frobenius structure. The category of *monotone relations* is one such example. It has pre-ordered sets as objects (that is, sets equipped with a pre-order: a reflexive and transitive binary relation), and, as morphisms $(X, \preceq) \rightarrow (Y, \preceq)$, relations $R \subseteq X \times Y$ that preserve the order in the following sense: if $(x, y) \in R$ and $x' \preceq x, y \preceq y'$ then $(x', y') \in R$. The composition of monotone relation is the same as the usual composition of relations (recalled in Example 5.3). Since the composition of two monotone relations is monotone, they form a category, with identity on each object (X, \preceq) given by the pre-order relation $\preceq \subseteq X \times X$ itself.

As for plain relations, the Cartesian product defines a monoidal product: $(X, \preceq) \times (Y, \preceq) := (X \times Y, \preceq \times \preceq)$ with $(x, y) \preceq \times \preceq (x', y')$ iff $x \preceq x'$ and $y \preceq y'$. The unit is still the singleton set $1 = \{\bullet\}$ with the only possible pre-order.

As we anticipated, this category is compact closed. The dual $(X, \preceq)^*$ of (X, \preceq) is (X, \succeq) , i.e., the same underlying set with the opposite pre-order relation $\succeq := \preceq^{op}$. The cups and caps on each object are then given by

$$\left[\begin{array}{c} \text{---}\cup\text{---} \\ \text{---}\cup\text{---} \end{array} \right] = \{(\bullet, (x', x)) \mid x \preceq x'\} \quad \left[\begin{array}{c} \text{---}\cap\text{---} \\ \text{---}\cap\text{---} \end{array} \right] = \{((x, x'), \bullet) \mid x' \preceq x\}$$

Let us check one of the defining equations of compact closed categories:

$$\begin{array}{c} \text{---}\cup\text{---} \\ \text{---}\cup\text{---} \\ \text{---}\cap\text{---} \end{array} = \text{---}\text{---}$$

The lhs of this equation has the following semantics:

$$\begin{aligned}
\llbracket \text{S} \rrbracket &= (\llbracket \text{C} \rrbracket \times \llbracket \text{A} \rrbracket) ; (\llbracket \text{A} \rrbracket \times \llbracket \text{B} \rrbracket) \\
&= (\{(\bullet, (x, x')) \mid x \preceq x'\} \times \preceq) ; (\preceq \times \{(x, x'), \bullet\} \mid x \preceq x'\}) \\
&= (\{(x, (x_1, x_2, x_3)) \mid x_2 \preceq x_1 \wedge x \preceq x_3\}) ; (\{(x_1, (x_2, x_3), x') \mid x_1 \preceq x' \wedge x_3 \preceq x_2\}) \\
&= \{(x, x') \mid \exists x_1 \exists x_2 \exists x_3 [x \preceq x_3 \wedge x_3 \preceq x_2 \wedge x_2 \preceq x_1 \wedge x_1 \preceq x']\} \\
&= \{(x, x') \mid x \preceq x'\}
\end{aligned}$$

where the last step hold by transitivity of \preceq . This is clearly the same relation as \preceq itself, which is the identity on (X, \preceq) , as we wanted.

Finally, in this SMC, every partial order is equipped with a canonical monoid and comonoid. They are given by the following monotone relations:

$$\llbracket \text{A} \rrbracket = \{(x, (x'_1, x'_2)) \mid x \leq x'_1 \text{ and } x \leq x'_2\} \quad \llbracket \text{A} \rrbracket = \{(x, \bullet) \mid x \in X\} \quad (26)$$

$$\llbracket \text{B} \rrbracket = \{((x_1, x_2), x') \mid x_1 \leq x' \text{ and } x_2 \leq x'\} \quad \llbracket \text{B} \rrbracket = \{(\bullet, x) \mid x \in X\} \quad (27)$$

Note that they are very similar to their standard relational cousins from Example 5.3. In fact, the former can be seen as the latter, composed with the partial order relation on each branch, to make them monotone: for example, if we momentarily reinterpret $\llbracket - \rrbracket$ as a mapping into Rel, we have

$$\llbracket \text{A} \rrbracket = \llbracket \text{A} \rrbracket ; \llbracket \text{A} \rrbracket ; \llbracket \text{A} \rrbracket$$

since $\llbracket \text{A} \rrbracket = \preceq = \{(x, x') \mid x \preceq x'\}$.

However, $\text{A}, \text{B}, \text{C}, \text{D}$ do not form a Frobenius structure, nor do they give rise to a (co)Cartesian structure. Their interaction is still interesting and can be axiomatised, but doing so requires turning to theories with *inequalities* rather than just equalities. We will look at these briefly in Section 6.3 and at monotone relations again in Example 6.4.

5.2 Soundness and Completeness

Recall that we write $\text{Free}_X(\Sigma)$ for the free X -category over some signature. Let $\llbracket - \rrbracket : \text{Free}_X(\Sigma) \rightarrow \text{Sem}$ be an X -preserving functor into some X -category Sem working as our semantics. In this context, it is common to study those diagrams that the semantics identifies; in other words, we are interested in equalities $\llbracket c \rrbracket = \llbracket d \rrbracket$ for diagrams c and d . Sometimes (if we are lucky) it is possible to characterise semantic equality completely by a set of axioms, called a theory.

We have already covered (symmetric monoidal) theories in Section 2.1. The same notion can be extended straightforwardly to other categories: Cartesian, traced, hypergraph etc. In all these cases, an (equational) theory E is still a set a set of pairs of diagrams of $\text{Free}_X(\Sigma)$ with the same arity and co-arity and \equiv is the smallest congruence relation (w.r.t. $;$ and \otimes) containing E . We are now in a position to study their compatibility with a given semantics. For this, we will need a bit more terminology: we say that a theory E is *sound* (for $\llbracket - \rrbracket$) whenever $c \equiv d$ implies that $\llbracket c \rrbracket = \llbracket d \rrbracket$; furthermore E is said to be *complete* if the reverse implication holds. When $c \equiv d$ iff $\llbracket c \rrbracket = \llbracket d \rrbracket$ the theory is sound and complete and sometimes called an *axiomatisation* of the target category. The reader will often read that a theory is “complete for Sem ”, rather than $\llbracket - \rrbracket$, when the functor $\llbracket - \rrbracket$ is obvious and therefore left implicit.

Moreover, as we saw in Section 2.1, given a signature and a theory we can form the X category $\text{Free}_X(\Sigma, E)$ obtained by quotienting the free X category $\text{Free}_X(\Sigma)$ by the equivalence relation over diagrams given by $\stackrel{E}{=}$. There is then an X functor $q : \text{Free}_X(\Sigma) \rightarrow \text{Free}_X(\Sigma, E)$ witnessing this quotient.

We may also wonder what the expressive power of our diagrammatic language is. In terms of X categories, we look to characterise precisely the image $\text{Im}(\llbracket \cdot \rrbracket)$ of the syntax via $\llbracket \cdot \rrbracket$.

The situation for a sound and complete theory is summarised in the commutative diagram below. Soundness simply means that $\llbracket \cdot \rrbracket$ factors as $s \circ q$ through $\text{Free}_X(\Sigma, E)$ and completeness means that s is a faithful X functor.

$$\begin{array}{ccc} \text{Free}_X(\Sigma, E) & \xrightarrow{\cong} & \text{Im}(\llbracket \cdot \rrbracket) \\ \uparrow q & \searrow s & \downarrow i \\ \text{Free}_X(\Sigma) & \xrightarrow{\llbracket \cdot \rrbracket} & \text{Sem} \end{array}$$

Example 5.17 (Relations, +). Recall from Example 5.7 that the SMC $(\text{fRel}, +)$ of relations between finite sets with the disjoint sum is a symmetric monoidal category. We can give ourselves the signature

$$\Sigma = (\{1\}, \{-\bullet\curvearrowright, -\bullet\curvearrowleft, \circ\curvearrowright, \circ\curvearrowleft\})$$

and quotient Σ -diagrams by the axioms of an idempotent, commutative bimonoid, which we recall below:

(28)

This equational theory turns out to be complete $\llbracket - \rrbracket$ as defined in Example 5.7. In other words, any two diagrams c, d made from $-\bullet\curvearrowright, -\bullet\curvearrowleft, \circ\curvearrowright, \circ\curvearrowleft$ are equal modulo the axioms in (28) iff they denote the same relation. The proof of this fact is typical: it works by showing how, given an arbitrary Σ -diagram d , we can rewrite it to some normal form, using only the equations of (28). The chosen normal form is the one from Example 5.7, from which the corresponding semantic entity—here, the corresponding relation—can be recovered uniquely. The fact that any diagram can be rewritten to a normal form using only the axioms above, is typically proven by induction, consider each individual cases, much like normalisation proofs in PL theory or cut elimination proofs in logic. And, much like these, they tend to be quite tedious and combinatorial.

Note that, depending on the context and the structure that one takes as a given, there are several ways of formulating the result above. For example, we could have also said that the *Cartesian* theory of an idempotent, commutative monoid is complete for $\llbracket - \rrbracket$, (now redefined to have the free Cartesian category over the specified theory as domain).

Example 5.18 (Matrices, \oplus). As we have already said, relations (resp. spans) with the disjoint sum as monoidal product can be seen as matrices with Boolean (resp. natural) coefficients. In fact, the equational theory of the previous example can be modified easily to accommodate any semiring.

For the syntax, we've already seen in Example 5.14, that it suffices to add a generator for each scalar $r \in R$, giving the signature

$$\Sigma = (\{1\}, \{-\bullet, -\bullet, \circ, \circ\} \cup \{-\boxed{r} \mid r \in R\})$$

The equational theory is very similar to that of the previous example. It has all axioms of (28), *except* the last one, $-\bullet \circ = -$ (which encodes $x + x = x$, a specific feature of the Boolean semiring). In addition, we need axioms that encode the additive and multiplicative structure of the chosen semiring R , namely:

$$\begin{array}{ccc} \begin{array}{c} \boxed{r} \\ \circ \\ \boxed{s} \end{array} = \boxed{r+s} & -\bullet \circ = \boxed{0} \\ \boxed{r} \boxed{s} = \boxed{rs} & - = \boxed{1} \end{array} \quad (29)$$

Furthermore, we need to make sure that the scalars can be copied/deleted and that scalar multiplication distributes over addition; we can obtain these from the usual (dup)-(del) and (codup)-(codel) for scalars:

$$\begin{array}{ccc} \boxed{r} \bullet = \bullet \begin{array}{c} \boxed{r} \\ \boxed{r} \end{array} & \boxed{r} \bullet = \bullet \\ \begin{array}{c} \boxed{r} \\ \boxed{r} \end{array} \circ = \circ \boxed{r} & \circ \boxed{r} = \circ \end{array} \quad (30)$$

Taken with the axioms of (28) minus the last one, the axioms listed in (29)-(30) give a complete theory for matrices over R : diagrams modulo these equations are equal iff they denote the same matrix.

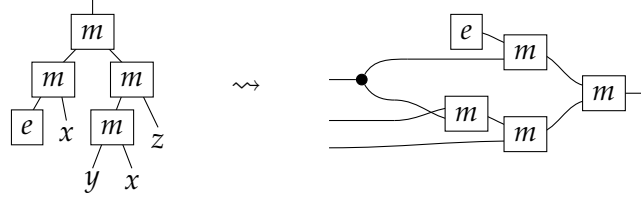
From this general result, combined with the equivalence between spans and matrices with coefficients in \mathbb{N} , we can derive the following corollary: the free biproduct category over a single generating object (and no morphism) is an axiomatisation of the SMC $(\mathbf{Span}(\mathbf{fSet}), +)$ (Example 5.10).

Remark 5.19 (Algebraic theories and Cartesian categories, reprise). *One often hears that string diagrams are a resource-sensitive syntax. This somewhat cryptic incantation means something more simple than it lets on: instead of variables, diagrams use wires. While variables can be used more (or less) than once—names cost little, at least when we are not concerned with computer implementations¹³—there is not necessarily a natural way to copy wires in order to plug them into multiple diagrams. We saw above that some diagrammatic languages allow copying and erasing of wires. It turns out that the diagrammatic syntax of Cartesian categories is the diagrammatic counterpart of the standard symbolic notation for algebraic theories. Key in this correspondence is the ability to copy and delete arbitrary diagrams using the wire splitting and ending. These diagrammatic laws formalise the correspondence between composition and the all-important logical operation of substitution. Let us see the correspondence on a simple example; the general formal correspondence between algebraic theories and Cartesian monoidal categories is worked out (for the single-sorted case) in [23].*

Consider the algebraic theory of monoids. It can be presented by two generating operations, $m(-, -)$ of arity 2 and e of arity 0 (a constant) satisfying the following three axioms: $m(m(x, y), z) = m(x, m(y, z))$ and $m(e, x) = x = m(x, e)$. Terms of this algebraic theory are syntax trees whose leaves are labelled with

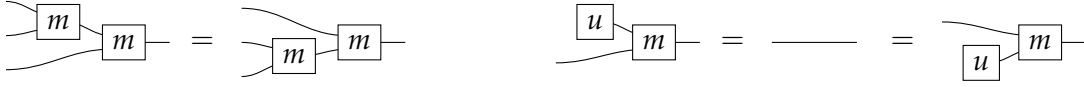
¹³Quantum computers present in this respect an interesting challenge, as quantum information, unlike classical one, cannot be perfectly copied. This is one of the starting point to the diagrammatic approach to quantum theory, see e.g. [2]

variable names, as on the left below.

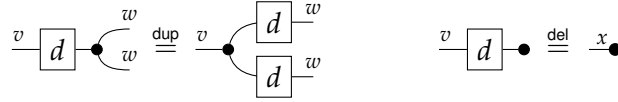


By simply turning the tree on its side and gathering all leaves labelled by the same variable with \bullet (or deleting those that we do not use with \bullet) we obtain the corresponding string diagram in the theory of Cartesian categories, as on the right above.

Two terms are equal if they one can be obtained from the other by applying a sequence of the defining equations of monoids. Similarly, two diagrams are equal if we can obtain one from the other by applying the following diagrammatic equalities (already encountered in Ex. 2.15):



with the axioms of Cartesian categories (cf. Section 4.2.5), which we recall here



where d can be any diagram.

What about models (in the usual algebraic sense) of algebraic theories? They are in one-to-one correspondence with Cartesian functors out of the free Cartesian category over the theory in question into the SMC (Set, \times) . For example, to specify such a functor for the Cartesian theory of monoids involves choosing a carrier set X and functions $\llbracket m \rrbracket : X \times X \rightarrow X$, $\llbracket e \rrbracket : 1 \rightarrow X$ of the appropriate arity that satisfy the relevant axioms.

Example 5.20 (Linear Relations). Recall the interpretation $\llbracket - \rrbracket$ of \bullet , \bullet , \circ , \circ and their duals \bullet , \bullet , \circ , \circ in terms of linear relations, from Example 5.15. The symmetry of the corresponding equational theory is particularly pleasing, though we will not reproduce it fully here. As for Example 5.18, \bullet , \bullet , \circ , \circ are precisely those relations that are the graph of some linear map (aka a matrix) and so the equational theory is the same as in that example: essentially, a commutative bimonoid, with additional axioms to encode scalar multiplication and addition. The nice thing is that their colour-swamp \bullet , \bullet , \circ , \circ satisfy exactly the same axioms. These two facts take care of all interactions between the black and white generators. It only remains to specify how diagrams of the same colour interact: each of the four diagrams of the same colours form an extraspecial commutative Frobenius structure (cf. Example 2.19). This theory is sometimes called the theory of *Interacting Hopf algebras* of IH for short. The reader will find further details in [22].

Remark 5.21. An interesting aspect of identifying the equational theory of a particular structure in a given semantic model is that we can find it again in a different model, and thereby identify seemingly unrelated algebraic objects as instances of the same abstract structure. For example, we have seen many different interpretations of Frobenius structures or bimonoids in different models (i.e. in different symmetric monoidal categories). Another common example is that of groups and Hopf algebras, both instances of bimonoids in different symmetric monoidal categories (sets and functions with the Cartesian product for the former, and vector spaces and linear maps with the tensor product for the latter). Even a complicated

theory such as IH occurs in other contexts than that of linear relations. Indeed, IH can be interpreted in the category of vector spaces with the tensor product as monoidal product, where its models are closely related to the notion of complementary observables in quantum physics [33, 45].

Example 5.22 (Cospans and corelations). We have mentioned that cospans with the disjoint sum as monoidal product (Example 5.11) not only form a hypergraph category, but are the *free hypergraph category on a single object* (and no additional morphism). In the language of this section, this means that the theory of a special commutative Frobenius structure is sound and complete for (the interpretation $\llbracket - \rrbracket$ into) $(\mathbf{Cospan}(\mathbf{fSet}), +)$ [80, 5.4]. The proof of this fact is essentially the spider theorem from Example 2.20. This theorem gives a normal form from which we can uniquely read the corresponding cospan: any diagram of the free hypergraph category on a single object is fully and uniquely characterised by the number of disconnected components (spiders) and to which of these each boundary point is connected. That’s it.

The only difference between corelations (also with the disjoint sum as monoidal product) and cospans is that the former do not allow empty equivalence classes (they are joint surjective cospans, as we saw). Diagrammatically, these correspond to isolated black dot: $\llbracket \bullet \rrbracket = \llbracket \bullet \bullet \rrbracket = \emptyset$. Since a network of black generators is fully characterised by its number of legs, there is—up to the laws of special commutative Frobenius structure—only one such network: the single dot $\bullet := \bullet \bullet$. Thus, we need only add an axiom to remove isolated dots to obtain a complete theory for the SMC corelations: $\bullet \bullet = \square$. The resulting theory is that of *extraspecial* commutative Frobenius structures [37, Theorem 1.1].

6 Other Trends in String Diagram Theory

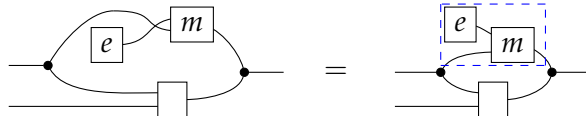
6.1 Rewriting

When reasoning about programs, *reductions* of a program p into another one q are important objects of study: such reduction may witness for instance the evaluation of p on a certain input (akin to β -reduction in the λ -calculus [9]), or more generically its transformation into a simpler program q . When considering programs as terms of an algebraic theory, reductions are typically formalised as *rewriting steps*: we may apply a *rewrite rule* $l \Rightarrow r$ inside p if the term l appears as a subterm of p , in which case we say that the rule has a *matching* in p ; if there is such a matching, then the outcome q of the rewriting step is the term $p[r/l]$ obtained by replacing r for l in p .

When it comes to string diagrams, rewriting presents additional challenges, that we do not experience on terms. The crux of the matter is matching: as string diagrams are invariant under certain topological transformations—crossing of wires, shifting of boxes, etc.—we would like matchings to exist or not regardless of which graphical presentation we choose for our string diagram. For example, consider the rule

$$\begin{array}{c} \boxed{e} \\ \diagup \quad \diagdown \\ \boxed{m} \end{array} \text{---} = \text{---}$$

We claim the rule has a matching on the string diagram below left. However, strictly speaking, the matching isolates a subterm only when we ‘massage’ the string diagram as on the right.



More formally, the point is that string diagrams are *equivalence classes* of terms, modulo the laws of SMCs. We want to be able to match a rewriting rule $l \rightsquigarrow r$ on a string diagram c whenever a term in the equivalence class of the string diagram l appears as a subterm in the equivalence class of the string diagram c .

Definition 6.1. Let Σ_0 be a signature, $l \rightsquigarrow r$ be a rewrite rule of Σ_0 -terms, and c be a Σ_0 -term. We say that c rewrites into d modulo E if

$$-\boxed{c}- =_E -\boxed{c'}- \text{ and } -\boxed{c'}- = -\boxed{c_1} \boxed{l} \boxed{c_2}- \text{ and } -\boxed{d}- = -\boxed{c_1} \boxed{r} \boxed{c_2}- \quad (31)$$

The standard case is when E are the laws of SMCs, but the notion can be adapted to fit other categorical structures where string diagrams occur—cf. Section 4. This definition seems reasonable enough from a mathematical viewpoint. However, it is completely unpractical when it comes to *implementing* string diagram rewriting. Exploring the space of all Σ -terms equivalent to a given one is an expensive computational task, and if done naively it may even not terminate, given that on principle there are infinitely many equivalent terms to be checked for a matching. This is an issue especially because rewriting is the way we formally reason about string diagrams in a computer: whenever we want to apply the equations of a theory such as those considered in Section 2.2, the first thing to do is orienting such equations as rewrite rules.

The solution from this impasse comes from the combinatorial interpretation of string diagrams, as introduced in Section 3. Recall that, under that interpretation, a string diagram is always mapped onto a *single* open hypergraph. In other words, if c is mapped onto g , and c and d are string diagrams equivalent modulo the laws of SMCs, then also d is mapped onto g . This feature makes open hypergraphs suitable data structures to reason about string diagram rewriting: if we want to rewrite with a rule $l \rightsquigarrow r$ and a string diagram c as above, we do not need to bother with the many equivalent syntactic presentations of these diagrams, but just need to consider the corresponding open hypergraphs. In fact, open hypergraphs do come with their own rewriting theory, called *double-pushout rewriting* [48]. The fundamental result linking double-pushout rewriting and syntactic rewriting is the following:

Theorem 6.2. c rewrites into d modulo scFrob if and only if $[c]_C$ rewrites into $[d]_C$ modulo double-pushout rewriting.

Theorem 6.2 is a consequence of the correspondence established by Theorem 3.5 between string diagrams modulo Frobenius structure and open hypergraphs. However, it is not completely satisfactory: we would like to interpret faithfully rewriting modulo the laws of SMCs, without the need of considering Frobenius equations too. It turns out it is still possible to obtain a correspondence, with a more restrictive notion of double-pushout rewriting, called *convex*.

Theorem 6.3. c rewrites into d modulo the laws of SMCs if and only if $[c]_C$ rewrites into $[d]_C$ modulo convex double-pushout rewriting.

We omit the details of the definition of convexity, which would require us delving into the theory of double-pushout rewriting, and refer the interested reader to the overview of string diagram rewriting offered in [14].

The above two theorems settle the question of rewriting for string diagrams in symmetric monoidal categories (Theorem 6.3) and hypergraph categories (Theorem 6.2). However, as we saw in Section 4, there are other structures for which we can draw string diagrams considered both of lower and higher complexity. Among the ones we have covered here, the question is not settled for monoidal, braided monoidal, traced monoidal, (self-dual) compact closed, and

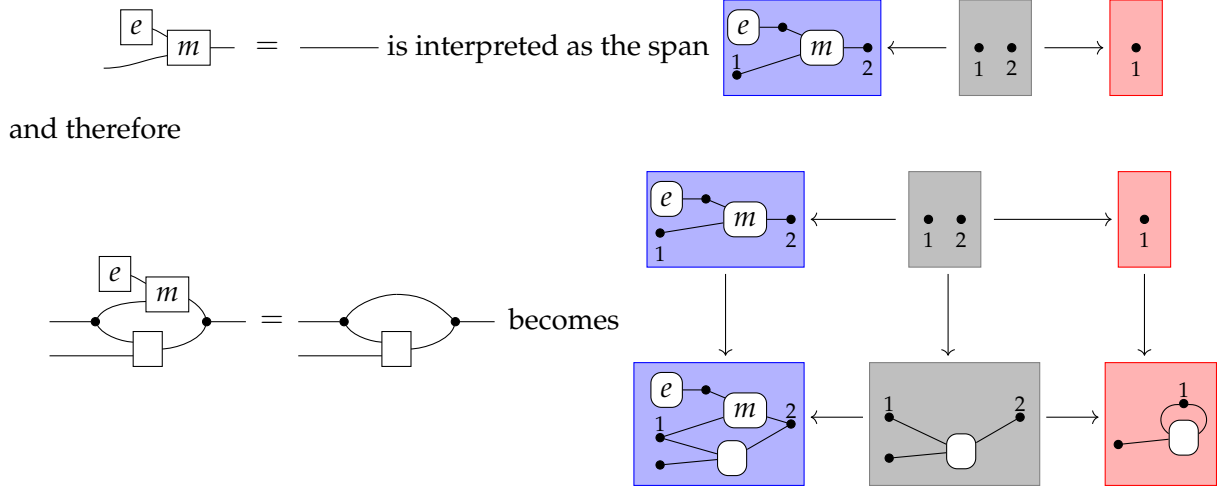


Figure 4: Example of how string diagram rewriting is interpreted as double-pushout rewriting. Intuitively, double-pushout rewriting matches the left-hand side of the rewrite rule to a subgraph and replaces it with the right-hand side. Here, the unitality rule of a monoid is applied in context.

Cartesian monoidal categories. It has been answered recently in [88] for copy-delete monoidal categories¹⁴, and in [64] for traced copy-delete categories. Finally, we point out the recent work [3], which studied rewriting for monoidal closed categories: classes of monoidal categories with a ‘function space’ which are relevant to the semantics of programming languages.

6.2 Higher-Dimensional Diagrams

The string diagrams we have presented in this survey are a particular case of a more general notation for higher-categories. It is helpful to highlight the connection. First, we need to explain what a n -category is. We will not do this here with any degree of rigour as this is not the topic of this survey. Instead, we will limit ourselves to giving some intuition and explain the link between their graphical language(s) and string diagrams. The reader should also know that there are several competing definitions of n -categories and that our lack of precision allows us to avoid committing to any one definition.

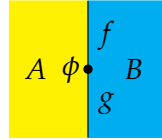
Very roughly, a n -category is a category that may have 2-morphisms between morphisms, 3-morphisms between 2-morphisms etc. For example, in a 2-category there can be 2-morphisms between morphisms:

$$A \begin{array}{c} \xrightarrow{f} \\ \Downarrow \phi \\ \xrightarrow{g} \end{array} B$$

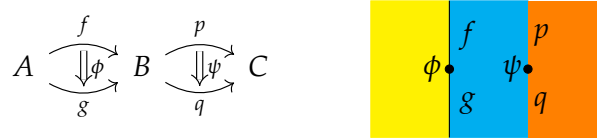
Like categories, 2-categories also admit a visual representation that generalises string diagrams, called *surface diagrams*: objects are represented as (labelled/coloured) regions of space, morphisms

¹⁴The work [57], which appeared at the same time as [88], also establishes the correspondence between string diagrams in copy-delete categories and suitably defined open hypergraphs, but without considering the rewriting angle.

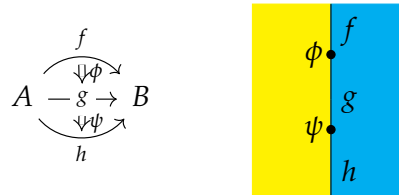
as (labelled) strings or wires, and 2-morphisms as (labelled) dots.



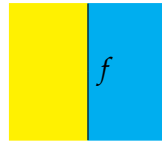
Let us see how these compose. There are two ways, just like there are two directions of composition for string diagrams: horizontally,



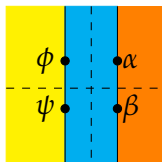
and vertically



The identity (1-)morphism $id_A : A \rightarrow A$ can be depicted as a single-coloured region of 2D space, while the identity 2-morphism $id_f : f \Rightarrow f$ can be depicted as a plain wire separating the domain and codomain objects of $f : A \rightarrow B$:



For these diagrams to make sense, horizontal and vertical compositions should satisfy additional associativity and unitality requirements, much like those of (1-)categories. In addition, for the diagrams to work, horizontal and vertical composition need to interact nicely; 2-categories should also verify a form of interchange law between horizontal and vertical composition. This law says that the two ways of decomposing the following diagram are equal:



There is a surprising correspondence between certain 2-categories and monoidal categories: a monoidal category is simply a 2-category with a single object! Take the 2-morphisms to be the ordinary morphisms of the corresponding monoidal category, the 1-morphisms to be its objects, and the monoidal product to be composition of 1-morphisms. Diagrammatically, we can simply depict the single object as the (white in this paper, but feel free to pick a nicer colour) background on which we draw our diagrams. In this sense, 2-categories can be thought of as typed monoidal categories (where the monoidal product cannot be applied uniformly, but has to match at the boundary 1-morphisms).

Note however that this correspondence is limited to monoidal categories, without any braiding or symmetry. To recover the ability to swap wires of symmetric monoidal categories a lot

more structure is required. Intuitively this is because, strictly speaking, if all we have are two dimensions of ambient space, wires cannot cross—what would that even mean? Braidings can occur in at least three dimensions where it makes sense to ask the question of which wire went over or under which other wire. This is why we need to move from 2-categories to 3-or more-categories. This may sound complicated, but just like 2-categories have morphisms between morphisms, we can define 3-categories, which have 3-morphisms between 2-morphisms, 4-categories, which have 4-morphisms between 3-morphisms etc. Each of these come with different ways of composing n -morphisms. As it turns out, braided or symmetric monoidal categories, and their graphical languages can be recovered as special cases of degenerate 3-and 4-categories, respectively. More precisely, a braided monoidal category is a 3-category with only a single object and (1-)morphism, while a symmetric monoidal category is a 4-category with a single object, 1-and 2-morphism. Phew! These correspondences are known as the *periodic table* of n -categories [7].

One last point: the category of categories is itself a 2-category in which objects are categories, (1-)morphisms are functors, and 2-morphisms are natural transformations between them. The graphical language of 2-categories can therefore be used to present key concepts in category theory. For an excellent introduction to category theory using this diagrammatic language (and an excellent introduction to the diagrammatic language of 2-categories itself) we warmly recommend [86].

6.3 Inequalities

In the same way that we can reason equationally about string diagrams (see Section 2.1), it is also possible to reason with *inequalities*. From the syntactic point of view, the changes are minimal: we can define theory with inequalities in the same way that we defined equational theories (equalities are recovered as two inequalities in both directions). To interpret inequalities requires a SMC with an order between the morphisms of the semantics that is coherent with the rest of the structure (composition, monoidal product etc.). More formally, we need a SMC in which the morphisms are partially ordered and for which the composition and monoidal product are monotone. This kind of structure appears naturally in the examples of relations that we have covered above, where morphisms can be ordered by inclusion: for two relations $R, S : X \rightarrow Y$, we write $R \leq S$ if R is included in S as a subset of $X \times Y$. If we look at inequalities, some fascinating structure starts to emerge.

Example 6.4 (Cartesian bicategories). Cartesian bicategories [27] are SMCs in which each object X has a distinguished monoid and comonoid structures, which we draw as $\frac{X}{X} \bullet \frac{X}{X}, \bullet \frac{X}{X}$ and $\frac{X}{X} \bullet \frac{X}{X}, \frac{X}{X} \bullet$, respectively. These have to satisfy the following additional axioms¹⁵:

$$\begin{array}{c} \text{) } \bullet \bullet \text{ (} \leq \text{ ——— } \quad \text{ ——— } \leq \text{ — } \bullet \text{ — } \bullet \text{ — } \quad \text{ — } \leq \text{ — } \bullet \text{ — } \bullet \text{ — } \quad \bullet \bullet \leq \square \end{array} \quad (32)$$

The SMC of monotone relations (Example 5.16) is an example of a Cartesian bicategory, with monoid-comonoid pair given by those of (27)-(26) for each pre-ordered set (X, \leq) . Note however that these, as we have seen, do not form a Frobenius structure in general. In fact, one can show that this is only the case when the underlying partial order is given by equality. In other words, the objects X in the category of monotone relations for which $\frac{X}{X} \bullet \frac{X}{X}, \bullet \frac{X}{X}$ and $\frac{X}{X} \bullet \frac{X}{X}, \frac{X}{X} \bullet$ form a Frobenius structure are just plain sets and monotone relations between them are just ordinary

¹⁵The categorically-minded reader will notice that these define an adjunction in the 2-categorical sense, between the comonoid and monoid structure, between $\text{—} \bullet \text{—}$ and $\text{—} \bullet \text{—}$ on the one hand, and $\text{—} \bullet$ and $\bullet \text{—}$ on the other.

relations! This precisely characterises the SMC of relations within the larger SMC of monotone relations.

We can characterise other well-known structures in this SMC. For example, we can require that there exists two more generators on the same object X —that we write $\frac{X}{X} \circ \frac{X}{X}$, $\circ \frac{X}{X}$ such that the dual of the inequalities (32) hold:

$$\frac{X}{X} \leq \frac{X}{X} \circ \frac{X}{X} \quad \frac{X}{X} \circ \frac{X}{X} \leq \frac{X}{X} \quad \frac{X}{X} \leq \frac{X}{X} \quad \frac{X}{X} \leq \frac{X}{X}$$

A set equipped with this structure in the SMC of monotone relations is precisely a semi-lattice whose binary meet and top can be identified with $\frac{X}{X} \circ \frac{X}{X}$ and $\circ \frac{X}{X}$ respectively. To get a lattice, we need to add $\frac{X}{X} \circ \frac{X}{X}$, $\frac{X}{X} \circ$ satisfying the same inequalities:

$$\frac{X}{X} \leq \frac{X}{X} \circ \frac{X}{X} \quad \frac{X}{X} \circ \frac{X}{X} \leq \frac{X}{X} \quad \frac{X}{X} \leq \frac{X}{X} \quad \frac{X}{X} \leq \frac{X}{X}$$

One might also wonder how $\frac{X}{X} \circ$, $\circ \frac{X}{X}$ and $\frac{X}{X} \circ \frac{X}{X}$, $\circ \frac{X}{X}$ interact. For an arbitrary lattice, there is not much one can say. However, when the lattice is a *Boolean algebra*, they form a commutative Frobenius structure!

6.4 Relationship with Proof Nets

Readers who have encountered proof nets before might wonder if there is a relationship with string diagrams. Given that proof nets are a graphical proof system for (multiplicative) linear logic [66] and that the natural categorical semantics for linear logic takes place in monoidal categories [87], there should be a connection between the two. However, classical multiplicative linear logic usually requires two different monoidal products: one for the multiplicative conjunction, usually written \otimes , and one for the multiplicative disjunction, usually written \wp . These have nontrivial interplay, axiomatised in the notion of linearly (or weakly) distributive category [32]; if we also want a classical negation, they are related via the usual DeMorgan duality, and the relevant semantics given by $*$ -autonomous categories [10].

The problem is that our diagrams already use two dimensions—one for the composition operation and one for the monoidal product. How should we deal with two monoidal products? There are different answers. The first (though not the first one historically) is to move one dimension up, from string diagrams in two-dimensional space, to surface diagrams in three-dimensional space. This is what the authors in [47] propose.

However, if we prefer to retain the typesetting ease of a two-dimensional notation, *proof nets* come to the rescue. Unlike standard string diagrams (for strict SMCs that is) proof nets include explicit generators for the two monoidal products¹⁶

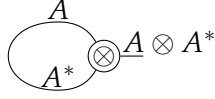
$$\begin{array}{c} A \\ \text{---} \otimes \text{---} B \\ \text{---} A \otimes B \end{array} \quad \begin{array}{c} A \\ \text{---} \wp \text{---} B \\ \text{---} A \wp B \end{array}$$

Like for compact closed categories, we also need cups and caps satisfying the usual snake equations, which the proof net literature tends to depict as undirected:

$$\begin{array}{c} A \\ \text{---} \cup \text{---} A^* \\ \text{---} A^* \end{array} \quad \begin{array}{c} A^* \\ \text{---} \cap \text{---} A \\ \text{---} A \end{array} \quad \text{s.t.} \quad \begin{array}{c} A^* \\ \text{---} \cup \text{---} A \\ \text{---} A^* \end{array} = \text{---} A^* \quad \begin{array}{c} A \\ \text{---} \cap \text{---} A^* \\ \text{---} A \end{array} = \text{---} A$$

¹⁶Note that, in the literature, proof nets are usually depicted going from top to bottom, but we prefer to maintain our convention here in order to make the link with string diagrams clearer.

However, not all diagrams we can draw in the free SMC over these generators are proof nets, in the sense they do not necessarily denote well-formed proofs in linear logic. For example, the following diagram is not a proof net and its conclusion $(A \otimes A^*)$ is not a theorem of linear logic:



This is why we need an additional criterion to distinguish correct proof nets among all the diagrams we are allowed to draw. There are several such criteria in the literature, under the name of *correctness criteria* [66, 39]. We will not cover these criteria here—the reader should just know that they usually boil down to detecting some form of acyclicity in graphs derived from the string diagram.

Proof nets can also be understood as a two-dimensional shadow of the natural three-dimensional notation used to represent both monoidal products in [47]. This projection comes at a cost: not all two-dimensional diagrams are shadows of a three-dimensional surface. Correctness criteria can therefore be seen as conditions guaranteeing that a proof net is the projection of some higher-dimensional surface diagram.

In the most degenerate cases (from the logic perspective), $\otimes = \wp$, and we are left with the diagrammatic language of compact closed categories (Section 4.2.2).

6.5 Software

With the spread of diagrammatic reasoning, it is natural to wonder to what extent it may be automated, and more generally how computers can assist humans in manipulating string diagrams. There are several tools dedicated to this task, each with their specific focus and area of predilection. Here is a (non-exhaustive) list.

- CARTOGRAPHER [100] deals with string diagrams for SMCs, the central concept of this introduction. It allows the user to specify arbitrary theories in this setting, and rewrite with them. String diagram rewriting is implemented as DPO hypergraph rewriting, following the approach of Section 6.1.
- CHYP (available at <https://github.com/akissinger/chyp>) is an interactive proof assistant for free SMCs over some signature and equational theory. The application works both with a conventional term syntax and with string diagrams. It also supports a hole-directed rewriting of terms (in the style of Agda).
- *DisCoPy* [41] is a Python library that defines a DSL for diagrams given by either, a free monoidal category or a free SMC over some signature. Furthermore, DisCoPy allows the user to define a semantics for diagrams, that is, to define functors out of free (symmetric) monoidal categories—through these, diagrams can be evaluated to some Python programs (as linear maps, for example). Note however that the package does not act as a proof assistant for diagrammatic equational theories, contrary to some of the other tools in the list.
- `homotopy.io` (itself the successor of a tool known as Globular [103]) is a more general tool that allows the user to construct finitely-generated n -categories. As a result, it is possible to encode string diagrams for SMCs into `homotopy.io`. However, the increased generality comes at the cost of significant sophistication: the user has to explicitly use the laws of SMCs in proofs, having to show the functoriality of the monoidal product by sliding two generators past each other, for example, instead of the two diagrams being equal.

- *Quantomatic* [79] is one of the earliest tools, which deals with a restricted subset of signatures (initially motivated by the ZX-calculus, see the paragraph on quantum physics in Section 7), namely signatures containing only commutative operations in compact closed categories. It allows the user to specify theories and rewriting strategies, as well as higher-order rules using so called !-boxes.

An important theoretical question for the above tools is which data structures best implement string diagrams and diagrammatic reasoning. Considering that string diagrams themselves are quotients of terms, it is a non-trivial task to represent their manipulation efficiently. This question has been explored recently in [107, 104].

7 String Diagrams in Science and Engineering: Some Applications

In the last few years, string diagrams have found application in several different fields of science and engineering. This section is intended as a succinct overview of such applications, with the main aim of providing to the reader references for a more focussed study. Clearly, a survey of this type cannot possibly begin to cover all the relevant material, and it will necessarily be a partial account. Our perspective will be pedagogical rather than historical: we will typically point to the most recent surveys and introductory materials, when available. A comprehensive literature review, including a rigorous reconstruction of “who did what first”, is out of our scope.

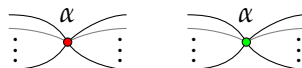
Many of the applications that we will describe share the same methodology. They involve noticing that the kind of systems and/or processes which constitute the focus of a given research area can be understood as the objects and/or morphisms of some SMC. This opens up the possibility of studying the topic from a functorial standpoint, using string diagrams as a syntax and the relevant systems and/or processes as semantics. In many cases the same approach also opens up the possibility of studying the equational properties of the resulting diagrammatic language. In some particular cases (this does not apply to all examples below), a universal set of generators can be found for the syntax and, if we are even luckier, the equational theory can be axiomatised by a complete monoidal theory.

It is fitting that we should start this section by applications of string diagrams in physics, since one of their ancestors were invented by Penrose to carry out the complex tensor calculations involved in the differential geometry of general relativity [94], *cf.* [93] for a more recent overview.

Quantum Physics. One of the most outstanding modern applications of diagrammatic reasoning has been to quantum computing.

Mathematically, quantum systems are modelled as Hilbert spaces, where joining two systems is represented by taking the tensor product of the respective spaces, and processes acting on a system are linear (unitary) maps. Linear maps between Hilbert spaces with the tensor product as monoidal product form a SMC, and are thus amenable to a string diagrammatic study.

There are several diagrammatic calculi to reason about linear maps between qubits (represented as vector spaces of dimension 2^d for some natural d) with the tensor product as monoidal product. These generalise and formalise the circuits representation that is ubiquitous in quantum computing. The first and most well-known such calculus is the *ZX-calculus*: its diagrams consist of nodes of two different colours (usually, green and red, but there are other conventions) called spiders, each labelled by an angle:



The two colours denote one of two classical observables or measurement bases—the computational and Hadamard basis respectively—and the angle denotes a phase relative to this basis, *i.e.* a rotation of the Bloch sphere along the axis determined by the chosen observable.

Equationally, the spiders form a special commutative Frobenius algebra and the two colours interact with each other to form bimonoid (more specifically, a Hopf algebra which is to a bimonoid what a group is to a monoid). Together with some other more complex equalities, the ZX-calculus completely axiomatises linear maps between qubits so that any semantic equality can be obtained by purely equational reasoning at the level of the diagrams themselves.

Because the ZX-calculus generalises quantum circuits, its axiomatisation provides a completely equational way to reason about those. Reasoning equationally at the level of circuits is much more difficult, so the ZX-calculus provides a more compositional setting in which to study the behaviour of circuits. In fact, one of its most successful applications has been to quantum circuit synthesis and simplification. Given some measure of complexity of circuits (usually based on the cost of implementing certain gates in a lab) one can compile a given circuit to its corresponding ZX diagram and simplify it using the axioms of the calculus, with the important caveat that one needs to guarantee that a bona fide circuit can be recovered at the end. (There are many technical papers on this topic; we could not find a more accessible survey, though the general introduction [102] contains some pointers to the literature.) String diagrams have now reached the mainstream quantum computing community, as even one of the founders of the field has adopted the ZX-calculus in a recent preprint [77].

There are other calculi with the same target semantics—linear maps between qubits—with different sets of generators as building blocks. The *ZW-calculus* was the first for which a completeness result was found and was instrumental in deriving a complete equational theory for the ZX-calculus (by translating one into the other). However, its generators are further removed from the conventional circuit gates that one can implement experimentally. The *ZH-calculus* is a variation of the ZX-calculus with which it is easier to represent certain multiply-controlled logic gates, like the Toffoli or AND gates (on the computational basis), and other related operations.

For a diagrammatic introduction to quantum computing and the foundations of quantum theory, the reference textbook is [34]. Alternatively, [69] provides a complementary (and more categorically minded) approach to some of the same topics. A comprehensive survey of the ZX-calculus (and its cousins) for the working computer scientist can be found in [102]. Beyond these, there is a wealth of recent developments that have taken the original work in different directions: a calculus which incorporates finite memory elements to the ZX-calculus [28], several calculi for quantum linear optical circuits [40, 30], and more... There are also automated tools to reason about large-scale ZX diagrams: the python library PyZX which we have already mentioned is the most recent [78]. Finally, the website <https://zxcalculus.com/> contains tutorials, a helpful guide to the publications in the field and even a map of ZX research community.

Signal flow graphs and control theory. Engineers and control theorists have long expressed causal flow of information between different components of a system by graphical means. One popular formalism is that of *signal flow graphs*, sometimes called *block diagrams*. They are directed graphs whose nodes represent system variables and edges functional dependencies between variables. They are used to represent networks of interconnected electronic components, amplifiers, filters etc. In signal flow graphs, cycles represent feedback between different parts of the system.

Giving signal flow graphs a functorial semantics, required a change of perspective: instead of only allowing *functional* dependencies between variables, we can generalise signal flow graphs to allow *relational* dependencies between them, *i.e.* arbitrary systems of (usually linear) equations.

This is entirely consistent with the underlying physics, where laws tend to express relationships between variables, without any explicit assumption about the direction of causality. This change of perspective allowed the reinterpretation of the fundamental building blocks of signal flow graphs as relations, their connection as relation composition, and their juxtaposition as taking the Cartesian product of the corresponding relations. In other words, these generalised signal flow graphs are string diagrams for a sub-category of the category of sets and relations! One important such subcategory is that of vector spaces over a field and *linear relations* – subset of the product that are also linear subspaces – between them. It turns out that signal flow graphs are intimately related to linear relations over the field of rational functions over \mathbb{R} . The connection was established independently in [20, 21] and [5], where the authors also give a complete equational theory, called *Interacting Hopf Algebra* (IH) by the first set of authors, to reason about the behaviour of these systems entirely diagrammatically. Since these early developments, the theory IH has been employed to reason algebraically about various tasks related to signal flow graphs: we mention the realisability of rational behaviours as circuits [20], semantic refinement [15], and a compositional criterion for controllability [53]. The interested reader should note however that there are some subtle discrepancies between this generalisation of signal flow graphs and the standard control-theoretic interpretation of their behaviour: an more accurate, but closely related semantics in terms of bi-infinite streams is given in [53], along with a complete equational theory.

Finally, linear relations are not just important because of the relationship with signal flow graphs; more generally the diagrammatic calculus and the equational theory IH provide a playground in which a substantial amount of standard linear algebra can be reformulated entirely diagrammatically. The blog `graphicallinearalgebra.net` is a great introduction to this topic, aimed at a general audience.

Circuit Theory. String diagrams are particularly compelling where they can give an algebraic foundation to existing graphical representations that are usually treated purely combinatorially. This is the case of many existing approaches to electrical or digital circuits. Despite the existence of a standard graphical representation for circuits, the string diagrammatic approach is not without challenges: taking the original graphical representation as a starting point, one needs to decompose them into a suitable set of generators from which all other circuits can be built and, more importantly, give this syntax a functorial interpretation that assigns to each circuit its intended behaviour. In traditional introductions to electrical circuits, this last step usually appeals informally to some intuitive connection between a circuit and the set of differential equations that it specifies. String diagrams can make this connection precise and compositional. In some particular cases, it is even possible to equip the resulting syntax with a complete equational theory that axiomatises semantic equivalence of circuits.

For *electrical circuits* with linear/affine behaviour (including resistors, inductors, capacitors, for example) this ambitious goal has not yet been achieved, though it is possible to compile them down to an intermediate representation in IH (or its affine extension) for which, as mentioned in the paragraph above, we do have a complete diagrammatic calculus. The case of circuits with passive components is treated in [6] while the extension to active components (current and voltage sources in AC regime only) is carried out in [17].

For *digital circuits*, there are many possible variants of interest to consider, each at their own level of abstraction (and each, with their own limitations). The simple case of acyclic circuits consisting only of logic gates, without any memory elements, reduces to boolean algebra and thus defines a Cartesian monoidal category, which can be presented by the symmetric monoidal version of the algebraic theory of boolean algebras, with only minor adaptations (as explained in

Section 4.2.5, in particular Remark 5.19)—details can be found in the pioneering [81].

More complex cases, involving delays or cycles are more delicate and no comprehensive account has ever been given. Recently, the sequential synchronous (*i.e.* where a global clock is assumed to define the time at which signals can meaningfully change) case has found a complete axiomatisation in [65]. Notably, this work also allows combinational (that is, without any delay) cycles in the syntax. This feature is usually avoided in traditional treatments of digital circuits because of the difficulty of handling these types of cycles compositionally. The case of asynchronous (cyclic) circuits remains elusive and an important open problem, although some work has been done in this direction [63].

Probability and statistics. Issues with the encoding of probability theory in set-theoretic measure theory have pushed researchers to develop a different, synthetic approach to probability theory. In recent years, some have sought alternative categorical foundations.

One approach studies categories of measurable spaces and Markov kernels (conditional probability measures) in an attempt to find an axiomatic setting for probability theory. This category is not only symmetric monoidal but a CD category. Moreover, in general, its morphisms satisfy only the (del) equation below:

$$\begin{array}{c} v \text{---} [d] \text{---} \bullet \begin{array}{l} \nearrow^w \\ \searrow_w \end{array} \end{array} \stackrel{\text{dup}}{=} \begin{array}{c} v \text{---} \bullet \begin{array}{l} \nearrow [d] \searrow^w \\ \searrow [d] \swarrow_w \end{array} \end{array} \quad \begin{array}{c} v \text{---} [d] \text{---} \bullet \end{array} \stackrel{\text{del}}{=} \bullet$$

At the semantic level, this equation simply means that conditional probabilities are measures that normalise to 1. Crucially, not all morphisms satisfy the (dup) equation above, so that categories of Markov kernels are not Cartesian; those morphisms that can be copied and do satisfy (dup) are precisely deterministic kernels, *i.e.* those that, given an element of their domain, map it to a single element in their codomain with probability one. We already see that a few elementary properties of random variables can be expressed in these categories, called *Markov categories*; their string diagrams for Markov categories give a graphical language to treat standard properties such as conditional independence, disintegration, almost sure properties, sufficient statistics and more. The interested reader will find [55] to be a good introduction to the topic, with applications to statistics.

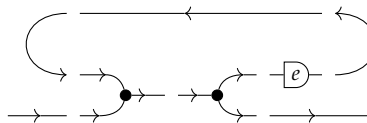
Much like the existing graphical methods of Bayesian networks and related representations, string diagrams make the flow of information between different variables explicit, highlighting structural properties, such as (conditional) independence. In fact, this connection was already explored in [49] which gave a functorial account of Bayesian networks. Around the same time, the authors of [36] proposed a diagrammatic calculus for Bayesian inference. Since then, a surprising amount of probability theory has been recast in this synthetic mold: a growing list of results have been reproven in this more general setting, many of which use string diagrams to streamline proofs, including zero-one laws [58], de Finetti theorem [56], the ergodic decomposition theorem [89], and more.

The same diagrams (in Markov or CD categories) also allow for a treatment of standard concepts in causal reasoning. In [71], the authors give a diagrammatic account of interventions and a sufficient criterion to identify when a given causal effect can be reliably estimated from observational data. The recent [82] extends this work to counterfactuals and shows how causal inference calculations can be carried out fully diagrammatically. Beyond its pedagogical value, one advantage of the diagrammatic approach is that it is axiomatic: as such, it is not restricted to the category of Markov kernels, but applies in all categories with the relevant structure.

Machine Learning with Neural Networks. Having mentioned string diagrammatic treatments of Bayesian networks, it is natural to wonder about analogous studies of neural networks, another chief graphical model of machine learning. Categorical approaches to these structures are fairly recent — see e.g. [99] for an overview — and so far have mainly focussed on providing a formal account of the gradient-based learning process — see in particular [54, 38, 61]. Use of string diagrams to describe the network structure only cursorily appear in [54]. String diagrams are heavily used in [38] to represent the categorical language of *lenses* in the context of machine learning, but presentations by generators and equations of these diagrams are not investigated. The works on gradient-based learning with “quantised” versions of neural networks, such as Boolean circuits [106] and polynomial circuits [108], adopt string diagrams in a more decisive way. These works define reverse derivatives compositionally on the diagrammatic syntax for circuits, building on the theory of reverse derivative categories [31] and Lafont’s algebraic presentation of Boolean circuits [81]. Going forward, the expectation is that such an approach may work also for real-valued networks, once the different neural network architectures are properly understood in terms of algebraic presentations. A starting point is provided in [54] for feedforward neural networks — see also the diagrammatic presentation of piecewise-linear relations found in [11], which may be used to model ReLu activation units. Another important research thread concerns automatic differentiation (AD): the work [4] uses rewriting of string diagrams in monoidal closed categories to describe an algorithm for AD and prove its soundness. In this context, string diagrams are appealing as they can be reasoned about as a high-level language, while at the same time exhibiting the same information of lower-level combinatorial formalisms, which in traditional AD are introduced via compilation. Finally, the webpage [60] is recommended as it maintains a list of papers at the intersection of category theory and machine learning.

Automata Theory. Automata have always been represented graphically, as state-transition graphs. However, the graphical representation is usually treated as a visual aid to convey intuition, not a formal syntax. Kleene introduced regular expressions to give finite-state automata an algebraic syntax. Their equational theory—under the name of Kleene algebra—is now well-understood and multiple complete axiomatisations have been given, for both language and relational models.

With string diagrams however, it is possible to go directly from the operational model of automata to their equational properties, without going through a symbolic algebraic syntax [95]. This approach lets us axiomatise the behaviour of automata directly, freeing us from the necessity of compressing them down to a one-dimensional notation like regular expressions. In addition, embracing the two-dimensional nature of automata guarantees a strong form of compositionality that the one-dimensional syntax of regular expressions does not have. In the string diagrammatic setting, automata may have multiple inputs and outputs and, as a result, can be decomposed into subcomponents that retain a meaningful interpretation. For example, the Kleene star can be decomposed into more elementary building blocks—generators of some monoidal signature—which come together to form a feedback loop:



A similar insight was already present in the work of Stefanescu [101] who studied the algebraic properties of traced monoidal categories (*cf.* Section 4.2.1) with several additional axioms in order to interpret automata, flowchart schemes, Petri nets, data-flow networks, and more.

Databases and Logic. As we have discussed above in several places, string diagrams are a convenient syntax for relations. They are particularly well-suited to *conjunctive queries*, the first-order language that contains relation symbols, equality, truth, conjunction, and existential quantification. This is a core fragment of query languages for relational databases with appealing theoretical property, such as NP-completeness (and thus, decidability) of query inclusion. Moreover, it admits a flexible diagrammatic language, which is exactly that of Cartesian bicategories (Example 6.4). These were introduced in [27] in the 1980s (without a diagrammatic syntax, most likely for typesetting reasons, though the authors give an axiomatisation that is easy to translate to string diagrams). The precise connection with conventional conjunctive query languages is worked out in [18]. More recently, the authors of [68] have generalised these diagrams to all of (classical) predicate logic. This requires adding boxes that represent negation to the diagrams of [27, 18].

Computability Theory. Computers are machines that can be programmed to exhibit a certain behaviour. The range of behaviours that they can exhibit (its processes) can be axiomatised into a monoidal category with additional properties: we require that every process the machine can perform has a name—its corresponding program—encoding the intentional content of the process. In turn, the category contains distinguished processes—called evaluators—which, given a program and an input state of the machine, runs the program on the given input. These simple requirements, with the ability to copy and delete data, are what [91] calls a *monoidal computer*. This structure is sufficient to reproduce a substantial chunk of computability (and complexity) theory using string diagrams. A textbook that does just that can be found online [92].

Concurrency. Concurrency lends itself to graphical methods, a fact noticed early by Petri. Initially, like so many other graphical representations, Petri nets were treated monolithically, and little attention was given to their composition. Once again, it is possible to take Petri nets seriously as a diagrammatic syntax with a functorial semantics. There are several ways to do so: one can either compose Petri nets along shared state (places) or shared actions (transitions). The former was developed in [8], while the latter was initiated in [25, 26]. The authors have contributed to developing this last approach further, by studying and axiomatising the algebra of Petri net transitions [16]. Significantly, the syntax is the same as that of signal flow graphs (see above)—only the semantics changes, replacing real numbers (modelling signals) with natural numbers (modelling non-negative finite resources like the tokens of Petri net). This simple change also changes the equational theory dramatically.

Linguistics. String diagrams have made a surprising appearance in formal linguistics and natural language processing. The core idea relies on a formal analogy between syntax and semantics of natural language. On the semantic side, vectors (in other words, arrays of numbers) are a convenient way of encoding statistical properties of words or features (*word embeddings*) extracted from large amounts of text by training machine learning model. The semantics obtained in this way is often called *distributional*. On the syntactic side, various formal structures of increasing complexity have been used to study the grammatical properties of languages and explain what distinguishes a well-formed sentence from an incorrect one.

Reconciling—or rather, combining the insights of—these two perspectives has been a long standing problem. One possible approach is premised on a formal correspondence between grammatical structure and distributional semantics: both fit into monoidal categories! We have already seen before that vector spaces and linear maps form a SMC, with the tensor as monoidal product. Similarly, formal grammars can be recast as certain (non-symmetric) monoidal categories in

which objects are parts-of-speech (think nouns, adjectives, transitive verbs etc.) and morphisms derivations of well-formed sentences. The analogy allows for a functorial mapping from one to the other. With this correspondence in place, it becomes possible to interpret grammatical derivations of sentences as string diagrams in the category of vector spaces and linear maps. This gives a compositional way to build the meaning of sentences from the individual meaning of words. Moreover, algebraic structures with the right properties can model grammatical features of language whose distributional semantics is less clear. Relative pronouns, for example, have been successfully interpreted as certain Frobenius algebras [96, 97], allowing equational reasoning about the meaning of sentences that contain them.

Game Theory. In classical game theory, games and their various solution concepts are usually studied monolithically. Remarkably, a compositional approach was shown possible in [62]. In this paper, the authors build games from smaller pieces, called *open games*. An open game is a component that chooses its next move strategically, given the state of its environment and some counterfactual reasoning about how the environment might react to its move (and what payoff it would derive from it). They form a SMC with an interesting diagrammatic syntax which contains forward and backward flowing wires (the latter represent this counterfactual reasoning, flowing from future to present). In fact, these diagrams are related to those for *lenses*, which we have encountered in the paragraph on machine learning. Closed diagrams represent classical games, whose semantics is given by the appropriate equilibrium condition. Initially developed only for standard games with pure Nash equilibria as solution concept, open games have been extended to more general settings, such as Bayesian games [12] with the appropriate solution concept.

References

- [1] Samson Abramsky. Retracing some paths in process algebra. In *CONCUR'96 Proceedings*, pages 1–17. Springer, 2005.
- [2] Samson Abramsky. No-cloning in categorical quantum mechanics. *Semantic Techniques in Quantum Computation*, pages 1–28, 2009.
- [3] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Rewriting for monoidal closed categories. In *FSCD'22 Proceedings*, volume 228 of *LIPICs*, pages 29:1–29:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [4] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. In *CSL'23 Proceedings*, volume 252 of *LIPICs*, pages 6:1–6:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
- [5] John Baez and Jason Erbele. Categories in control. *Theory and Applications of Categories*, 30:836–881, 2015.
- [6] John C Baez, Brandon Coya, and Franciscus Rebro. Props in network theory. *Theory and Applications of Categories*, 33(25):727–783, 2018.
- [7] John C Baez and James Dolan. Higher-dimensional algebra and topological quantum field theory. *Journal of mathematical physics*, 36(11):6073–6105, 1995.
- [8] John C. Baez and Jade Master. Open petri nets. *Math. Struct. Comput. Sci.*, 30(3):314–341, 2020.

- [9] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.
- [10] Michael Barr. **-Autonomous categories*, volume 752. Springer, 2006.
- [11] Guillaume Boisseau and Robin Piedeleu. Graphical piecewise-linear algebra. In *FOS-SACS'22 (ETAPS) Proceedings*, pages 101–119. Springer International Publishing Cham, 2022.
- [12] Joe Bolt, Jules Hedges, and Philipp Zahn. Bayesian open games. *arXiv preprint arXiv:1910.03656*, 2019.
- [13] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. Rewriting modulo symmetric monoidal structure. In *LICS'16 Proceedings*, pages 710–719. IEEE, 2016.
- [14] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobocinski, and Fabio Zanasi. String diagram rewrite theory I: rewriting with Frobenius structure. *J. ACM*, 69(2):14:1–14:58, 2022.
- [15] Filippo Bonchi, Joshua Holland, Dusko Pavlovic, and Paweł Sobociński. Refinement for signal flow graphs. In *CONCUR'17 Proceedings*, pages 24:1–24:16, 2017.
- [16] Filippo Bonchi, Joshua Holland, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Diagrammatic algebra: from linear to concurrent systems. *POPL'19 Proceedings*, 3:1–28, 2019.
- [17] Filippo Bonchi, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Graphical affine algebra. In *LICS'19 Proceedings*, pages 1–12, 2019.
- [18] Filippo Bonchi, Jens Seeber, and Paweł Sobocinski. Graphical conjunctive queries. In *CSL'18 Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- [19] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Interacting bialgebras are Frobenius. In *FOSSACS'14 Proceedings*, volume 8412 of *LNCS*, pages 351–365. Springer Berlin Heidelberg, 2014.
- [20] Filippo Bonchi, Paweł Sobocinski, and Fabio Zanasi. Full abstraction for signal flow graphs. In *POPL'15 Proceedings*, volume 50 of *ACM Sigplan Notices*, pages 515–526. ACM, 2015.
- [21] Filippo Bonchi, Paweł Sobocinski, and Fabio Zanasi. The calculus of signal flow diagrams I: linear relations on streams. *Information and Computation*, 252:2–29, 2017.
- [22] Filippo Bonchi, Paweł Sobocinski, and Fabio Zanasi. Interacting Hopf algebras. *Journal of Pure and Applied Algebra*, 221(1):144–184, 2017.
- [23] Filippo Bonchi, Paweł Sobocinski, and Fabio Zanasi. Deconstructing Lawvere with distributive laws. *J. Log. Algebraic Methods Program.*, 95:128–146, 2018.
- [24] Roberto Bruni and Fabio Gadducci. Some algebraic laws for spans (and their connections with multirelations). In *RelMiS'01 Proceedings, ENTCS*, volume 44. Elsevier, 2001.
- [25] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Connector algebras, Petri nets, and BIP. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 19–38. Springer, 2011.

- [26] Roberto Bruni, Hernán C. Melgratti, Ugo Montanari, and Paweł Sobociński. Connector algebras for C/E and P/T nets' interactions. *Log Meth Comput Sci*, 9(16), 2013.
- [27] Aurelio Carboni and R. F. C. Walters. Cartesian bicategories I. *Journal of Pure and Applied Algebra*, 49:11–32, 1987.
- [28] Titouan Carette, Marc De Visme, and Simon Perdrix. Graphical language with delayed trace: Picturing quantum computing with finite memory. In *LICS'21 Proceedings*, pages 1–13. IEEE, 2021.
- [29] Eugenia Cheng. Iterated distributive laws. *Math. Proc. Camb. Philos. Soc.*, 150(3):459–487, 2011.
- [30] Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix, and Benoît Valiron. LOv-calculus: A graphical language for linear optical quantum circuits. In *MFCS'22 Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- [31] J. Robin B. Cockett, Geoff S. H. Crutwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon D. Plotkin, and Dorette Pronk. Reverse derivative categories. In *CSL'20 Proceedings*, volume 152 of *LIPICs*, pages 18:1–18:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [32] J Robin B Cockett and Robert AG Seely. Weakly distributive categories. *Journal of Pure and Applied Algebra*, 114(2):133–173, 1997.
- [33] Bob Coecke and Ross Duncan. Interacting quantum observables. In *ICALP'08 Proceedings, Part II*, pages 298–310, 2008.
- [34] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes - A first course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.
- [35] Bob Coecke, Dusko Pavlovic, and Jamie Vicary. A new description of orthogonal bases. *Math. Struct. Comp. Sci.*, 23(3):557–567, 2012.
- [36] Bob Coecke and Robert W Spekkens. Picturing classical and quantum bayesian inference. *Synthese*, 186:651–696, 2012.
- [37] B Coya and Brendan Fong. Corelations are the prop for extraspecial commutative Frobenius monoids. *Theory and Applications of Categories*, 32(11):380–395, 2017.
- [38] Geoffrey S. H. Crutwell, Bruno Gavranovic, Neil Ghani, Paul W. Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning. In *ESOP'22*, volume 13240 of *LNCS*, pages 1–28. Springer, 2022.
- [39] Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, 1989.
- [40] Giovanni De Felice and Bob Coecke. Quantum linear optics via string diagrams. *arXiv:2204.12985*, 2022.
- [41] Giovanni de Felice, Alexis Toumi, and Bob Coecke. DisCoPy: Monoidal categories in python. *Electronic Proceedings in Theoretical Computer Science*, 333:183–197, feb 2021.

- [42] Elena Di Lavore, Giovanni de Felice, and Mario Román. Monoidal streams for dataflow programming. In *LICS'22 Proceedings*, pages 1–14, 2022.
- [43] Elena Di Lavore, Alessandro Gianola, Mario Román, Nicoletta Sabadini, and Paweł Sobociński. A canonical algebra of open transition systems. In *FACS'21 Proceedings 17*, pages 63–81. Springer, 2021.
- [44] Lucs Dixon and Aleks Kissinger. Open-graphs and monoidal theories. *Mathematical Structures in Computer Science*, 23(2):308–359, 2013.
- [45] Ross Duncan and Kevin Dunne. Interacting Frobenius algebras are Hopf. In *LICS'16*, pages 535–544, 2016.
- [46] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John Van De Wetering. Graph-theoretic simplification of quantum circuits with the ZX-calculus. *Quantum*, 4:279, 2020.
- [47] Lawrence Dunn and Jamie Vicary. Coherence for Frobenius pseudomonoids and the geometry of linear proofs. *Logical Methods in Computer Science*, 15, 2019.
- [48] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *SWAT'73 Proceedings*, pages 167–180. IEEE, 1973.
- [49] Brendan Fong. Causal theories: A categorical perspective on Bayesian networks. Master's thesis, Univ. of Oxford, 2012. arXiv:1301.6201.
- [50] Brendan Fong. Decorated cospans. *Theory and Applications of Categories*, 30(33):1096–1120, 2015.
- [51] Brendan Fong. *The Algebra of Open and Interconnected Systems*. PhD thesis, University of Oxford, 2016. arXiv:1609.05382.
- [52] Brendan Fong. Decorated corelations. *Theory & Applications of Categories*, 33, 2018.
- [53] Brendan Fong, Paweł Sobociński, and Paolo Rapisarda. A categorical approach to open and interconnected dynamical systems. In *LICS'16 Proceedings*, pages 495–504, 2016.
- [54] Brendan Fong, David I. Spivak, and Rémy Tuyéras. Backprop as functor: A compositional perspective on supervised learning. In *LICS'16 Proceedings*, pages 1–13. IEEE, 2019.
- [55] Tobias Fritz. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Advances in Mathematics*, 370:107239, 2020.
- [56] Tobias Fritz, Tomáš Gonda, and Paolo Perrone. De finetti's theorem in categorical probability. *Journal of Stochastic Analysis*, 2(4):6, 2021.
- [57] Tobias Fritz and Wendong Liang. Free gs-monoidal categories and free Markov categories. *Applied Categorical Structures*, 31(2):21, 2023.
- [58] Tobias Fritz and Eigil Fjeldgren Rischel. The zero-one laws of Kolmogorov and Hewitt–Savage in categorical probability. *Compositionality*, 2:3, 2020.
- [59] Fabio Gadducci and Reiko Heckel. An inductive view of graph transformation. In *WADT'97 Proceedings*, pages 223–237, 1997.

- [60] Bruno Gavranovic. Category theory \cap machine learning. https://github.com/bgavran/Category_Theory_Machine_Learning. Accessed: 2023-09-06.
- [61] Bruno Gavranovic. Compositional deep learning. *arXiv:1907.08292*, 2019.
- [62] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. Compositional game theory. In *LICS'18 Proceedings*, pages 472–481, 2018.
- [63] Dan R. Ghica. Diagrammatic reasoning for delay-insensitive asynchronous circuits. In *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, pages 52–68. Springer, 2013.
- [64] Dan R Ghica and George Kaye. Rewriting modulo traced comonoid structure. *arXiv:2302.09631*, 2023.
- [65] Dan R Ghica, George Kaye, and David Sprunger. Full abstraction for digital circuits. *arXiv:2201.10456*, 2022.
- [66] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
- [67] Masahito Hasegawa. Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. In *TLCA'97 Proceedings*, pages 196–213. Springer, 1997.
- [68] Nathan Haydon and Paweł Sobociński. Compositional diagrammatic first-order logic. In *Diagrams'20 Proceedings*, pages 402–418. Springer, 2020.
- [69] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: an introduction*. Oxford University Press, 2019.
- [70] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. In *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*, volume 172 of *Electronic Notes in Theoretical Computer Science*, pages 437–458. Elsevier, 2007.
- [71] Bart Jacobs, Aleks Kissinger, and Fabio Zanasi. Causal inference via string diagram surgery: A diagrammatic approach to interventions and counterfactuals. *Mathematical Structures in Computer Science*, 31(5):553–574, 2021.
- [72] André Joyal and Ross Street. Braided monoidal categories. *Mathematics Reports*, 86008, 1986.
- [73] André Joyal and Ross Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, 1991.
- [74] Piergiulio Katis, Nicoletta Sabadini, and Robert FC Walters. Feedback, trace and fixed-point semantics. *RAIRO-Theoretical Informatics and Applications*, 36(2):181–194, 2002.
- [75] G Maxwell Kelly. Many-variable functorial calculus. I. In *Coherence in categories*, pages 66–105. Springer, 1972.
- [76] Gregory M Kelly and Miguel L Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.
- [77] Andrey Boris Khesin, Jonathan Z Lu, and Peter W Shor. Graphical quantum Clifford-encoder compilers from the ZX calculus. *arXiv:2301.02356*, 2023.

- [78] Aleks Kissinger and John van de Wetering. PyZX: Large scale automated diagrammatic reasoning. *arXiv:1904.04735*, 2019.
- [79] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. *CoRR*, abs/1503.01034, 2015.
- [80] Stephen Lack. Composing PROPs. *Theory and Application of Categories*, 13(9):147–163, 2004.
- [81] Yves Lafont. Towards an algebraic theory of Boolean circuits. *Journal of Pure and Applied Algebra*, 184(2–3):257–310, 2003.
- [82] Robin Lorenz and Sean Tull. Causal models in string diagrams. *arXiv:2304.07638*, 2023.
- [83] Saunders Mac Lane. Categorical algebra. *Bulletin of the American Mathematical Society*, 71:40–106, 1965.
- [84] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.
- [85] Dan Marsden and Fabrizio Genovese. Custom hypergraph categories via generalized relations. In *CALCO’17 Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- [86] Daniel Marsden. Category theory using string diagrams. *arXiv:1401.7220*, 2014.
- [87] Paul-André Mellies. Categorical semantics of linear logic. *Panoramas et syntheses*, 27:15–215, 2009.
- [88] Aleksandar Milosavljevic and Fabio Zanasi. String diagram rewriting modulo commutative monoid structure. *To appear in CALCO’23 Proceedings*, arXiv:2204.04274, 2023.
- [89] Sean Moss and Paolo Perrone. A category-theoretic proof of the ergodic decomposition theorem. *arXiv preprint arXiv:2207.07353*, 2022.
- [90] Koko Muroya and Dan R Ghica. The dynamic geometry of interaction machine: A call-by-need graph rewriter. *CSL’17 Proceedings*, 2017.
- [91] Dusko Pavlovic. Monoidal computer I: Basic computability by string diagrams. *Information and computation*, 226:94–116, 2013.
- [92] Dusko Pavlovic. Categorical computability in monoidal computer: Programs as diagrams. *arXiv:2208.03817*, 2022.
- [93] R. Penrose. Applications of negative dimension tensors. In D. J. A. Welsh, editor, *Combinatorial Mathematics and its Applications*, pages 221–244. Academic Press, 1971.
- [94] Roger Penrose. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications*, 1:221–244, 1971.
- [95] Robin Piedeleu and Fabio Zanasi. A finite axiomatisation of finite-state automata using string diagrams. *Logical Methods in Computer Science*, 19, 2023.
- [96] Mehrnoosh Sadrzadeh, Stephen Clark, and Bob Coecke. The Frobenius anatomy of word meanings I: subject and object relative pronouns. *Journal of Logic and Computation*, 23(6):1293–1317, 2013.

- [97] Mehrnoosh Sadrzadeh, Stephen Clark, and Bob Coecke. The Frobenius anatomy of word meanings II: possessive relative pronouns. *Journal of Logic and Computation*, 26(2):785–815, 2014.
- [98] Peter Selinger. A survey of graphical languages for monoidal categories. *Springer Lecture Notes in Physics*, 13(813):289–355, 2011.
- [99] Dan Shiebler, Bruno Gavranovic, and Paul W. Wilson. Category theory in machine learning. *arXiv:2106.07032*, 2021.
- [100] Paweł Sobociński, Paul W Wilson, and Fabio Zanasi. Cartographer: A tool for string diagrammatic reasoning. In *CALCO’19 Proceedings*, page 25, 2019.
- [101] G. Stefanescu. *Network Algebra*. Discrete Mathematics and Theoretical Computer Science. Springer London, 2000.
- [102] John van de Wetering. ZX-calculus for the working quantum computer scientist. *arXiv:2012.13966*, 2020.
- [103] Jamie Vicary, Aleks Kissinger, and Krzysztof Bar. Globular: an online proof assistant for higher-dimensional rewriting. *Logical Methods in Computer Science*, 14, 2018.
- [104] Paul Wilson and Fabio Zanasi. Data-parallel algorithms for string diagrams. *arXiv:2305.01041*, 2023.
- [105] Paul W. Wilson, Dan R. Ghica, and Fabio Zanasi. String diagrams for non-strict monoidal categories. In *CSL*, volume 252 of *LIPICs*, pages 37:1–37:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
- [106] Paul W. Wilson and Fabio Zanasi. Reverse derivative ascent: A categorical approach to learning boolean circuits. In *ACT’20 Proceedings*, volume 333 of *EPTCS*, pages 247–260, 2020.
- [107] Paul W. Wilson and Fabio Zanasi. The cost of compositionality: A high-performance implementation of string diagram composition. In *ACT’21 Proceedings*, volume 372 of *EPTCS*, pages 262–275, 2021.
- [108] Paul W. Wilson and Fabio Zanasi. Categories of differentiable polynomial circuits for machine learning. In *ICGT*, volume 13349 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2022.
- [109] Fabio Zanasi. *Interacting Hopf Algebras: the theory of linear systems*. PhD thesis, Ecole Normale Supérieure de Lyon, 2015.
- [110] Fabio Zanasi. The algebra of partial equivalence relations. *Electronic Notes in Theoretical Computer Science*, 325:313–333, 2016.